Reconfigurable Conveyor Transfer System using IEC-61499 Function Blocks

Leonardo de Oliveira Souza Mendonça

Dissertation presented to the School of Technology and Management of Bragança to obtain the Master's Degree in Electrical and Computer Engineering

Under the supervision of:

Professor PhD Paulo Leitão

Professor PhD Frederico Fagundes

Bragança

2024

Reconfigurable Conveyor Transfer System using IEC-61499 Function Blocks

Leonardo de Oliveira Souza Mendonça

Dissertation presented to the School of Technology and Management of Bragança to obtain the Master's Degree in Electrical and Computer Engineering in the scope of the double diploma program with the Federal Center for Technological Education of Minas Gerais.

Under the supervision of:

Professor PhD Paulo Leitão

Professor PhD Frederico Fagundes

Bragança

2024

Dedication

I dedicate this work to my family, especially my parents, my sister, and my girlfriend, all of whom supported me to get to where I am and finish this work.

Acknowledgements

Here I would like to thank all those who contributed in some way to the development of this work.

First of all, I would like to thank the organizers of the Double Degree program who made the collaboration between the university in Brazil, CEFET-MG, and the university in Portugal, IPB, possible. This program gave me the opportunity to have an incredible experience, sharing and living moments that will always be remembered.

I would also like to thank my supervisors, Professor PhD Paulo Leitão from IPB and Professor PhD Frederico Fagundes from CEFET-MG, as well as PhD students Gustavo Funchal and Victória Melo, for the guidance, opportunities, challenges, and knowledge generated. They were responsible for my motivation in this research, always getting the most out of me and believing in my potential.

I would like to thank all the friends I made during this period of study, especially the members of Republica Bairro Nobre, Bruno, Edilson, Lucas and Vinicius, with whom I shared my home during this year and who undoubtedly formed bonds of brotherhood. Without them, all this work would have been extremely difficult and I am very grateful to all of them.

II would especially like to thank my girlfriend Ana Paula, the woman I love and who I was very lucky to meet this year, always believing in my potential. She is a fundamental part of my being able to complete all my work, with all the support, affection and love I needed to get through the difficult times, as well as all the celebrations for my achievements.

Finally, I would like to thank my mother, Maria Eunice, my father, Wellington, and my sister, Laís, who gave me all the support and encouragement I needed to pursue my dreams and achieve the things I wanted. Even though we are very far apart, I know that they have always cheered me on and believed in my success. Nothing could have been done without them.

Abstract

In the 4th Industrial Revolution era, automation processes operate in a more modular, flexible, and reconfigurable manner, leveraging decentralized decision-making and distributed control. As the backbone of this revolution lies the Cyber-physical Production System (CPPS) concept, which emphasizes the strong interlink and interdependence of the digital/cyber and physical components. It requires the use of a technology capable of incorporating intelligence through the entities in a decentralized manner. IEC-61499 Function Blocks technology stands out as a suitable approach to implementing distributed automation control systems.

This work presents the development of a dynamic and on-the-fly reconfiguration conveyor transfer system based on the IEC-61499 standard, that automatically adapts its operation to face changes in the number and position of the conveyor modules. For this purpose, the developed control system considers a Python-based Function Block (FB) network using the DINASORE framework, capable of effectively communicating between system components and identifying changes in real time.

The experimental tests showed promising results regarding the system's ability to adapt to condition changes automatically and on the fly, as well as its scalability and robustness.

Keywords: Industry 4.0, Cyber-physical Systems, Dynamic Reconfiguration, Function Blocks, IEC-61499 Standard, DINASORE Framework.

Resumo

Na era da 4ª Revolução Industrial, os processos de automação operam de maneira mais modular, flexível e reconfigurável, aproveitando a tomada de decisão descentralizada e o controle distribuído. O conceito de Sistemas de Produção Cíber Físicos (do inglês Cyber-physical Production System - CPPS) surge como o principal componente dessa revolução, enfatizando a forte interligação e interdependência dos componentes digitais/cibernéticos e físicos, exigindo o uso de uma tecnologia capaz de incorporar inteligência por meio das entidades de forma descentralizada. A tecnologia de Function Blocks baseados na norma IEC-61499 destaca-se como uma abordagem adequada para implementar sistemas de controle de automação distribuídos.

Este trabalho apresenta o desenvolvimento de um sistema dinâmico de transferência de esteira transportadora e de reconfiguração baseado na norma IEC-61499, adaptando automaticamente sua operação para enfrentar mudanças no número e na posição dos módulos da esteira. Para isso, o sistema de controle desenvolvido considera uma rede de FB baseada em Python usando o DINASORE, capaz de se comunicar efetivamente entre os componentes do sistema e identificar alterações em tempo real.

Os testes experimentais mostraram resultados promissores com relação à capacidade do sistema de se adaptar às mudanças automaticamente e em tempo real, bem como sua escalabilidade e robustez.

Palavras-chave: Indústria 4.0, Sistemas Ciberfísicos, Reconfiguração Dinâmica, Function Blocks, Norma IEC-61499, DINASORE Framework.

Contents

List of Tables

List of Figures

Acronyms

API Application Programming Interfaces

CeDRI Research Centre in Digitalization and Intelligent Robotics

CPPS Cyberphysical Production System

DCS Distributed Control Systems

DINASORE Dynamic Intelligent Architecture for Software and Modular Reconfiguration

FB Function Block

IDE Integrated Development Environment

IEC International Electrotechnical Commission

IIoT Industrial Internet of Things

IL Instruction List

JSON JavaScript Object Notation

LD Ladder Diagram

MAS Multi-Agent System

MQTT Message Queuing Telemetry Transport

MSECC Master-Slave Execution Control Chart

OPC UA Open Platform Communications Unified Architecture

PCP Priority Ceiling Protocol

PLC Programmable Logic Controller

RAFAH Reconfiguration Architecture for Fault Handling

RCA Reconfiguration Application

RFBs Reconfigurable Function Blocks

SFC Sequential Function Chart

ST Structured Text

Chapter 1

Introduction

This work is framed in the context of Industry 4.0, focusing on the concepts of "Reconfigurability", "FB based on International Electrotechnical Commission (IEC)-611499 standard" and "Cyberphysical Production System (CPPS)". These key technological areas are at the forefront of promoting a transformative combination of the physical and computational domains. The transition towards distributed intelligence and system decentralization precipitates notable enhancements across various sectors, promoting efficiency, resilience, and adaptability improvements.

1.1 Motivation

The 4th Industrial Revolution, represented by Industry 4.0, introduces a new era in the manufacturing sector, driven by the integration of digital technologies, intelligent automation, and interconnected systems. Therefore, to meet the growing demands of production and adapt to emergent technologies, implementing Industry 4.0 requires increased autonomy through decentralized decision-making and distributed control across the shop floor, enabling a more cooperative and flexible production environment [1].

At the core of this revolution lies the CPPS concept, which is a new structure that emphasizes the strong interlink and interdependence of the digital/cyber and physical components, decentralizing and distributing the computation entities among a mesh network of nodes and subsystems [2], [3].

With this transition, there has been some notable transformation in various crucial aspects: i) portability means the engineering support systems can accept and correctly interpret software components and system configurations produced by other engineering

tools, ii) interoperability means that hardware devices can operate together to perform the cooperative functions specified by one or more distributed applications, and iii) configurability means that multiple engineering tools can dynamically configure field devices and their software components [4], [5].

To ensure optimal functionality, the control logic behind the automation process requires standardization. The IEC-61499 standard [4] provides a highly accessible framework for implementing distributed automation control systems, where at the core of this framework lies the FBs, modular components that encapsulate specific control functionalities. This allows for the seamless integration of diverse controllers from different manufacturers into a single program and improves some aspects such as portability, reconfigurability, and interoperability. Moreover, effective communication is imperative in automation systems, necessitating a mediator capable of facilitating message exchange.

The most prevalent methods for message exchange in such systems are the Open Platform Communications Unified Architecture (OPC UA) and the Message Queuing Telemetry Transport (MQTT). The utilization MQTT proves advantageous due to its lightweight and efficient messaging protocol, with a scalable publish-subscribe architecture overcoming interoperability problems in connected systems with different levels of connectivity through its integration capability [6]. However, even the OPC UA standing as the most widely used protocol in the industry, MQTT also offers protocols that are well-suited for Industrial Internet of Things (IIoT) environments, characterized by resource constraints and the continuously evolving nature of the system environment [7], [8].

Therefore, the grand dissemination of the IEC-61499 standard in the industrial environment is due to the easy implementation of such modular systems, allowing the creation of FB that meet the system's specific needs. However, identifying and correcting problems in an FB system can be challenging, especially for reconfigurable systems that need to identify and analyze these problems in real-time to make the best decisions to achieve the system's objectives.

Having this in mind, this work presents the application of the Python-based FB approach to developing a dynamically reconfigurable transfer conveyor system based on Fischetechnik conveyor modules. For this purpose, the developed solution uses the Dynamic Intelligent Architecture for Software and Modular Reconfiguration (DINASORE) [9]. This implementation forms a decentralized control network within the conveyor system, communicating through the MQTT broker and allowing the development of an intelligent decision-making mechanism that adapts dynamically to changes in the conveyor modules.

1.2 Objectives

The main objective of this work is to develop FB based on the IEC-61499 standard that allows the application of a self-organizing system to adapt dynamically to changes and support the addition, removal, and position change in real-time.

In order to achieve this objective, secondary objectives have been established:

- Study and understanding of the FB structure based on the IEC-61499 standard;
- Analysis of the behavior and operation of the conveyor system made up of FischerTechniks conveyors modules;
- Development of essential FBs for data transmission;
- Development of a FB responsible for the control and implementation of self-organization;
- Analysis of system performance, considering self-organization and scalability.

1.3 Document Structure

This document is organized into six chapters, beginning with the present chapter, which introduces the problem and contextualizes it, as well as the objectives of the work.

The second chapter, "State of the Art", provides relevant concepts and information about the theoretical background, which is essential for developing and understanding the work.

The third chapter, "Experimental Case Study," presents an overview of the system under study, emphasizing its cyber-physical components and the behaviors around the system to implement self-organization.

The fourth chapter, titled "Implementation of the Reconfigurable Control System," provides an overview of the system's structure, focusing on the control structure designed specifically for a single conveyor. It highlights the communication architecture and reconfigurable control mechanisms employed in creating the control FBs. Furthermore, it offers a detailed representation of the FB network and describes the process of integrating the system into the conveyor modules.

The fifth chapter presents the results obtained when applying the system's self-organization, presenting the tests carried out focusing on adding, removing, and changing the position

of the conveyor. It also graphically presents the results obtained in the system's scalability and performance tests to validate its operation.

Finally, the sixth chapter, entitled "Conclusions and Future Work", resumes the work with the conclusions and points out future work.

The document also contains an appendix that comprises the codes developed in this work.

Chapter 2

State of The Art

This chapter provides an overview of key concepts central to the research, presenting an introduction to Industry 4.0 and to CPPS. It subsequently explores a comparison between the IEC-61131 and IEC-61499 standards, highlighting the latter's advancements and elucidating the construction and functionality of FB within this updated standard. The chapter also engages in discussions on related works in the research field. Furthermore, it offers insights into the software and technologies utilized in the project, including the Python-based DINASORE framework and the 4DIAC Integrated Development Environment (IDE).

2.1 The Industry advancement

The advent of the 4th Industrial Revolution, known as Industry 4.0, signifies a profound transformation in manufacturing, extending its influence beyond the industry to various manufacturing sectors. To comprehend the scope of this revolution, it is imperative to delve into its historical antecedents and evolutionary phases, as depicted in Figure 2.1. This journey begins with the mechanization of labor through steam power during the first industrial revolution of the 18th century, followed by the electrification-driven mass production of the second revolution. The third revolution inaugurated digitalization and automation, with electronic and computer technology playing pivotal roles [10], [11].

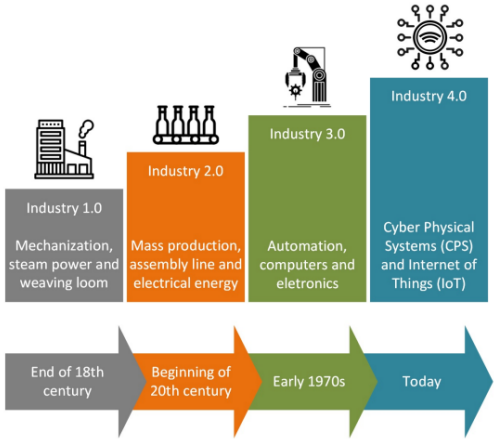


Figure 2.1: Evolution of the industry [12].

Therefore, the 4th Industrial Revolution introduced a new era in the manufacturing sector, driven by the integration of digital technologies, intelligent automation, and interconnected systems. The implementation of Industry 4.0 requires decentralized decision-making and distributed control across the shop floor, enabling a more cooperative and flexible production environment [1].

Furthermore, industrial systems can be successfully implemented by connecting the industrial assets, as shown in Figure 2.2, including networked smart objects, cyber-physical assets, associated generic information technologies, and optimal cloud or edge computing platforms [13]. As a result, the large amount of collected data can feed analytical solutions, enabling the real-time and intelligent analysis of data, improving the communication and collaboration between components belonging to the industrial process, leading to optimal industrial operations [13], [14].

Figure 2.2: Key concepts of the industry 4.0 (adapted from [10]).

At the core of this new perspective lies the CPPS concept, which is a new structure that emphasizes the strong interlink and interdependence of the digital/cyber and physical components, decentralizing and distributing the computation entities among a mesh network of nodes and subsystems [2], [3]. Furthermore, these systems can interact with humans and be engaged in machine-to-machine communication, enabling them to make decisions and optimize the production system, improving the system's overall performance by focusing on efficiency, agility, and adaptability while also addressing the various complexities, uncertainties, and operational dynamics that are common in new environments [12], [15].

2.2 Function Block Standards in Industry

To ensure the optimal functionality, the control logic behind the automation process needs to be standardized, which was initially possible by following the IEC-61131 standard [16], which provides the guidelines and directives of how industrial control applications must be implemented, including the variables definition, data types, and programming languages. The widespread installation of the IEC-61131 standard in the industrial environment underscores its established presence and fundamental importance in industrial automation.

Notably, within this standard, it is worth emphasizing the presence of five programming languages: Ladder Diagram (LD), the Instruction List (IL), Sequential Function Chart (SFC), Structured Text (ST), and particularly, FB [17]. The FB is characterized by a block structure with inputs and outputs specifically designed for data handling and an embedded program within the block that orchestrates its functionality, i.e., the inputs are interpreted, and the outputs are updated based on the program within the FB. To do a logical program using the FB, it is necessary to interconnect the FBs by flow lines, which visually represent the program's flow or sequence. These flow lines ensure the smooth execution of the program, enabling a systematic and efficient implementation of control or automation tasks [16].

However, the IEC-61131 standard primarily focused on centralized control, and as automation systems evolve and become increasingly distributed, there arises a need for a more flexible and scalable framework, the IEC-61499 standard.

2.2.1 IEC-61499 Standard

The IEC-61499 standard [4] provides a highly accessible framework for implementing distributed automation control systems, allowing the seamless integration of diverse controllers from different manufacturers into a single program and improving the aspects of portability, configurability, and interoperability. The distributed aspect of this approach is illustrated in Figure 4.1, where a network of FBs orchestrates the control logic across three distinct devices. Regardless of the order in which FBs are connected to the devices, the overall behavior and performance of the system remain consistent, i.e., each device can be responsible for controlling different parts of the system, where through communication, their functionalities come together and perform the desired control. Therefore, it demonstrates the system's capability to span multiple hardware platforms while maintaining cohesive control functionality.

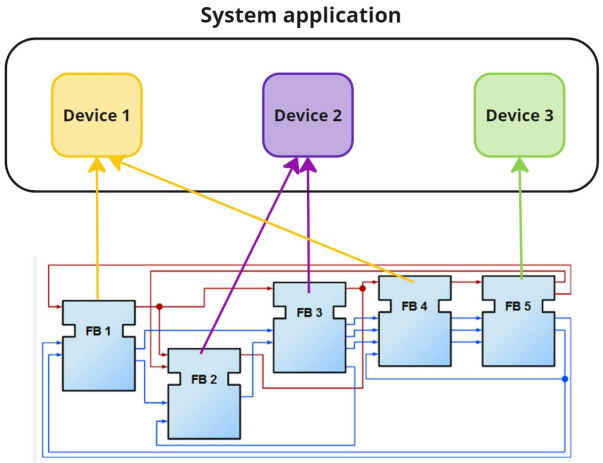


Figure 2.3: Distributed FB system (adapted from [18]).

As seen in Figure 4.1, the central element of the IEC-61499 standard is the use of a network of FBs which are connected to control the system. The internal structure of the FB, as depicted in Figure 2.4, consists of Events and Data interfaces, allowing to receive and send information and enabling to perform specific control tasks processed by the FB's internal code. This modular design not only simplifies programming for developers but also allows the implementation of diverse solutions to meet varying automation needs.

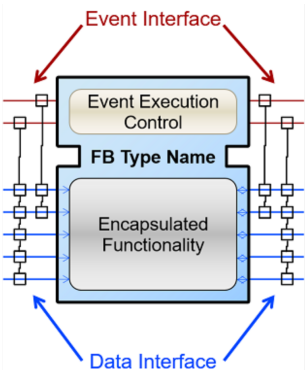


Figure 2.4: FB structure [18].

On the basic FB level, algorithm functions are activated when triggered by event inputs, and the associated input variables are read to be used in the code. The code's functions are designed to handle specific tasks, possibly involving internal support variables, and then returning values to the FB's output [18], [19]. Notably, outputs from one FB can be connected to inputs of another, even across different devices, but it must be compatible: you can not connect the event to the data. This interconnection capability allows the forming of a FB network, facilitating the implementation of distributed systems [1]. In such systems, devices can seamlessly communicate and perform specialized tasks to collectively achieve system-wide objectives. [4]

2.3 Dynamic Reconfiguration using IEC-61499 Function Blocks

In automation industrial systems, the ability to dynamically adjust the system configurations and behaviors in real-time is a pivotal requirement. Approaches like Multi-Agent System (MAS), Distributed Control Systems (DCS), and IEC-61499 FBs emerge as focal points in addressing this demand. Therefore, IEC-61499 holds a unique position due to its ease of graphical programming and its modular design, which simplifies the system development and maintenance, in addition to its compatibility with Programmable Logic Controller (PLC), making it especially applicable and practical for industrial settings.

In this context, the use of supervisory control theory to reconfigure systems is discussed in [20], developing a methodology to perform the reconfiguration control. The process involves temporarily replacing FBs or modifying their connections to accommodate the new system configuration. When the system reaches a safe state, the FBs are

changed to match the new desired configuration. On the other hand, [21] developed a standards-compliant reconfiguration methodology. Basically, this methodology consists of mapping the old system and then making the reconfiguration changes by adding or removing the necessary FBs to meet the desired functionality of the system. In this step, the management FB is present in each device, interacting through predefined functions and sending commands to be executed by the system. A similar approach was introduced by [22], although using the OPC-UA, showing the potential to offer an open configuration interface for these devices. Therefore, before applying the reconfiguration, it is necessary to analyze if the changes made can be applied to the system.

Furthermore, an alternative strategy is presented by [23] that relies on the implementation of the predefined functions used to build the FB network. It does not rely on an FB manager but instead utilizes a reconfiguration application capable of listing the reconfiguration needs according to their importance and dependencies, making the changes more precise and ensuring service continuity. Additionally, [24] employs schedulability analysis and the Priority Ceiling Protocol (PCP) to ensure that real-time tasks can meet their deadlines during reconfiguration. This approach provides a robust framework for managing low-level scheduling intricacies and resource access, making it well-suited for addressing technical feasibility.

In the same context, [25] developed an application called Reconfiguration Application (RCA) that is an interface capable of making the reconfiguration while the application is being run, avoiding costly shutdowns and ramp-up times. Meanwhile, this reconfiguration is limited in the context of the device's capacity to support RCA.

Additionally, [26] focuses his research on developing a system called Reconfiguration Architecture for Fault Handling (RAFAH), which is designed to identify faults and errors in real-time within an industrial system. When it detects a fault or error, it can adapt and change the program logic or control strategy to address the issue. It is limited by the types of faults, sensors, actuators, and structure of FBs present in the library. Meanwhile, [27] discusses the concept of dynamic adapter connections, contributing to enabling the run-time reconfiguration of component connections and addressing the limitations of static and design-time connections.

However, despite the focus on reconfiguration control approaches of the referred studies, a common observation emerges: the absence of dedicated reconfiguration FBs capable of dynamically identify condition changes and adapt the system operation in real-time. Nevertheless, [28] introduces the concept of Reconfigurable Function Blocks (RFBs) as an

extension to the IEC-61499 standard, emphasizing the significance of dynamic reconfiguration within FBs. This research presents a toolchain for modeling, formal verification, and quantitative analysis of reconfigurable distributed control systems. The proposal includes a dynamic Master-Slave Execution Control Chart (MSECC) to separate the reconfiguration model from the control model.

In this context, this work aims to make a distinctive contribution by proposing an innovative approach utilizing a Python-based FB reconfiguration approach, seeking to apply it to implement a dynamic reconfigurable conveyor transfer system. Integrating Python into FBs enables dynamic control capabilities, taking advantage of extensive and versatile libraries, facilitating the efficient data processing, identification, and correction of errors, real-time adaptability, and the seamless incorporation of advanced algorithms to improve system performance.

2.4 Framework and Visualization tool

To apply the concepts of FB, it is important to introduce and explain the framework used to develop the FB-based system and the IDE used to visualize the connections.

2.4.1 DINASORE

The DINASORE is a framework that uses Python to develop FB-based systems. It is an innovative framework that facilitates the orchestration of a pipeline of FBs across an array of devices, ultimately establishing a distributed and harmonized control paradigm [9]. Consequently, to craft the FBs within the DINASORE application, creating two essential files that collaboratively define the FB's characteristics is necessary. The initial file carries an XML file, as shown in the Listing 2.1.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE FBType SYSTEM"http://www.holobloc.com/xml/LibraryElement.dtd">
<FBType Name="" OpcUa="" Comment="">
  <InterfaceList>
    <EventInputs>
      <Event Name="" Type="Event" Comment="">
        <With Var=""/>
      </Event>
    </EventInputs>
    <EventOutputs>
      <Event Name="" Type="Event" Comment="">
      </Event>
    </EventOutputs>
    <InputVars>
      <VarDeclaration Name="" Type="" OpcUa="" Comment="">
      </VarDeclaration>
    </InputVars>
    <OutputVars>
      <VarDeclaration Name="" Type="" OpcUa="" Comment="">
      </VarDeclaration>
    </OutputVars>
  </InterfaceList>
</FBType>
```

Listing 2.1: Structure of the XML file

The XML file establishes the foundational structure of the FB, delineating input and output configurations of the event and data for each FB. Additionally, it specifies variable types, orchestrates the alignment of input and output data with respective events, and incorporates explanatory comments to enhance the understanding of each element's function.

Furthermore, a salient characteristic of the XML file is the unique type of each FB, identified by the FBType name and categorized into two primary categories, notably the Loop FBs and the Service FBs. The first category is orchestrated to execute in a loop to ensure the continuous retrieval of provided data, comprising the DEVICE.SENSOR type, signifying a sensor entity, and the POINT.STARTPOINT type, symbolizing a data-receiving protocol. On the other hand, the Service category is designed to activate exclusively when the corresponding event is triggered, encompassing the SERVICE type, denoting a processing module, and the POINT.ENDPOINT type, representative of a

data-sending protocol [9].

The other component comprises a Python file containing meticulously developed code encapsulated within a Python class carrying the same name as the previously created XML file, as shown in the Algorithm 1.

Algorithm 1: Schedule function with XML_FILE_NAME class

Input : event_name, event_value, inputs

Output: resulting_values

if *event_name == 'EVENT_NAME1'* **then**
  resulting_values ← [event_value, outputs1];
end

if *event_name == 'EVENT_NAME2'* **then**
  resulting_values ← [event_value, outputs2];
end

Therefore, within this class, a scheduled function takes center stage, where its primary function is to systematically verify whether events have been triggered and execute commands according to the specific requirements of the FB. In other words, the arguments passed to this function serve as representations of the FB's inputs and, after the code's execution, return the resulting output values. Moreover, it is essential to emphasize that the order in which inputs and outputs are defined within this code must correspond precisely to the order established in the initial XML file.

Additionally, a TEXT file, called *data_model*, is specifically crafted for each controller, manually or by the 4DIAC IDE, utilizing commands to create a FB network, establish interconnections between them, and configure necessary parameters. The commands and their respective instructions are as follows:

- **START**
  - **Instruction:** Used to start the system
  - **Implementation:** <Request ID="" Action="START"/>|

- **CREATE**
  - **Instruction:** Used to create FB and interconnection lines
  - **Implementation 1:** <Request ID="" Action="CREATE">|
    <FB Name="" Type="" /></Request>|
  - **Implementation 2:** <Request ID="" Action="CREATE">|
    <Connection Source="" Destination="" /></Request>|
- **WRITE**
  - **Instruction:** Used to write parameters in the input of the FB
  - **Implementation:** <Request ID="" Action="WRITE">|
    <Connection Source="" Destination="" /></Request>|

Therefore, the file contains commands tailored for the request type, featuring a unique ID for each task alongside the action represented by the commands. Specifically, the CREATE command requires defining the FB's name and type or specifying the connection's source and destination. Similarly, the WRITE command involves detailing specific parameters for the connection source and destination. Therefore, this file verifies the two previous files mentioned to get the information about the FB, crafting the FB network that leads the system.

2.4.2 4DIAC IDE

To aid in visualizing and streamlining the construction and behavior of FBs created by DINASORE, the 4diac IDE assumes a pivotal role. The 4DIAC IDE is a specialized tool for graphically developing systems with a focus on industry automation to provide an open IEC-61499 standard-compliant framework that allows establishing an automation and control environment as well as to provide a reference implementation [29], [30].

Moreover, the 4DIAC IDE excels beyond its modeling capacities, offering real-time simulation and visualization of control systems and enabling users to assess the behavior and performance of their designs before actual deployment. Equipped with a range of features, including online monitoring, debugging tools, and run-time analysis, the 4DIAC IDE plays a pivotal role in ensuring the reliability and efficiency of control systems within diverse industrial applications [30].

Figure 2.5, elucidates the integration within the DINASORE framework installed on the devices and the 4DIAC IDE. Notably, each device has its own FB network, which is

downloaded through the IDE, being interpreted and executed by the DINASORE, where the results could be visualized on the 4DIAC.

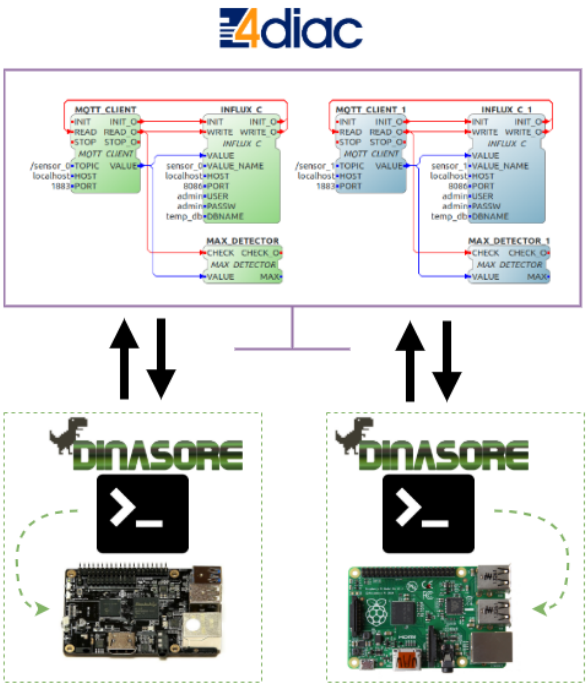


Figure 2.5: DINASORE and 4DIAC connection [31].

Furthermore, the 4DIAC IDE is divided into two primary windows: the System Configuration and the Pipeline. The System Configuration window is dedicated to configuring Ethernet connections within the devices. It provides a palette that enables users to drag-and-drop Ethernet segments and connect predefined devices easily, such as Raspberry PI, FORTE_PC, and others. It is worth emphasizing that this window is crucial for seamlessly downloading the FB network to the connected devices.

On the other hand, the Pipeline window streamlines program creation by allowing users to easily drag and drop FBs from the IDE's library, providing a visual map of device connections. Moreover, the FBs can be designated to specific devices and linked across multiple devices, simplifying connection management. Additionally, each device has its dedicated space within the system configuration, enabling independent control of its functions and specific FB to be added, enhancing overall system efficiency and organization [30].

Chapter 3

Experimental Case study

This section describes the experimental case study and the practical implementation of the FBs-based control system for the conveyor transfer system. It supports on-the-fly reconfiguration to face changes in the system configuration without the need to stop, reprogram, and restart the control system.

3.1 System Description

The case study is related to a conveyor transfer system of modular Fischetechnik conveyors, as shown in Figure 3.1, with the aim of simulating Industry 4.0, implementing multiple CPPS. Each conveyor consists of two main components: a cyber part housing the Raspberry Pi controller and a physical part operating at 24V, encompassing sensor lights, which indicate the part's arrival and departure from the conveyor, and a motor powering the conveyor. The primary objective of this system is to transfer a part from its starting point to the designated endpoint.

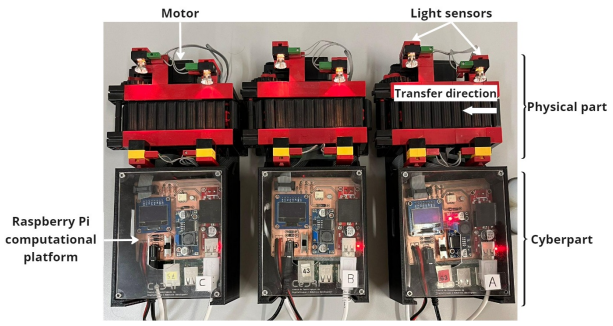


Figure 3.1: Conveyor transfer system.

Emphasizing the Raspberry Pi controller, it is a cost-effective hardware platform offering extensive I/O capabilities for automation and device control through digital ports. Its single-board computer design ensures consistent and reliable performance, facilitating swift wireless communication [15]. This feature makes the Raspberry Pi an indispensable tool for this case study due to the great exchange of messages within the modules to achieve the main objective of the system and also the specific objectives of each module to be able to communicate with the others and make the control according to its behavior.

Essentially, the conveyor modules exhibit four distinct behaviors: i) as a single conveyor, ii) as the first conveyor, iii) as an intermediate conveyor, or iv) as the last conveyor.

- Single conveyor: Triggering the input sensor starts the conveyor while the output sensor stops it;
- First conveyor: Triggering the input sensor starts the conveyor itself, and the output sensor starts the next conveyor;
- Intermediate conveyor: Triggering the input sensor stops the previous conveyor, and the output sensor starts the next conveyor;
- Last conveyor: Triggering the input sensor stops the previous conveyor, and the output sensor stops the conveyor itself.

However, The system's functionality is modeled using the Petri nets formalism, illustrated in Figure 3.2, where all the conveyors modules present the same functionality.

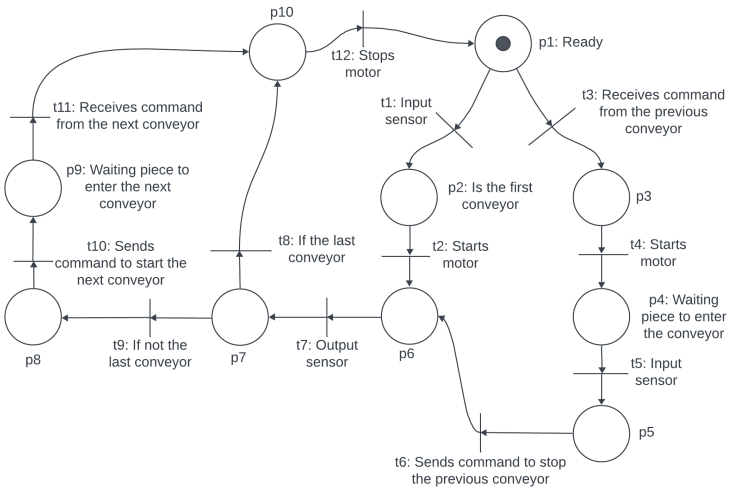


Figure 3.2: Functionality of the conveyor transfer system (adapted from [32]).

Briefly, when a part reaches the first conveyor, the input sensor is activated (transition *t1*), starting the conveyor motor and transferring the part, while the trigger of the output sensor transmits a signal to activate the next conveyor (transition *t10*). When the next conveyor receives this signal (identified by transition *t3*), it starts its motor, and when the part reaches its input sensor, a signal is sent to the previous conveyor (transition *t6*), informing that its motor can be turned off. This process is repeated until the part reaches the end of the last conveyor, with transition *t8* triggering the motor's deactivation.

3.2 Reconfiguration behaviors

The system's reconfiguration control relies on input sensor signals transmitted via MQTT, which is crucial for maintaining functionality and achieving its primary objectives. Upon receiving this information, the control FB is responsible for analyzing the system, checking for possible changes, and updating the outputs on-the-fly. Therefore, defining conditions that can automatically detect system alterations becomes pivotal, where four critical conditions are identified: i) the inclusion of a device, ii) the removal of a device, iii) a change in the initial conveyor, and iv) alterations in the intermediate conveyor's position.

3.2.1 Addition and removal of devices

First of all, to enable the inclusion of a new device, it is necessary the modification of the *data_mode* file mentioned in the subsection 2.4.1. This file contains comprehensive information regarding the interconnections and configurations of the FB within each controller. Figure 3.3 represents how the system behaves in this condition.

Examining the details in the Petri net formalism, the first step is to define the essential parameter (transition *t1*), including the parameters to connect to the broker, the definition of the Raspberry Pi pins, and the conveyor module's ID.

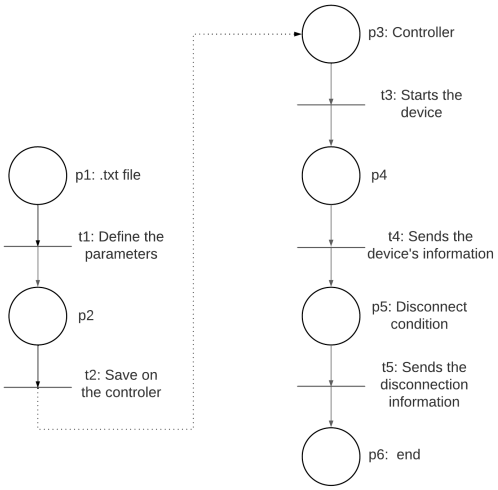


Figure 3.3: Steps to add and remove devices.

Furthermore, the *data_mode* file (represented by the place *p1*) specifically defined for the conveyor module is saved on the controller (transition *t2*). Inside the controller (represented by the place *p2*), the conveyor module can now be started with transition *t3*, exchanging the information necessary for the functionality of the system (transition *t4*). Additionally, the removal of a conveyor module is made by the conditions of the DINASORE framework disconnection on the controller or to some external issue, sending messages to the broker to inform its disconnection to the other modules (transition *t5*).

3.2.2 Conveyor's position change

Moreover, the approach to validate the first conveyor's change involves implementing a flag mechanism (a variable admitting True or False state), depicted in Figure 3.4, indicating the passage of an item from the first to the last conveyor.

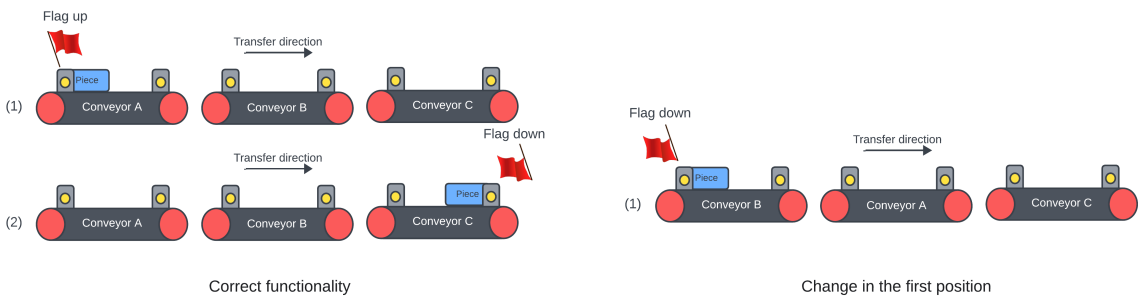


Figure 3.4: Behavior of the system when the first conveyor changes.

Basically, the flag is raised upon the part's arrival at the first conveyor and lowered when it reaches the end of the last conveyor, reflecting uninterrupted functionality. However, if any change occurs at the first position, for example, switching the conveyor A with B, the part insert on the system activates the sensor of the new first conveyor with the flag remaining unraised, indicating that a change has occurred in the system.

Furthermore, to identify the changes in the other positions, the strategy used is the time-out, depicted in Figure 3.5. This strategy involves activating a timer when a part triggers the output sensor of a conveyor and then resetting it to zero when the input sensor on the next conveyor is activated.

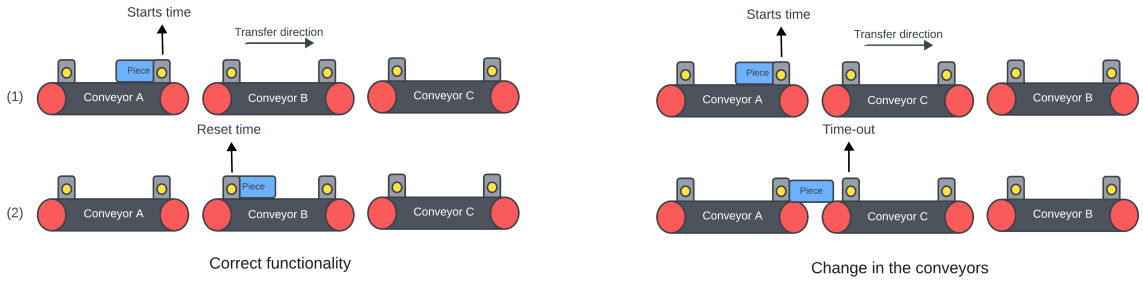


Figure 3.5: Behavior of the system in a time-out situation.

However, the condition mentioned is invalidated if alterations are introduced to the system, for example, switching the conveyor B with C, then counting until the predefined time and indicating the change on the system.

3.2.3 Capabilities

In essence, the meticulously designed network of FBs stands as a cornerstone for the conveyor transfer system, managing sensor signals to make the control. Each device operates independently and communicates with the others to exchange information, ensuring efficient part transfer. Therefore, the system is built to support the changes, even adding and removing devices or changing the conveyor's positions, without stopping or restarting.

Moreover, the key features of the FB system manifest in practical benefits:

- Reconfigurability: The system's reconfigurability enables it to evolve and expand without the need to stop or restart, ensuring adaptability to the changing needs of the production line. This capability enhances productivity and efficiency as the system can dynamically adjust to varying operational requirements.

- Scalability: The inherent modularity not only allows for the independent functioning of each conveyor but also ensures that the system can be easily scaled and adapted to accommodate changes. For instance, the addition or removal of conveyors can be seamlessly integrated without disrupting overall functionality.
- Interoperability: The standardized communication with the broker makes the system reachable to other frameworks. This feature facilitates communication with devices running different systems, fostering efficient interconnectivity between diverse systems.

Chapter 4

Implementation of the Reconfigurable Control System

For the correct functionality of the system described, a network of FBs serves as the critical component for efficient communication, control, and self-organization. Figure 4.1 illustrates the control system's structure, presenting the physical layer that supports the transfer of parts, the cyber part that uses FBs to implement the control logic for each conveyor transfer module, and the communication layer that ensures the interconnection between the individual control modules.

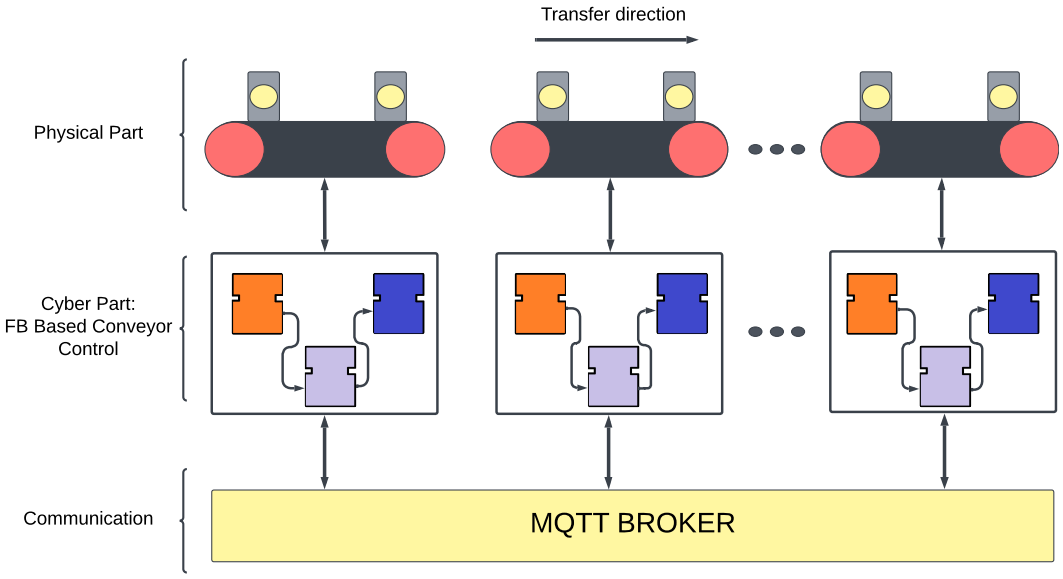


Figure 4.1: System structure.

4.1 Single conveyor transfer

The FB network shown in Figure 4.2 presents a scope designed to control a conveyor module. It is composed of FBs responsible for data handling, such as reading the input sensor's values and transmitting the information of the connected modules, as well as controlling the conveyor's motor. Also, there are FBs responsible for making the communication between conveyors, sending the data information through the MQTT broker, and receiving the information from the other conveyors. Moreover, to implement self-organization control, it is necessary to interpret the data received, check the positions of the conveyors, identify possible changes in the transfer system, and apply the appropriate control logic to each situation. This approach allows the system's scalability without the need for significant changes.

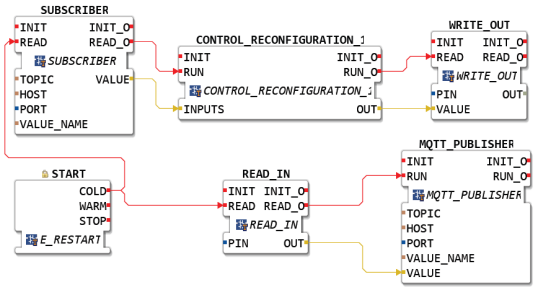


Figure 4.2: FB network to control an individual transfer conveyor module.

However, as seen earlier, these FBs are constructed by combining two files, XML and Python, but making continuous modifications to accommodate system changes can be challenging. To address this issue, the FBs were designed in this case study to save the values received in a vector, supporting the changes in the number of devices connected, maintaining the code functional and adaptable.

4.2 Communication

Since the conveyor transfer system is built up of individual and modular conveyor modules, a direct connection between their FB-based control modules is not possible, requiring the use of a communication middleware that easily and efficiently manages and routes the exchanged messages. For this purpose, MQTT was selected, more specifically, the Mosquitto broker [33] from the Eclipse, a lightweight and efficient messaging protocol

designed for communications between devices in sensor networks and IIoT systems, based on the TCP/IP protocol [34]. Figure 4.3 represents how the MQTT works.

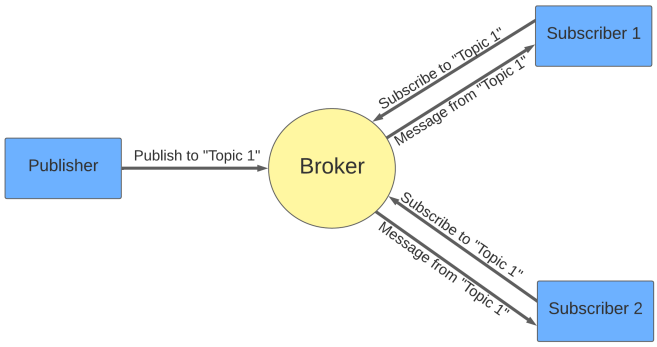


Figure 4.3: Broker Connection.

The MQTT makes use of a broker capable of mediating the exchanged messages, requiring one client to publish a particular message to a specific topic containing some value or command and sending this message to all the clients interested in this topic by subscribing to it [35].

Furthermore, the message format employed is JavaScript Object Notation (JSON), designed to be easily understandable by humans and machines. JSON 's simplicity and versatility make it language-independent, meaning many programming languages can understand it. This characteristic greatly optimizes the exchange of messages between different systems, promoting standardized and effective communication [36]. The JSON format makes the application of MQTT more viable in the conveyor transfer system than OPC-UA communication, which is commonly used in FB-based systems. This is due to the ease with which messages in JSON format can be interpreted by various technologies that can be used in self-organization applications, facilitating the system interoperability.

With this, each conveyor module sends input information to the broker, as shown in Figure 4.4, subscribing itself to receive information related to the others.

- DINASORE:
  - Device_X:
    - ID:{"ID": Value}
    - CONNECT:{"CONNECT": Value}
    - SENSOR:{"SENSOR_IN": Value, SENSOR_OUT: Value}
    - POSITION:{"POSITION": Value}
    - SWAP:{"SWAP": Value}

Figure 4.4: Payload of messages sent by a device.

This structure outlines a series of conveyor modules, each bearing distinct attributes, allocating a unique ID defined by the user, used to identify each conveyor module, holding data regarding its connectivity (CONNECT), sensors status (SENSOR_IN and SENSOR_OUT), position (POSITION) in the assembly, and information (SWAP) about a change in the conveyor's position. Figure 4.5 describes how the conveyor modules communicate with each other and the MQTT broker.

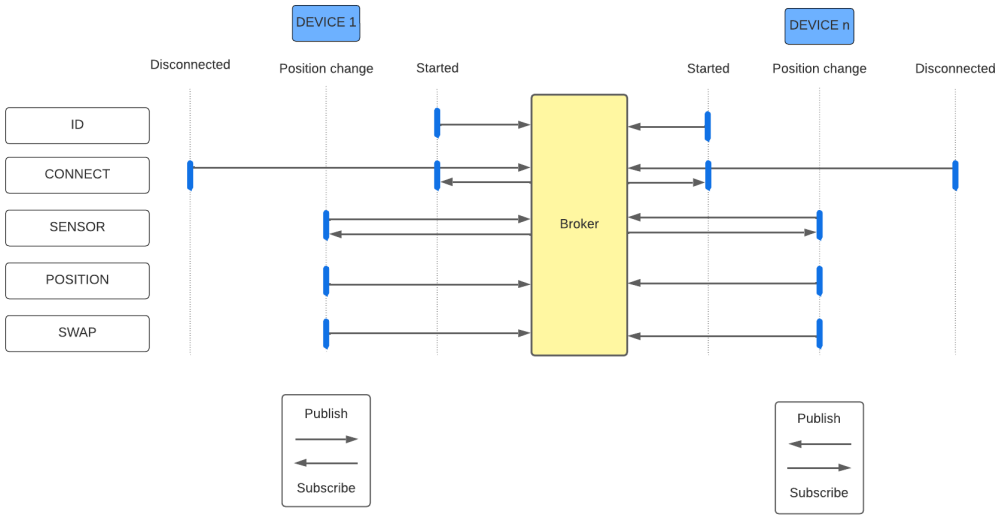


Figure 4.5: Schematic of the trigger of the messages sent to the broker.

In essence, the conveyor modules send information in three situations: i) when it is started, ii) in the standard functionality of the system, indicating the transportation of the part and the position's change, and iii) when they are disconnected. Furthermore, in the first situation, the conveyor modules sends the attributed ID and the information about its connectivity to the broker and receives the connectivity information about the others.

In the second situation, it exchanges the sensor information with the other conveyor modules to know where the part is and sends the information about its defined position in the assembly. Also, when a change occurs in the system, it sends information to the broker indicating this event, facilitating communication among other platforms. Lastly, when the device is disconnected, it informs the disconnection to the other conveyor modules.

4.2.1 Establishing MQTT Broker Communication

However, to establish the communication with the MQTT broker and exchange messages with each other to transport the necessary information to achieve the system's objective, the SUBSCRIBER and MQTT_PUBLISHER FBs, shown in Figure 4.6, were developed.

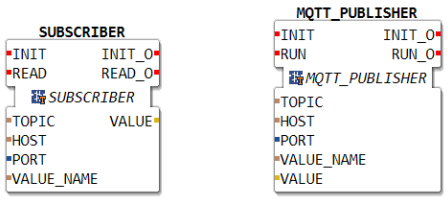


Figure 4.6: Structure of the SUBSCRIBER and MQTT_PUBLISHER FBs.

Furthermore, for the FB to function appropriately, it is necessary to provide initial parameters in their input data, enabling the effective connection with the MQTT broker and facilitating the identification of exchanged messages. To establish a secure and reliable connection, crucial information such as the broker's address (HOST) and the communication port (PORT) must be included. Similarly, it is necessary to specify the topic (TOPIC) to which messages will be published and received, along with the name that will identify the message values (VALUE_NAME) and the actual value itself (VALUE).

To establish the communication with the MQTT broker, the Python codes were developed using the Paho MQTT client package. The Paho project is an Eclipse IoT initiative that provides MQTT libraries to clients, enabling them to utilize Application Programming Interfaces (API), that serve as mechanisms to facilitate communication between components [37]. In this case, the API enables the clients to effectively send and receive messages from the MQTT broker, allowing them to establish a connection specifically for this purpose.

Additionally, two basic security measures have been implemented to ensure the security of message exchange among the controllers, effectively safeguarding against certain types of external attacks. Firstly, the port through which the Mosquitto broker connects has been changed, as the default port could be easily identified and targeted. Secondly, according to [38], it is possible to establish a username and password to connect with the broker, thereby restricting unauthorized access. As shown in Figure 4.7, the password has been generated using a HASH code formed by merging the hash function with a salt.

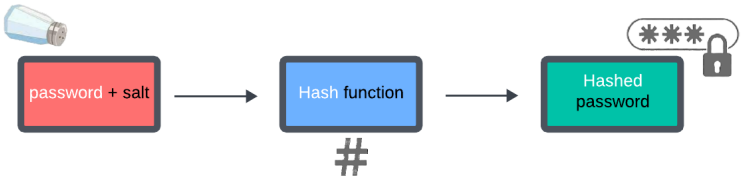


Figure 4.7: Securities measures implemented on the password generation.

The HASH function is defined as a process that transforms any random dataset into a fixed-length character series, regardless of the size of input data [39]. However, this alone might not guarantee the code's uniqueness. To address this, a technique known as salting is employed, which involves appending a randomly generated character set to the password before hashing. This added step enhances password security by introducing an extra layer of complexity, where only the encoder possesses knowledge of this salted approach [40].

It is important to note that it opted to insert the username and password directly in the code of the FBs to private the visualization of these parameters since, if they were requested on the creation of the FBs, it would make the communication more vulnerable. Moreover, the Petri net formalism, shown in Figure 4.8, demonstrates how the Python code is used to make the subscription and publication of the messages.

Furthermore, the code of both FBs is similar, with just some particularities that define their functionality. Firstly, in the SUBSCRIBER FB, the schedule function (place 3) has two events identified by INIT (place *p4*) and READ (place *p6*). The INIT event establishes the FB's connection with the broker by the _connect_to_block function (place *p5*), which uses the previously set password and username and also handles the subscription into the topics. Additionally, it initiates a loop (transition *t5*) to continuously verify if any messages arrive on the topics, decoding the JSON messages. Moreover, the READ event receives the decoded message (identified by transition *t6*) and retrieves the value corresponding to the variable's name, with transition *t8* inserting it into the FB's input.

Similarly, the MQTT_PUBLISHER FB's schedule function has two events: INIT (place *p4*) and RUN (place *p7*). The INIT event connects to the broker by place *p5*, similarly to the SUBSCRIBER FB. In addition, it sets a function called will_set (place *p6*), ensuring that a message with the False state is sent to all the broker's topics in case of faults. Moreover, the RUN event encodes the JSON message (transition *t7*) and publishes it on the topics (represented by transition *t8*).

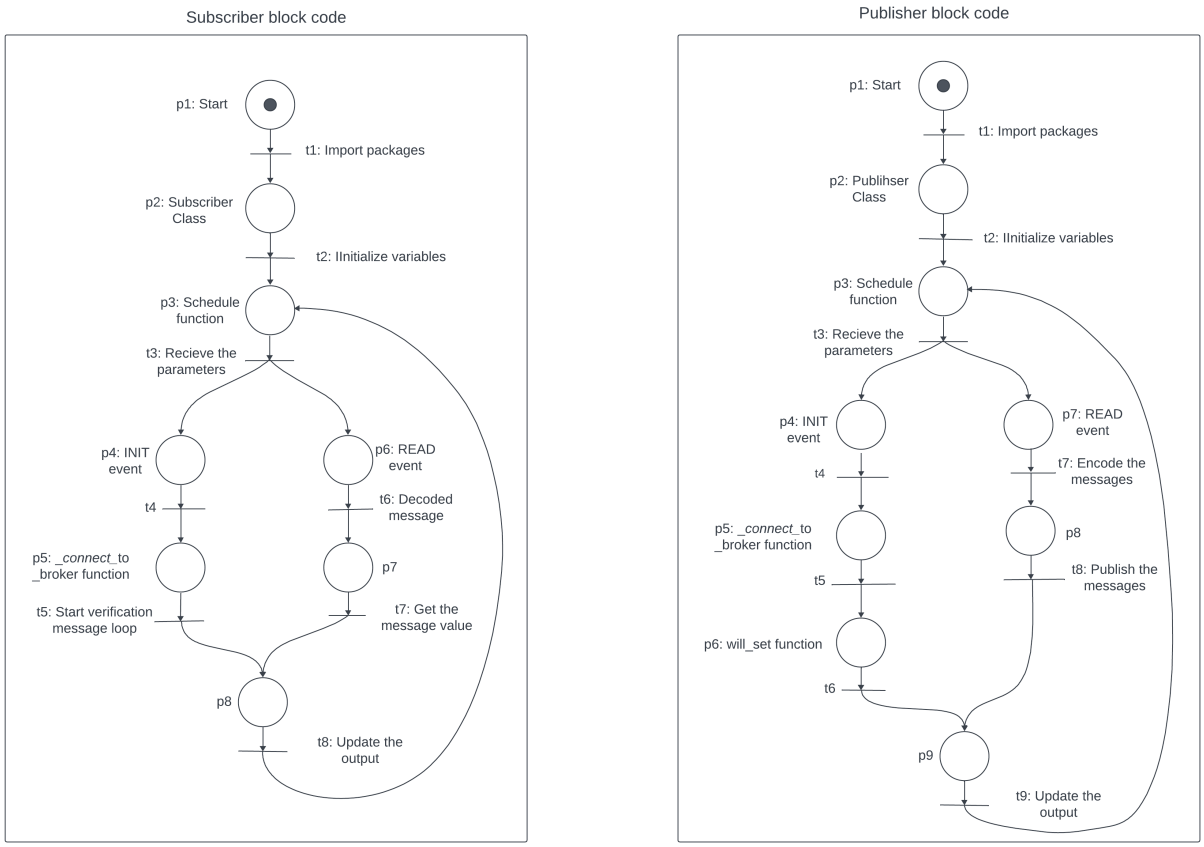


Figure 4.8: Schematic of the SUBSCRIBER and MQTT_PUBLISHER FB's Python code.

4.3 Control of the Conveyors

However, the need arose to achieve more flexible control for various conveyors, bringing the challenge of accurately determining the number of conveyor modules connected in the system, even with the addition and removal of conveyors. For that, two FBs were developed: the CONNECT and NUM_DEV.

The CONNECT FB, depicted in Figure 4.9, is responsible for indicating the connection and disconnection of the conveyor modules and its ID through the MQTT broker. Therefore, the outputs are activated when the system starts, connecting them to an MQTT_PUBLISHER FB to transmit this information to the other devices and send a False state in the event of the conveyor's disconnection.

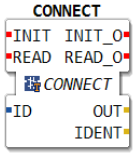


Figure 4.9: Structure of the CONNECT FB.

The NUM_DEV FB, shown in Figure 4.10, is designed to receive information about the connected conveyor modules through a SUBSCRIBER FB and then count the number of conveyor modules present in the system.

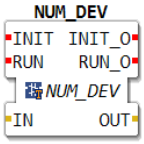


Figure 4.10: Structure of the NUM_DEV FB, responsible for counting connected devices.

Furthermore, the FB has been programmed to increase the count in the event of connection and decrease the count in the event of disconnection, ensuring the system's continued operation with the remaining devices. It sends this counted value to its output terminal, sharing this information with the other FBs that need it.

4.3.1 Reconfiguration Mechanism

The system's self-organization control relies on input sensor signals transmitted via MQTT, crucial for maintaining functionality and achieving its primary objectives. Several considerations were taken when developing the self-organization control, originating FBs responsible for analyzing the changes and adapting the system accordingly.

4.3.1.1 Conveyor's positions verification

The interconnection of the input and output of the FBs is fixed, which requires modifying the variables within the Python code to match the physical changes. The first step in following the modifications was developing the VERIFICATION_POS_CONV FB, depicted in Figure 4.11.

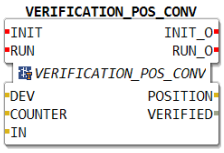


Figure 4.11: Structure of the VERIFICATION_POS_CONV FB.

The VERIFICATION_POS_CONV FB implements a scanning process to track the defined conveyor positions, receiving in the input the number of connected devices (DEV), a counter value that indicates in which conveyor has occurred the change (COUNTER), and the sensor's input data (IN), sending on the output the result of the scan (POSITION) and the information that the positions were verified (VERIFIED). The Petri net formalism shown in Figure 4.12 illustrates how the verification logic works.

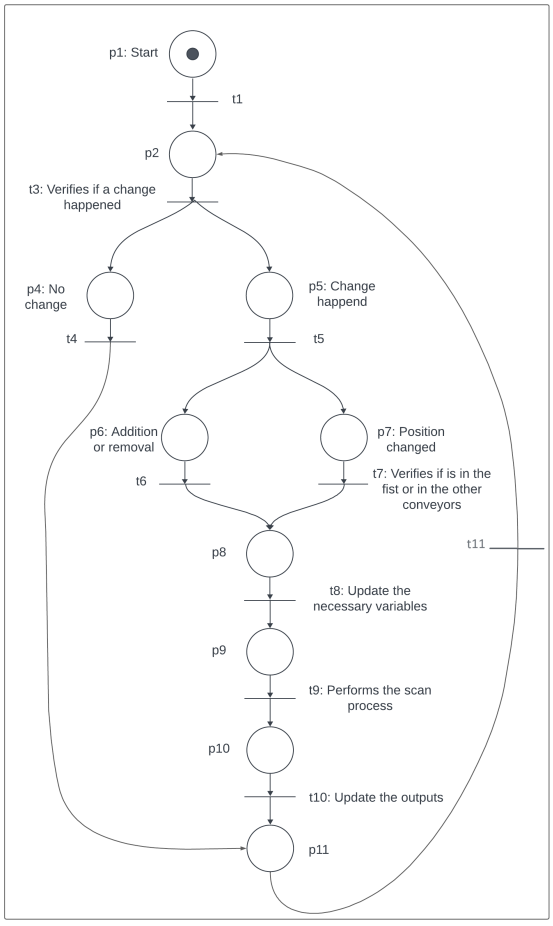


Figure 4.12: Schematic of the VERIFICATION_POS_CONV's logic.

The verification logic includes checking when the change occurred (transition 3) and then identifying the change type, where the code must modify some variables necessary to make the scan possible.

Therefore, if conveyors were added or removed (place *p6*), the code updated the number of connected devices to know when to stop the scan process. Similarly, if there was some change in the assembly (identified by place *p7*), the code verifies whether it was a change on the first conveyor or in the other conveyor's position (transition *t7*). If it was in the first conveyor, it checks which conveyor had the last sensor input, attributing this conveyor to the first position and scanning the other conveyor's position. Also, if there was a change in the other conveyor's position, the scan begins from the conveyor that has not been changed.

In this way, the scan checks whether the conveyor's input sensor state is active, storing in sequence the correspondent ID of the conveyor in a vector named self.position, which contains the numerical assignment of the conveyors according to their assembly position. An example illustrating the arrangement of values within this vector is provided below.

```
self.position=[0,1,3,4,2,5,6,8,7,9]
```

It can be concluded that, even though conveyors two and seven were correctly connected in their designated positions within the VERIFICATION_POS_CONV FB, their positions in the assembly differ. In this case, conveyor number two is the fifth, and conveyor number seven is the ninth conveyor in the assembly. Moreover, with the VERIFICATION_POS_CONV FB, it is possible to scan all the changes made to the system, even in the addition and removal of conveyor modules, and inform the new sequence to the other FBs, as well as the exact position of the conveyor in the assembly through the POSITION parameter.

4.3.1.2 Input variables update

Having the conveyor positions stored in the vector, it becomes necessary to update the input sensor variables within the code using the VAR_UPDT FB, depicted in Figure 4.13.

The VAR_UPDT FB is employed to modify the sensor's values based on the number of connected devices, particularly after reconfiguration. The input sensor's values received by the IN1 input and the output sensor's values received by the IN2 input are fixed, which

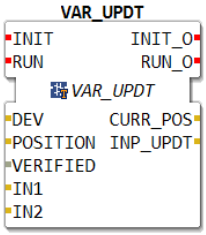


Figure 4.13: Structure of the variables Update FB.

means it is passed a vector with the sensor's values placed in the position corresponding to the device's ID.

For instance, as shown in Figure 4.14, the sensor's values from the conveyor with the ID 1 are arranged on the first position of the vector, extending to the other conveyors, independent of the assembly order.

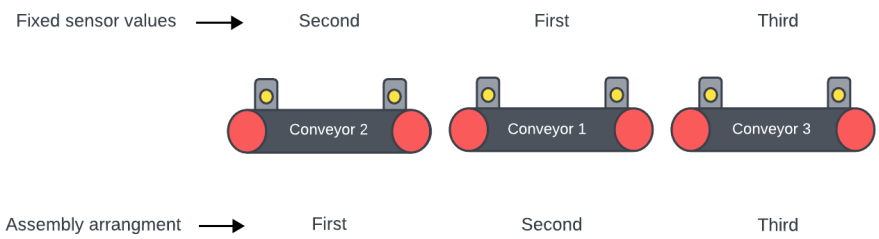


Figure 4.14: Example of the rearrangement of variables.

Therefore, the VAR_UPDT FB takes the values as arguments and reassigns them to the appropriate positions based on the conveyor setup information. Again, an example of this reassign is thinking about the conveyor with ID 1, which, even with the fixed sensor's values being in the first position, is rearranged to be in the second position due to the assembly order.

Ultimately, the function scans which conveyor the piece is currently on and sends this information via the CURR_POS output, which is necessary to know where the piece is when a change occurs.

4.3.1.3 Reconfiguration FB

The RECONFIGURATION FB illustrated in Figure 4.15 is critical in handling changes within the conveyor assembly.

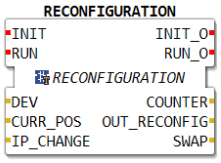


Figure 4.15: Structure of the RECONFIGURATION FB

When a change is detected, the RECONFIGURATION FB receives an ID change, where ID 1 indicates a change in the first conveyor and ID 2 indicates a change in the other conveyors. Then, it identifies the conveyors impacted by the changes and sends a signal to the FB responsible for the self-organization control to activate the motors, beginning from the point of change and continuing until the final conveyor in the sequence. Basically, the conveyor's motors are activated on two different events: when the system starts, all the motors are activated, or the conveyor's motors are activated by the changes in the system.

Furthermore, it sends a command to the VERIFICATION_POS_CONV FB to start the scan again, facilitating the system's automatic adjustment in response to physical alterations in the conveyor positions, thus ensuring uninterrupted and flawless operation.

4.3.1.4 Self-organization Control

The FBs introduced above have been developed to process the data generated in the system, making it more structured and suitable for effective reconfiguration. Therefore, it was necessary to develop a main FB, illustrated in Figure 4.16, capable of retaining this information and analyzing the system, looking for possible changes, whether changing positions, adding or removing conveyors modules, and coordinating the operation of them among the positions they can occupy, as mentioned in Section 3.1.

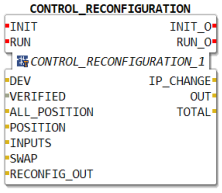


Figure 4.16: Structure of the CONTROL_RECVONFIGURATION FB.

In this way, the CONTROL_RECVONFIGURATION FB inputs correspond to the outputs of the FBs responsible for processing the data. To analyze the information, four main functions were created in the Python code to control the operation of the conveyors

in the four possible positions. Except in the case of a single conveyor, where the sensors themselves coordinate the activation and deactivation of the conveyor motor, each of these functions has different behaviors, represented in Table 4.1, that match its operating mode, receiving information from the other conveyors to the changes be noticed.

Table 4.1: Parameterization of conveyor control functions.

Specific Parameters	**Sensor Data Dependencies**		
	First conveyor	**Intermediate conveyor**	**Last conveyor**
Turn on the conveyor	Own output sensor	Previous conveyor output sensor	Previous conveyor output sensor
Turn off the conveyor	Next conveyor input sensor	Next conveyor input sensor	Own output sensor
First position change	All conveyors input sensor	All conveyors input sensor	All conveyors input sensor
Intermediate change start time-out	All conveyors output sensor	All conveyors output sensor	All conveyors output sensor
Intermediate change reset time-out	Intermediate conveyor output sensor	Intermediate conveyor output sensor	Intermediate conveyor output sensor
Raise the flag	Own output sensor	First conveyor input sensor	First conveyor input sensor
Lower the flag	Last conveyor output sensor	Last conveyor output sensor	Own output sensor

The table shows the specific parameters used in the Python code to shape the functions that drove the behavior of the conveyors in the different positions in which they could be inserted. In this way, the parameters defined were based on the activation and deactivation of the conveyors, as well as the possible changes that could occur, described in Section 3.2. The dependencies related to each parameter concern the sensor values of the other conveyors that would be received within the functions to deal with each situation.

In the case of turning the conveyors activation and deactivation, when the dependencies were reached, the conveyors would follow the respective on-and-off behaviors. In the case of the two types of changes dealt with, when the changes are detected, functions related to these changes are triggered to transmit this to the RECONFIGURATION FB, responsible for updating the outputs accordingly and informing the other FBs about the change, using an ID change.

Furthermore, a function responsible for detecting the addition and removal of conveyors was created to control the activation of the conveyors in this case. It was defined that in this situation, all the conveyors should be turned on for new learning since the

added and removed conveyors could be in any position in the system. This means that when these changes were detected, the function would wait for some input sensor to be activated before sending the activation commands. Moreover, the same behavior was used when the system was started.

4.4 Conveyor Control Setup

The FBs created using DINASORE were intended for controlling the conveyor modules, and to enable its operation on the Raspberry Pi, a series of specific configurations must be undertaken to ensure its proper functionality. The initial step involves downloading the DINASORE files from GitHub [31] and allocating them to a designated directory within the Raspberry Pi. Subsequently, executing the requirements.txt file is pivotal, encompassing the required Python packages for the framework's execution. With the prerequisites fulfilled, DINASORE is installed on the device, with directories containing important files that shape the operation of the framework.

However, focusing on the *resource* directory it is crucial in interpreting the FBs created, containing the *data_model* file responsible for setting up the process flow and interconnecting the FBs, as mentioned in Section 2.4.1, and an error log file, which records and indicates errors related to the initialization of the FBs system, making it easier to correct problems in the codes developed and the interconnections made. In addition, within the resource directory, there is a directory called *function_block*, where all the Python and XML files must be inserted so that when the system interprets the *data_model* file, it can link the FBs mentioned with those in this directory.

Thus, DINASORE is now ready for use and execution. For this to happen, a command must be passed to the device, following the instructions in Figure 4.17, where the respective device's IP must be indicated to be correctly identified.

```
# Default values:
# <ip_address>=localhost, <port_diac>=61499, <port_opc>=4840,
# <log_level>=ERROR, <number_samples>=5, <seconds_per_sample>=20
python core/main.py -a <ip_address> \
                    -p <port_diac> \
                    -u <port_opc> \
                    -l <log_level> \
                    -m <number_samples> <seconds_per_sample>
```

Figure 4.17: DINASORE instructions. Edited from [31]

As mentioned above, the interconnection of the FBs via the *data_model* file can be done manually or via the 4DIAC IDE, which makes it easier to see the interconnections and the FBs created. Figure 4.18 shows the entire structure of the system developed in 4DIAC IDE, where it is possible to see all the FBs developed performing the functions described.

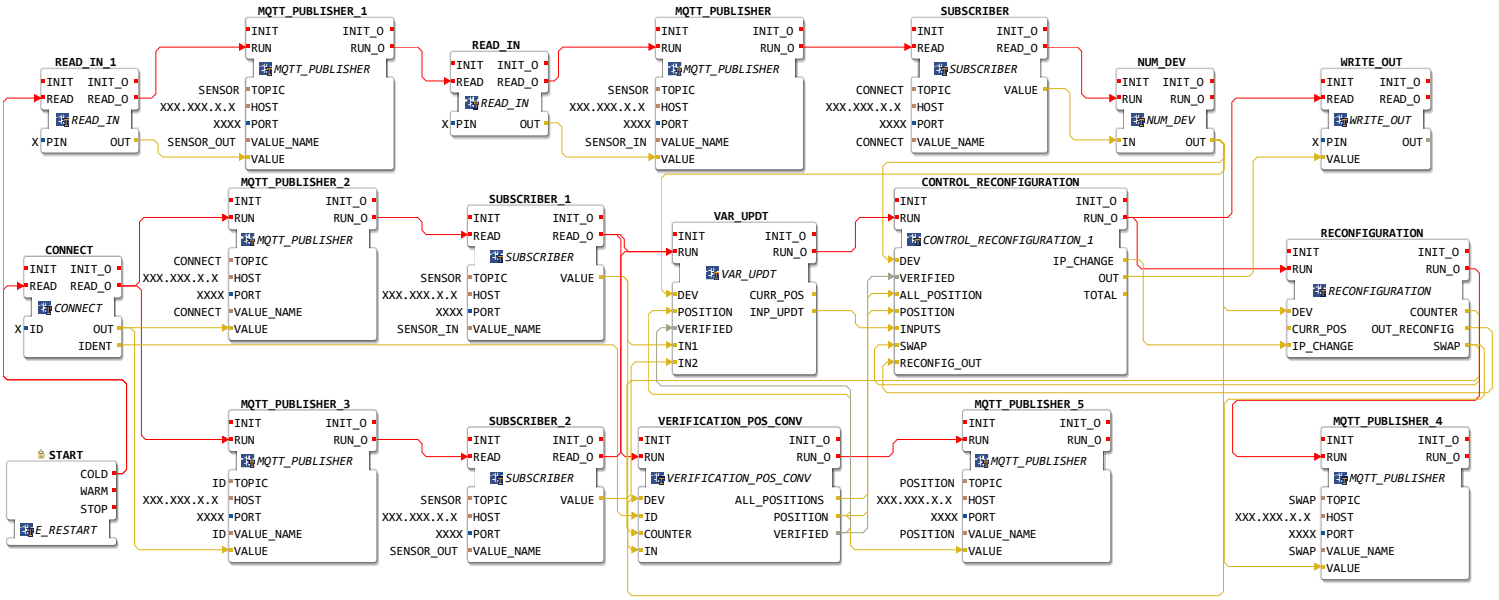


Figure 4.18: FB network made to one conveyor module to apply self-organization in the conveyor transfer system.

Therefore, the FB network was developed generically to guarantee the system's easy replication between the connected conveyor modules. This indicates that Figure 4.18 shows the interconnection between the FBs in one conveyor module, representing the same configuration in the others and demonstrating the system's modularity. This approach means that any changes to the FB network can be transferred easily and effectively to all the devices, ensuring a standardized control framework and seamlessly enabling reconfigurable control.

Thus, in order to extend the system to additional devices, it is only necessary to change specific parameters for each conveyor in the *data_model* file, such as the MQTT connection parameters and the Raspberry Pi controller pin configurations, demonstrating that the system can be easily scaled and adapted to multiple conveyors. In addition, the network shows that the SUBSCRIBER FBs are responsible for handling data related to a single message topic in the MQTT broker, increasing the efficiency of message processing.

Chapter 5

Results

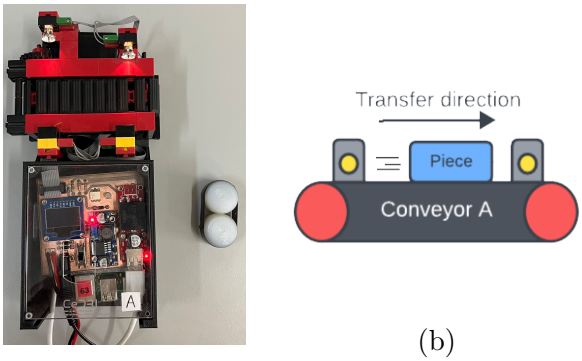
This chapter presents the results achieved within the system developed in Chapter **??**. It highlights the reconfigurability, showing the evaluation test made on the conveyor's system and how it behaves with the changes. Also, in the context of the system's scalability, the stress and performance tests present graphics containing important information about the exchanging of messages within the cyber systems.

5.1 Reconfigurability

The performance test related to the reconfigurability of the system was made on the Research Centre in Digitalization and Intelligent Robotics (CeDRI) using the Fischetechnik conveyor system made up of three modules independent of each other and labeled with different letters: A, B, and C. The test intended to show how the system behaves when the conveyors are interchanged in different assemblies, passing a piece to them in each situation. This test was separated into various steps, each one capable of showing different scenarios and characteristics.

System with one module

First, the conveyor A shown in Figure 5.1a was started with the DINASORE, passing the parameters through the `data model` file. Thus, the system is built with one conveyor, which is identified as a single conveyor, leading to its behavior.



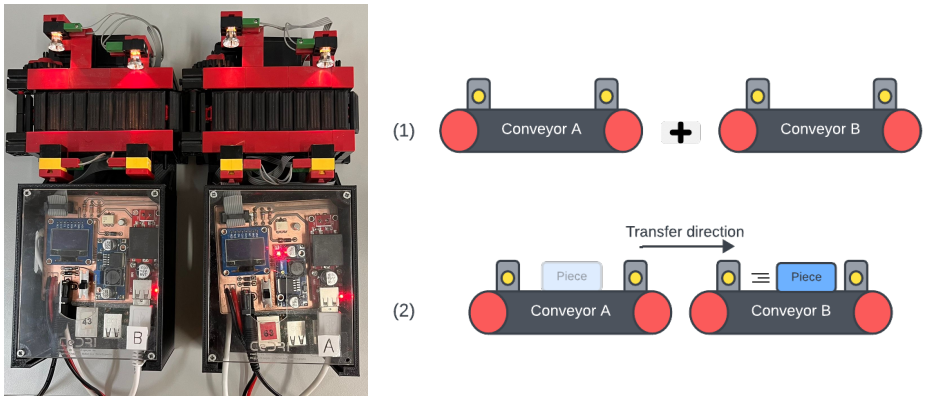
(a) (b)

Figure 5.1: System with one conveyor: (a) Conveyor A connected, (b) Behavior of the system.

The first time the piece passes through the conveyor, it learns and understands that it has just one module connected. Then, as illustrated in Figure 5.1b, the piece can be transported from one side to another of the conveyor, starting and stopping it when the input and output sensors are triggered, respectively.

Adding the second module

Subsequently, the conveyor B was integrated into the system in the same way, as depicted in Figure 5.2. This alteration required the system to discern whether the conveyors were assembled in the first or last position.



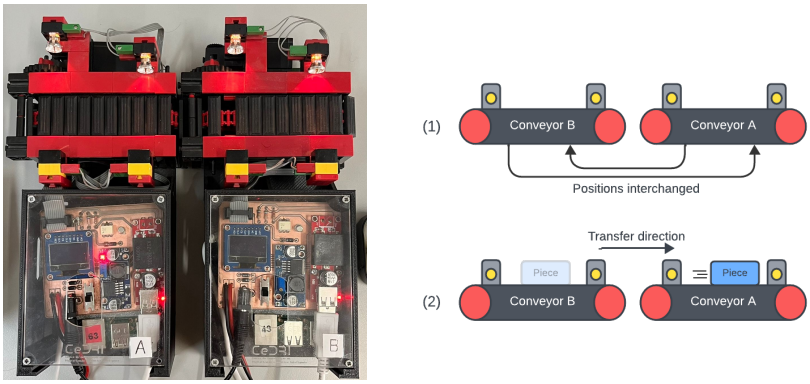
(a) (b)

Figure 5.2: System with two conveyors: (a) Conveyors A and B connected, (b) Behavior of the system with two conveyors.

Therefore, to determine the positions, both conveyors are activated, and the piece is passed through them to scan their positions with the trigger of the input sensor. Once these positions are recorded, the system can effectively transport the piece via the conveyors, activating and deactivating selectively based on their positional requirements.

Changing the position

Furthermore, the first interchange on the system was made, identified in Figure 5.3, which shows module B in the first position.



(a) (b)

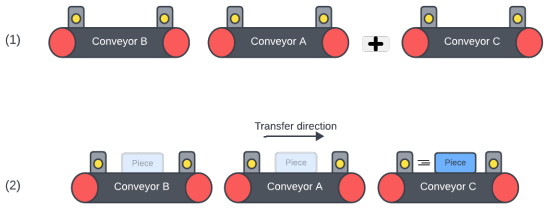
Figure 5.3: System with two conveyors interchanged: (a) Conveyors A and B interchanged, (b) Behavior of the system with an interchange.

Upon this alteration, when a piece passes through conveyor B, which is now in the first position, the system detects a change due to a down flag indicating that a piece was inserted into the system in a different conveyor. Consequently, the system activates both conveyors to relearn their positions, ensuring correct functionality.

Adding the third module

Then, the conveyor C was integrated into the system, as illustrated in Figure 5.4. Its behavior mirrored the system's response when the second conveyor was incorporated.

(a)



(b)

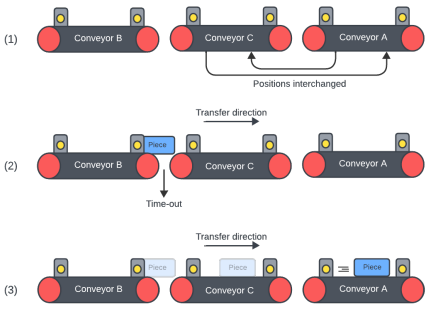
Figure 5.4: System with three conveyors: (a) Conveyors A, B, and C connected, (b) Behavior of the system with three conveyors.

Upon examination, it becomes evident that even with the introduction of a third conveyor, the system adeptly maintains its capability to accurately identify the positions of each conveyor and transfer the piece through them.

Changing the second conveyor with the last conveyor

Following the system's establishment, modifications were implemented in the conveyor positions. The initial alteration involved swapping the second and last conveyor, as illustrated in Figure 5.5.

(a)



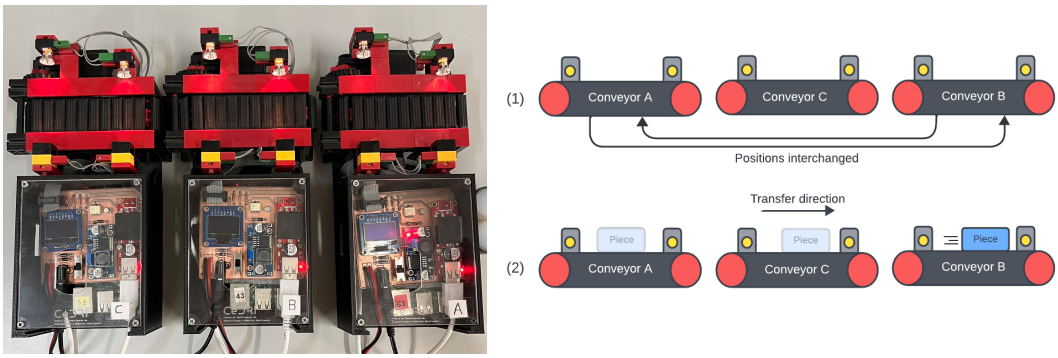
(b)

Figure 5.5: The second conveyor interchanged with the last conveyor: (a) Conveyors B and C interchanged, (b) Behavior of the system with the time-out.

To detect this change, the timeout mechanism is triggered when the command to activate the second conveyor is sent, and the piece fails to reach the input sensor within the predefined time. As depicted in Figure 5.5b, the piece stops in the middle of the system, awaiting the system's activation of all the subsequent conveyors to make the scan and relearn their positions.

Changing the last conveyor with the first conveyor

Another test involves assessing whether the system comprehended the repositioning of the first conveyor among three interconnected conveyors. As illustrated in Figure 5.6, the conveyor A was placed in the first position, the conveyor B in the second, and the conveyor C in the third position.



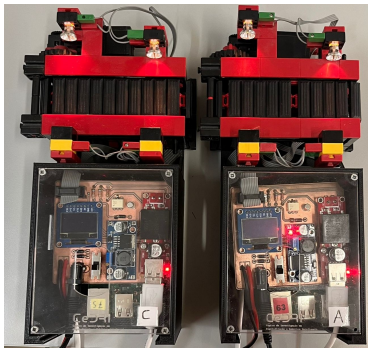
(a) (b)

Figure 5.6: Full interchanged conveyors: (a) Conveyors positioned in the C, B, and A order, (b) Behavior of the system.

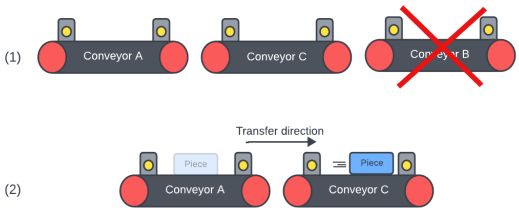
Upon this reconfiguration, facilitated by the flag's assistance, the system effectively recognized the alteration in the first conveyor's position. Consequently, all conveyors were activated to adapt and learn the new arrangement.

Removing a module

The final test conducted on the system assessed its capability to readapt the sequence when a conveyor is disconnected, demonstrated in Figure 5.7a where conveyor B was removed from the setup.



(a) C



(b)

Figure 5.7: Conveyor removed: (a) Conveyor B removed, (b) Behavior of the system in a removal.

After disconnecting a conveyor, the system promptly recognizes the change in the module count. Subsequently, when an input sensor is triggered, the system activates all remaining conveyors to re-establish the sequence. Despite the change, the system adeptly reconfigured and successfully achieved its objective of transporting the piece.

Thus, the system's consistent performance in correctly discerning the conveyor positions, despite adding, changing, and removing modules, signifies its robust adaptability, ensuring consistent functionality in transportation. This process involves a dynamic feedback mechanism where the system promptly responds to alterations in conveyor positions by relearning and readjusting its operational parameters.

Moreover, its scalable design enables effortless integration of new components or adjustments to accommodate varying operational scales, further showcasing its adaptability and flexibility. Additionally, a video showing the system operation explained previously is located at https://youtu.be/2N5E6h_n16Q.

5.2 Scalability

In the context of analyzing messages exchanged within the system's conveyors, two distinct tests were conducted: the stress test and the performance test. Apache JMeter [41] was the chosen software for these evaluations due to its functionality in conducting functional and performance tests, allowing for publishing specific messages on the topics within the MQTT broker. Therefore, to achieve the desired configuration, a test plan shown in Figure 5.8 was built containing three elements: the Once Only Controller, the Loop Controller, and the Table View.

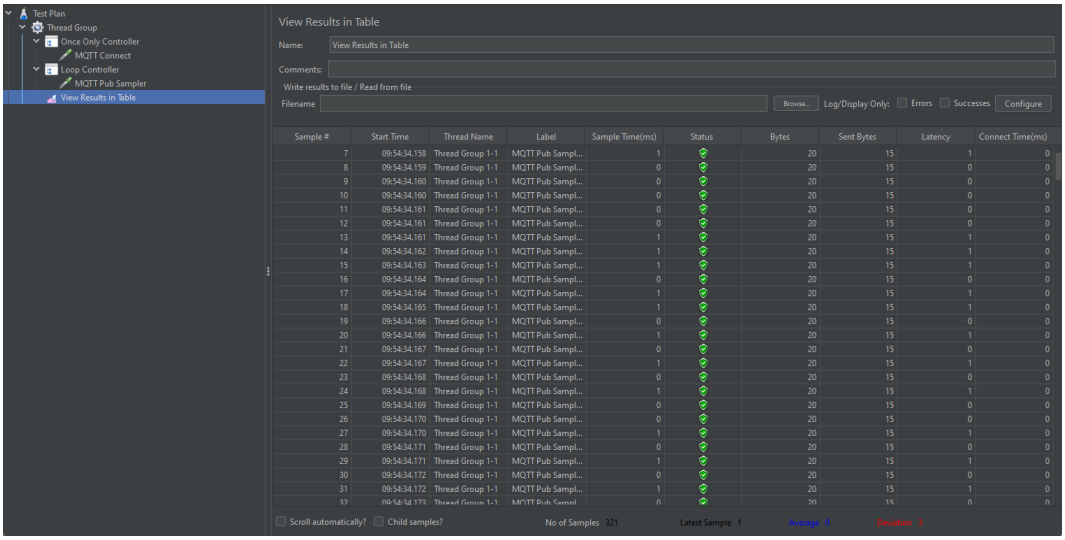


Figure 5.8: Test plan created in the Apahce JMeter.

The Once Only Controller is strategically positioned to trigger its controllers only once throughout the test plan. Its primary function is to facilitate the insertion of critical parameters required for establishing a connection with the MQTT broker, including configuring server addresses, port settings, and authentication details. Furthermore, the Loop Controller is responsible for orchestrating the transmission of varying quantities of messages to the MQTT broker in each iteration. By parameterizing the number of interactions, the Loop Controller allows the simulation of diverse scenarios and evaluates the system's performance under varying workloads. Lastly, the Table View serves as a visual representation mechanism, offering insights into critical characteristics of the transmitted messages and facilitating real-time observation and analysis of crucial metrics, including response times, success/failure statuses, or any custom data pertinent to the sent messages.

By utilizing Apache JMeter, the tests aimed to stress the system by pushing its message-handling capabilities to their limits. Additionally, they sought to assess the system's performance under regular workloads, providing insights into how well it operates under different stress and activity levels. However, it is important to note that the hardware used in the tests can impact the performance of the message exchange due to the limitations of the processing capacity of the modules. To minimize the interference, the hardware used to follow the tests is listed in Table 5.1.

Table 5.1: Description of the hardware specifications used in the tests.

Components	**Hardware**
Broker MQTT	Raspberry Pi model 3 Quad Core 1.2 GHz 64-bit CPU 1GB RAM
Function Blocks system	Raspberry Pi model 3 Quad Core 1.2 GHz 64-bit CPU 1GB RAM

5.2.1 Stress test

The system's stress tests were divided into three stages: the first was to check the SUBSCRIBER FB limit curve, the second was to assess the impact of increasing the volume of messages on reception, and the third was to perform loss mitigation to check the maximum number of messages that could be sent without losses. Furthermore, the first test evaluates the performance of the SUBSCRIBER FB in receiving and processing the message under high workloads, sending different numbers of messages to determine its limit, as shown in Table 5.2.

Table 5.2: Number of messages sent in the stress test.

Stress test		
Test	**Number of messages**	**Message size (Bytes)**
1	10	30
2	20	30
3	40	30
4	80	30
5	160	30
6	320	30
7	640	30
8	1280	30
9	2560	30
10	5120	30

Therefore, the test started with ten messages, doubling this value until a limit of processed messages was reached, and each test was repeated five times to take the average. The results are illustrated in the Figure 5.9.

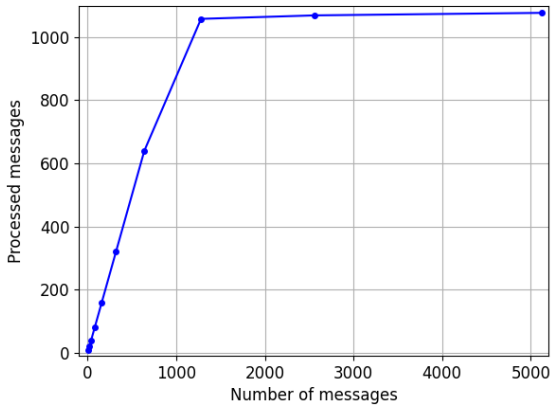


Figure 5.9: Stress test curve.

The graph illustrates a characteristic stress test curve, with an average processing of approximately 1050 messages per second through the SUBSCRIBER FB, signifying a constraint within it. Thus, with the FB's limitation identified, a test was carried out so that it would be possible to analyze the behavior and performance in delivering messages with extreme workloads, continuing to double the number of messages sent until 81920 messages. Figure 5.10 shows the results of this test.

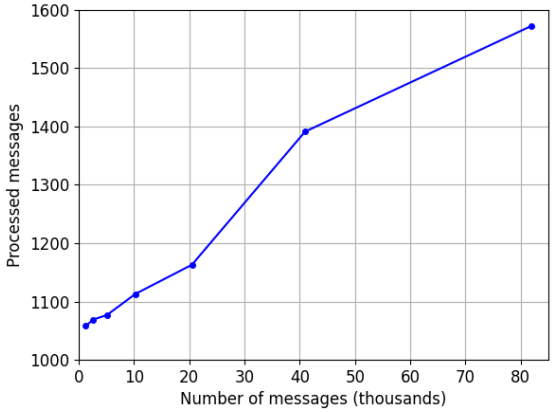


Figure 5.10: Messages received in extreme workloads.

Therefore, the graph shows that as the number of messages sent increases, the number of messages the SUBSCRIBER FB can process also increases, compared with the limitation found in the previous test, which could indicate that it can process more messages than the limit found. However, a percentage analysis of the number of messages delivered with the number of messages sent was carried out, as shown in Figure 5.11.

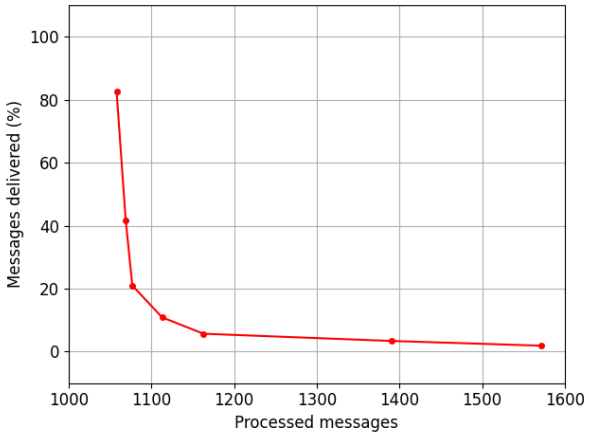


Figure 5.11: Rate of delivered messages on extreme workloads.

Analyzing this graph, it presents a decreasing exponential behavior, which indicates that even though the SUBSCRIBER FB can process a greater number of messages when subjected to a heavy workload, the percentage of messages delivered decreases significantly, resulting in a considerable loss of messages sent, indicating that sending a large number of messages is not feasible. As a result, it was necessary to carry out a loss mitigation test led by the values in Table 5.3, which would find an optimum value at which all the messages sent could be delivered and processed.

Table 5.3: Number of messages sent in the loss mitigation test.

Loss mitigation test		
Test	**Number of messages**	**Reduced messages**
1	1280	80
2	1200	80
3	1120	40
4	1080	20
5	1060	10
6	1050	5
7	1045	0

Basically, the test was carried out by reducing the number of messages sent as the percentage of messages delivered increased; in other words, as closer to 100%, the variations between the messages sent became less reduced. The results are shown in Figure 5.12

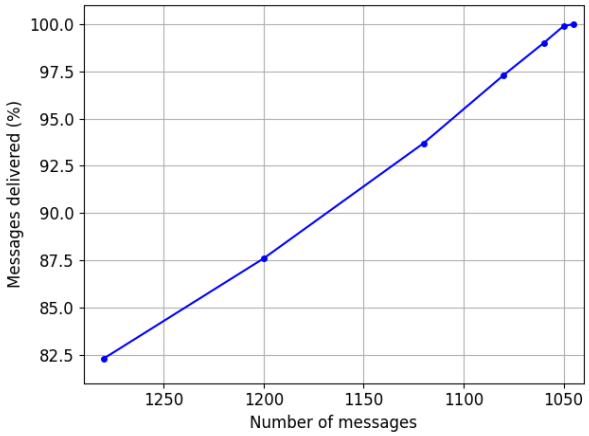


Figure 5.12: Loss mitigation test.

The results show that sending up to 1060 messages is possible to achieve a satisfactory result, with a delivery rate very close to 100%, indicating a minimal loss of messages. However, to ensure that all messages sent are delivered, up to 1045 messages can be sent simultaneously without any loss.

Therefore, as each SUBSCRIBER FB is responsible for handling one message topic, the system can support up to 1045 connected modules, which is an extraordinarily high number of modules in this case study. These insights are crucial as they indicate the SUBSCRIBER FB's processing capacity, showcasing the system's scalability up to this specific threshold without compromising its functionality.

5.2.2 Performance test

The performance test aimed to evaluate the system's processing time when receiving different quantities of messages via the MQTT topics, ranging from 1 to the limit of 1045, measuring the time taken to process each set. Each test was repeated ten times to ensure accuracy, enabling an average calculation of the processing time, as shown in Figure 5.13.

Furthermore, the graph exhibited consistency depicted by its linear behavior, where the time increases by the number of messages sent to the SUBSCRIBER FB but with the overall processing time for the given amount of messages remaining satisfactory to the case study, underscoring the system's reliability even when subjected to different message loads.

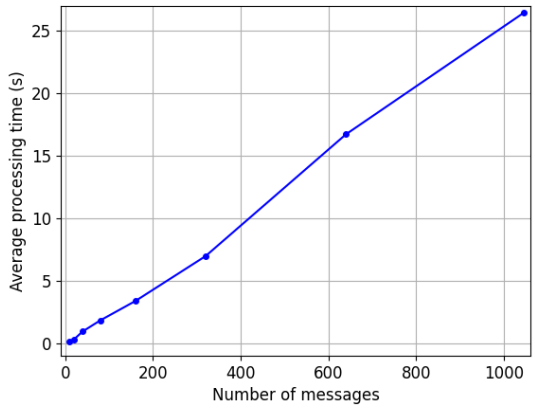


Figure 5.13: Performance test.

However, the latency graph shown in Figure 5.14 was plotted to analyze the influence of the increased workload on message processing.

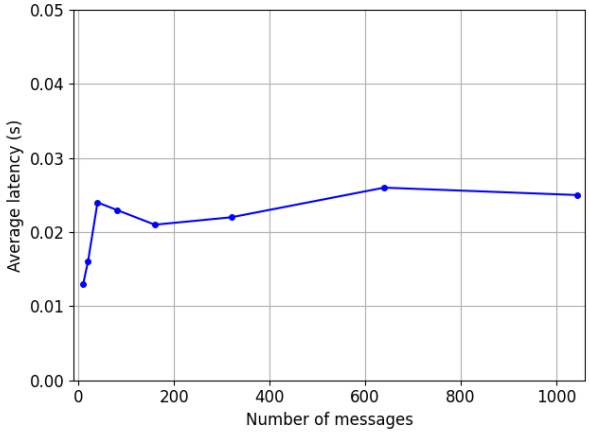


Figure 5.14: Impact of increasing the workload in the latency time.

Therefore, as the batch increases, there is a corresponding rise in the average latency time to process each message, indicating that larger batches introduce complexities that impact the system's ability to process messages while maintaining a satisfactory average process time of 0.022 seconds.

In this way, a throughput graph was carried out, as shown in Figure 5.15, to present the influence of increased latency on message processing in one second.

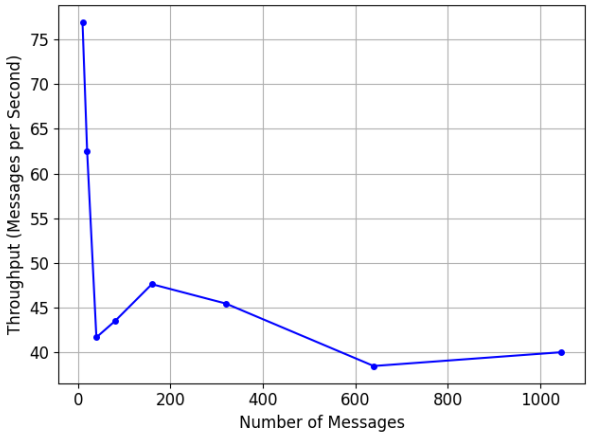


Figure 5.15: Impact of increasing the workload in the number of messages sent per second.

As a result, due to the high message traffic in this situation. the graph shows a decrease of approximately 40 messages that can be processed by the SUBSCRIBER FB per second when it is subjected to the maximum number of simultaneous messages it can handle.

Chapter 6

Conclusion and Future Work

The 4th Industrial Revolution, introduces a new era in the manufacturing sector, driven by the integration of digital technologies, intelligent automation, and interconnected systems. The CPS stands out in terms of the concepts related to Industry 4.0, making it possible to implement a decentralized control character and distributed intelligence in control systems. For the practical application of these concepts, the introduction of FBs based on the IEC-61499 standard is extremely important, managing to meet the needs of the industrial revolution using a technology that is widely disseminated in the industrial environment as an alternative to the new technologies that have emerged in the digital environment.

Therefore, this work presents the application of the Python-based FB approach using the DINASORE to develop a dynamic reconfigurable conveyor transfer system. The system aims to transport parts between conveyor modules, support changes in their positions, and allow the addition and removal of modules without the need to stop, restart, or reprogram the system.

For this implementation, FBs were developed to enable the system to communicate with the MQTT broker, effectively exchanging information between the conveyor modules. The tests carried out on the communication FBs were aimed at analyzing the system's limitations. The tests showed a processing capacity of 1045 messages delivered simultaneously, maintaining an average latency of 0.022 seconds per message. According to how the system was developed, with each SUBSCRIBER FB responsible for handling one MQTT's topic, it can support up to 1045 connected conveyor modules, demonstrating optimum scalability and robustness for this modular conveyor system.

Moreover, a dedicated reconfiguration FB was developed, which was responsible for applying the self-organizing logic, identifying and adapting to changes in the conveyor positions, and even supporting the addition and removal of conveyor modules. Therefore, the system's effectiveness was proved as it demonstrated the ability to relearn and self-organize, ensuring consistent functionality in transportation.

Future work will be devoted to:

- Comparative analysis of self-organization performance between the FB-based system and other technologies, such as Multi-Agent Systems.
- Integration of the FB-based conveyor transfer system with other technologies used in self-organization;
- Analysis of system responses to errors and fault tolerance;
- Studies related to the creation and implementation of FBs in the context of machine learning;
- Analysis of the implementation and integration of the Function blocks system using Python in an industrial controller.

Bibliography

[1] Guolin Lyu and Robert W. Brennan. "Towards IEC 61499 Based Distributed Intelligent Automation: Design and Computing Perspectives". In: *Proceedings of IEEE 17th International Conference on Industrial Informatics (INDIN)*. IEEE, July 2019. DOI: 10.1109/indin41052.2019.8972153.

[2] Diogo Oliveira et al. "A Plug-and-Play Solution for Smart Transducers in Industrial Applications Based on IEEE 1451 and IEC 61499 Standards". In: *Sensors* 22.19 (Oct. 2022), p. 7694. ISSN: 1424-8220. DOI: 10.3390/s22197694.

[3] Paulo Leitão, Armando Walter Colombo, and Stamatis Karnouskos. "Industrial automation based on cyber-physical systems technologies: Prototype implementations and challenges". In: *Computers in Industry* 81 (Sept. 2016), pp. 11–25. ISSN: 0166-3615. DOI: 10.1016/j.compind.2015.08.004.

[4] Neil Higgins et al. "Distributed Power System Automation With IEC 61850, IEC 61499, and Intelligent Control". In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 41.1 (Jan. 2011), pp. 81–92. ISSN: 1094-6977. DOI: 10.1109/tsmcc.2010.2046322.

[5] Feng Xia et al. "Function block oriented architecture for open distributed automation". In: *Proceedings of Fifth World Congress on Intelligent Control and Automation (IEEE Cat. No.04EX788)*. WCICA-04. IEEE. DOI: 10.1109/wcica.2004.1342090.

[6] Diego R. C. Silva et al. "Latency evaluation for MQTT and WebSocket Protocols: an Industry 4.0 perspective". In: *Proceedings of IEEE Symposium on Computers and Communications (ISCC)*. IEEE, June 2018. DOI: 10.1109/iscc.2018.8538692.

[7] Alexandra Schlemitz and Vitaliy Mezhuyev. "Approaches for data collection and process standardization in smart manufacturing: Systematic literature review". In:

Journal of Industrial Information Integration 38 (Mar. 2024), p. 100578. ISSN: 2452-414X. DOI: 10.1016/j.jii.2024.100578.

[8] Denis Ivanović et al. "MQTT based monitoring and control system for remote stations in automotive industry". In: *Proceedings of Zooming Innovation in Consumer Technologies Conference (ZINC)*. IEEE, May 2023. DOI: 10.1109/zinc58345.2023.10174189.

[9] Eliseu Pereira, Joao Reis, and Gil Gonçalves. "DINASORE: A Dynamic Intelligent Reconfiguration Tool for Cyber-Physical Production Systems". In: *Proceedings of Eclipse Conference on Security, Artificial Intelligence, and Modeling for the Next Generation Internet of Things (Eclipse SAM IoT)*. 2020, pp. 63–71. URL: http://ceur-ws.org/Vol-2739/#paper_9.

[10] Pranab K. Muhuri, Amit K. Shukla, and Ajith Abraham. "Industry 4.0: A bibliometric analysis and detailed overview". In: *Engineering Applications of Artificial Intelligence* 78 (Feb. 2019), pp. 218–235. ISSN: 0952-1976. DOI: 10.1016/j.engappai.2018.11.007.

[11] Gurinder Singh et al. "Industry 4.0: The Industrial Revolution and Future Landscape in Indian Market". In: *Proceedings of International Conference on Technological Advancements and Innovations (ICTAI)*. IEEE, Nov. 2021. DOI: 10.1109/ictai53825.2021.9673154.

[12] L. Sakurada. "Development of Industrial Agents in a Smart Parking System for Bicycles". MA thesis. Polytechnic Institute of Bragança, 2019.

[13] Hugh Boyes et al. "The industrial internet of things (IIoT): An analysis framework". In: *Computers in Industry* 101 (Oct. 2018), pp. 1–12. ISSN: 0166-3615. DOI: 10.1016/j.compind.2018.04.015.

[14] Emiliano Sisinni et al. "Industrial Internet of Things: Challenges, Opportunities, and Directions". In: *IEEE Transactions on Industrial Informatics* 14.11 (Nov. 2018), pp. 4724–4734. ISSN: 1941-0050. DOI: 10.1109/tii.2018.2852491.

[15] Gustavo Vieira et al. "Low-Cost Industrial Controller based on the Raspberry Pi Platform". In: *Proceedings of IEEE International Conference on Industrial Technology (ICIT)*. IEEE, Feb. 2020. DOI: 10.1109/icit45562.2020.9067148.

[16] Ramakrishnan Ramanathan. "The IEC 61131-3 programming languages features for industrial control systems". In: *Proceedings of World Automation Congress (WAC)*. IEEE, Aug. 2014. DOI: 10.1109/wac.2014.6936062.

[17] Karl-Heinz John and Michael Tiegelkamp. *IEC 61131-3: Programming Industrial Automation Systems*. Springer Berlin Heidelberg, 2001. ISBN: 9783662078471. DOI: 10.1007/978-3-662-07847-1.

[18] *Eclipse 4DIAC*. URL: https://eclipse.dev/4diac/en_help.php?helppage=html/before4DIAC/iec61499.html.

[19] Valeriy Vyatkin. "The IEC 61499 standard and its semantics". In: *IEEE Industrial Electronics Magazine* 3.4 (Dec. 2009), pp. 40–48. ISSN: 1932-4529. DOI: 10.1109/mie.2009.934796.

[20] Leandro I. Pinto, Andre B. Leal, and Roberto S. U. Rosso. "Safe dynamic reconfiguration through supervisory control in IEC 61499 compliant systems". In: *Proceedings of IEEE 15th International Conference on Industrial Informatics (INDIN)*. IEEE, July 2017. DOI: 10.1109/indin.2017.8104866.

[21] Thomas Strasser et al. "Standardized Dynamic Reconfiguration of Control Applications in Industrial Systems". In: *International Journal of Applied Industrial Engineering* 2.1 (Jan. 2014), pp. 57–73. ISSN: 2155-4161. DOI: 10.4018/ijaie.2014010104.

[22] Muddasir Shakil and Alois Zoitl. "OPC UA based IEC 61499 Device Configuration Interface". In: *Proceedings of IEEE Conference on Industrial Cyberphysical Systems (ICPS)*. IEEE, June 2020. DOI: 10.1109/icps48405.2020.9274770.

[23] Laurin Prenzel and Sebastian Steinhorst. "Automated Dependency Resolution for Dynamic Reconfiguration of IEC 61499". In: *Proceedings of 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, Sept. 2021. DOI: 10.1109/etfa45728.2021.9613156.

[24] Laurin Prenzel, Simon Hofmann, and Sebastian Steinhorst. "Real-time Dynamic Reconfiguration for IEC 61499". In: *Proceedings of IEEE 5th International Conference on Industrial Cyber-Physical Systems (ICPS)*. IEEE, May 2022. DOI: 10.1109/icps51978.2022.9816872.

[25] A Zoitl et al. "A real-time reconfiguration infrastructure for distributed embedded control systems". In: *Proceedings of IEEE 15th Conference on Emerging Technologies; Factory Automation (ETFA 2010)*. IEEE, Sept. 2010. DOI: 10.1109/etfa.2010.5641259.

[26] Herbert A. S. Leitao et al. “Fault Handling in Discrete Event Systems Applied to IEC 61499”. In: *Proceedings of 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, Sept. 2020. DOI: 10.1109/etfa46521.2020.9212177.

[27] Paavo Kajola et al. “Dynamic Adapter Connections for IEC 61499”. In: *Proceedings of 22nd IEEE International Conference on Industrial Technology (ICIT)*. IEEE, Mar. 2021. DOI: 10.1109/icit46573.2021.9453625.

[28] Safa Guellouz et al. “Reconfigurable function blocks: Extension to the standard IEC 61499”. In: *Proceedings of IEEE/ACS 13th International Conference of Computer Systems and Applications (AICCSA)*. IEEE, Nov. 2016. DOI: 10.1109/aiccsa.2016.7945784.

[29] Thomas Weber, Alois Zoitl, and Heinrich HuBmann. “Usability of Development Tools: A CASE-Study”. In: *Proceedings of ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. IEEE, Sept. 2019. DOI: 10.1109/models-c.2019.00037.

[30] Thomas Strasser et al. “Framework for Distributed Industrial Automation and Control (4DIAC)”. In: *Proceedings of 6th IEEE International Conference on Industrial Informatics*. IEEE, July 2008. DOI: 10.1109/indin.2008.4618110.

[31] DIGI2-FEUP. *Dinasore Repository*. 2023. URL: https://github.com/DIGI2-FEUP/dinasore.

[32] José Barbosa, Paulo Leitão, and Joy Teixeira. “Empowering a Cyber-Physical System for a Modular Conveyor System with Self-organization”. In: *Studies in Computational Intelligence*. Springer International Publishing, 2018, pp. 157–170. ISBN: 9783319737515. DOI: 10.1007/978-3-319-73751-5_12.

[33] *Eclipse Mosquitto*. Jan. 2018. URL: https://mosquitto.org/.

[34] Lavinia Nastase. “Security in the Internet of Things: A Survey on Application Layer Protocols”. In: *Proceedings of 21st International Conference on Control Systems and Computer Science (CSCS)*. IEEE, May 2017. DOI: 10.1109/cscs.2017.101.

[35] Dan Dinculeană and Xiaochun Cheng. “Vulnerabilities and Limitations of MQTT Protocol Used between IoT Devices”. In: *Applied Sciences* 9.5 (Feb. 2019), p. 848. ISSN: 2076-3417. DOI: 10.3390/app9050848.

[36] *JSON*. Aug. 2023. URL: https://www.json.org/json-en.html.

[37] Rohit Dhall and Vijender Kumar Solanki. “An IoT Based Predictive Connected Car Maintenance Approach”. In: *International Journal of Interactive Multimedia and Artificial Intelligence* 4.3 (2017), p. 16. ISSN: 1989-1660. DOI: 10.9781/ijimai.2017.433.

[38] *MQTT FAQ*. URL: https://mqtt.org/faq/.

[39] Peiyao Chen et al. “Hash-Polar Codes With Application to 5G”. In: *IEEE Access* 7 (2019), pp. 12441–12455. ISSN: 2169-3536. DOI: 10.1109/access.2019.2892969.

[40] European Data Protection Supervisor Agencia Espanôla Protección datos. *Introduction to the Hash Function as a Personal Data Pseudonymisation Techinique*. EUROPEAN DATA PROTECTION SUPERVISOR. Oct. 2019.

[41] Emily H. Halili. *Apache JMeter. A practical beginner's guide to automated testing and performance measurement for your websites*. 1. publ. From technologies to solutions. Includes useful references (p. [115]-116) and index. Birmingham [u.a.]: Packt Publ., 2008. 129 pp. ISBN: 9781847192950.

Appendix A

Source code developed in this work

This appendix comprises the link to access the codes developed in this work, with a folder called FB_codes and a file called data_model.

The FB_codes folder contains the Python and XML codes used to develop the blocks. Each FB contains both types of files with the same name assigned to it. This folder is separated into two other folders, one containing the codes for the self-reconfiguration FBs and the other containing the auxiliary FBs.

The data_model file contains the commands used to create the FB network.

The access link is: https://github.com/LeonardoMendonca08/Conveyor-Transfer-System.git.