FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

A Digital Twin for Education with FACTORY IO

André Miguel do Vale Ventura de Sousa

Mestrado em Engenharia Eletrotécnica e de Computadores

Supervisor: Paulo José Lopes Machado Portugal

July 25, 2022

Resumo

Com o passar dos anos, a pressão nas indústrias para aumentar a eficiência, diminuir tempos de entrega e melhorar a qualidade dos seus produtos foi criando a necessidade de novas abordagens em relação a planeamento, execução, e monitorização do processo de produção. Esta necessidade é uma base da Indústria 4.0, com conceitos como Sistemas Ciberfísicos e Internet das Coisas tornando-se o modo desejável de desenvolver os sistemas complexos que constituem processos de produção. A capacidade de prever e simular mudanças ao nível da fábrica é uma mais-valia ganha pela utilização de Digital Twins, cujas cópias digitais de uma fábrica real trazem flexibilidade e maior facilidade de simulação ao processo de engenharia. Enquanto que Digital Twins são muito úteis, são difíceis de desenvolver para uma planta de fábrica genérica e, enquanto que existem ferramentas disponíveis para o seu desenvolvimento, não são bem adaptadas para propósitos educacionais já que são caras e demasiado complexas. Neste documento, será proposta uma implementação de um Digital Twin modular, versátil e altamente estruturada para o simulador de fábrica FACTORY I/O, usando o softPLC CoDeSys, que pode ser rapidamente montada para virtualizar qualquer configuração de fábrica construida no FACTORY I/O.

Abstract

With every passing year, the pressure on industries to increase efficiency, shorten delivery times and improve product quality has created the need of new approaches regarding planning, executing and monitoring of the manufacturing process. This need is one of the roots of Industry 4.0, with concepts like Cyber-Physical Systems (CPS) and Internet of Things (IoT) becoming the desirable way of engineering the complex systems that constitute production processes. The ability to predict and simulate changes in the factory level is the main value gained by the employment of Digital Twins, whose digital copy of the real factory environment bring flexibility and easiness of simulation to the engineering process. While Digital Twins are very useful, they are hard to develop for a generic factory plant, and, while there are tools available to aid with their development, they are not well suited for educational purposes as they are expensive and too complex. In this document, it will be proposed a modular, versatile and highly structured implementation of a Digital Twin for the factory simulator FACTORY I/O, using the softPLC CoDeSys, which can be quickly setup to virtualize any factory layout built in FACTORY I/O.

Acknowledgments

I would like to thank everyone that contributed to the successful completion of this work, in one way or another, with special thanks to:

My supervisor, Professor Paulo Portugal, for the suggestions and helpful insight;

My friends, Jorge Ferreira, Inês Silva, Mariana Silva and António Machado, for always being there when I needed;

My friends and colleagues, Nuno Ricardo and Lucas Agostinho, for all the help given throughout these years. I am certain they will become great engineers;

My parents, Fernando and Dalila, my siblings Susana and Nuno, and my siblings-in-law Hugo and Konstanca - their constant support means everything.

Finally, I would like to thank the rest of my family, which I am very proud of being a part of, for always believing in me.

André Sousa

Contents

List of Figures

List of Tables

Abbreviations

API	Application Programming Interface
CBS	Cyber-Physical Systems
COM	Component Object Model
CoDeSys	Controller Development System
FB	Function Block
HMI	Human Machine Interface
IoT	Internet of Things
MES	Manufacturing Execution System
OPC UA	OPC Unified Architecture
POU	Program Organization Unit
SCADA	Supervisory Control and Data Acquisition
TAPP	Two Axis Pick & Place

Chapter 1

Introduction

1.1 Context

To be able to respond quickly to unexpected events without re-planning, future manufacturing systems will need to become more autonomous. These systems employ highly intelligent machines that know their capabilities (skills) and can use them to perform high-level tasks specifications without being explicitly programmed, through the use of flexible orchestration of elementary capabilities. To do this, a machine needs to access a large volume of information regarding its state, the state of the system in which is embedded, and the consequences of its interactions with the environment [1]. This creates extremely complex systems, in most cases implemented in real physical factories. In order to optimize such systems, it is important to have a way to test different possibilities with the factory layout, analyse data produced by the system, and be able to change parameters in the system without actually touching the physical process. This is important because it is highly inconvenient to have to stop the system to run these optimization possibilities, creating a huge efficiency drop in the factory. In order to tackle this, a virtual mirror of the real operating conditions in real-time behaviour is implemented – a Digital Twin.

Throughout the years, different definitions from different authors of a "Digital Twin" have been given. Before the rise of Industry 4.0, this definition was more directed towards the aerospace industry, with the intent of aiding with the prediction of long-term performance of airspace crafts [2],[3], but with the industrial revolution it shifted towards manufacturing and smart products [4]. In the context of this document, a Digital Twin will be defined as a virtualization of a factory floor process which contains all the relevant information and characteristics needed to correctly represent the physical system, allowing for optimization strategies to be tested in this virtual environment without touching the physical system. It is a virtual and easily manipulated process, which is able to respond to a user's input and provide the information needed for the user to estimate the impact of such inputs in the real system.

1.2 Motivation

A complete professional Digital Twin is a really complex tool with many functionalities. There is interest, at FEUP, of using a simplified version of a Digital Twin specifically adapted to education, which in its core still virtualizes a factory plant and is capable of communicating with a higher-level application, like a Manufacturing Execution System (MES), but it does not have a very steep learning curve for the user. There is software already developed for the professional implementation of Digital Twins but, even though some are adapted to academic environments, they are still quite complex as they contain features that steer away from factory plant simulation, and are expensive to acquire. The main motivation for the development of this work is, therefore, the lack of good options for the simulation of industrial scenarios, adapted to teaching students studying at M.EEC.

1.3 Objectives

The main objective of this work is to develop a simplified, academic Digital Twin with software available to the Faculty. For this reason, the choice of the factory simulator fell upon FACTORY I/O, as its licenses are already available at FEUP, and it is a simple graphical factory simulator, but very complete as well regarding the equipment and functionalities that it provides, when considering industrial environments. Because this software is only a factory simulator by itself, there is also the need to use software capable of controlling this simulator. For this, CoDeSys is a very good option as it is free to use, rather simple to learn, and it can be used as a softPLC which, in an industrial automation course, is always an asset.

This work does not concern only the adaptation of Digital Twins though. One of the big drawbacks of the development of Digital Twins is that it is a time-consuming manual task that requires a specialized engineer to perform it, and is only applicable to the factory to which it was developed. For every factory layout, a different Digital Twin has to be developed which requires more work and specialized personnel. This is a problem both in professional and academic environments, as time is of the essence in both. Given this, a main objective of this work is also to develop a way to make the creation of Digital Twins for an arbitrary factory plant as simple as if for a specific plant. To do this, the solution has to be highly modular, flexible and the development methodical, to also keep this tool simple to learn how to use.

1.4 Document Structure

Besides this Introduction, this document is constituted of five more Chapters.

Chapter 2 presents the theoretical aspects of this work in and the current state of the art.

Chapter 3 presents, in a detailed manner, the architecture developed for this work.

Chapter 4 contains the aspects of how this work is implemented in practice.

Chapter 5 discusses the tests performed to validate the work and gives an example of the experience of using this work.

Final conclusions and future work are presented in chapter 6.

Appendix A contains the numerical correspondences used in the implementation of FACTORY I/O terms.

Chapter 2

Background and state of the art

2.1 Background

In this section, the main software and technologies used to complete this work will be exposed, namely FACTORY I/O, CoDeSys, OPC UA and IEC 61131-3.

2.1.1 FACTORY I/O

Factory I/O is a 3D factory simulator for learning automation technologies. Designed to be easy to use, it allows to quickly build a virtual factory layout using a selection of common industrial parts [5]. Overall, this is where the user builds the factory to simulate, and little to no input is made here from the perspective of someone who is going to develop a tool to work with FACTORY I/O.

FACTORY I/O's equipment pallet contains all the elements one could find in a typical factory floor. The most important ones are divided in roughly five groups: Loads, Sensors, Items, Stations and Operators.

- **Items** are divided into two categories: Pallets, and Parts. Pallets are support items which can carry parts on top of them. Parts, on the other hand, can be light-load or heavy-load parts (Figure 2.1). While the light-load parts are more directed towards assembly lines, with the possibility of transforming raw materials into lids or bases and assembling these parts together, the heavy-load parts are more used for storage management, with equipment like the Warehouse and items like stackable boxes. These items can be inserted into the factory through the use of equipment called Emitters and removed with Removers (Fig 2.2).
- **Loads** refer to all the conveyors, rollers, arms and parts that move items around. These are, just as the items, divided into heavy-load and light-load. While light-load equipment should not carry heavy-load items, the opposite can be true, as a pallet can have light-load parts on top of it, and be carried by heavy-load conveyors. Nevertheless it is a rather uncommon setup and not very relevant for this work.
- **Sensors** can be placed almost anywhere in the factory and have different characteristics. FACTORY I/O has available the most used types of sensors in the industrial environment.

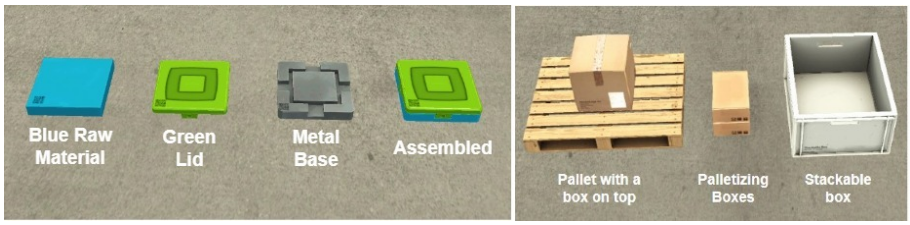

Figure 2.1: Light-load parts (left) and heavy-load parts (right)

The most used in this work are: the capacitive sensor, which has a close range but can detect any material; the retro reflective sensor which has a very long range but needs a reflector to work; and finally the vision sensor, which can determine which type of part it is detecting and output a value accordingly.

- **Stations** can also be called cells and are made of groups of other simpler parts like a specific configuration of conveyors and sensors or more complex pre-built equipment like a machining center. These stations are more complex in nature than the individual parts.

- **Operators** are all kinds of human-system interfaces, like buttons, alarms, and lights.

A brief description of the most relevant equipment of FACTORY I/O for this work will be made, to provide an easier understanding of this document.

2.1.1.1 Conveyors

Conveyors, shown in Figure 2.2, are the most basic element of FACTORY I/O. They can be configured to move in both directions and are used to connect all the different equipment in the factory. Even though they exist in different shapes and sizes (different lengths, curved conveyors, ramp conveyors, etc.) their operation is always similar.

Figure 2.2: A light-load conveyor with an emitter (green) and a remover (orange)

2.1.1.2 Emitters and Removers

Emitters, the green arrow in Figure 2.2, are arguably the most important equipment in the factory as its their job to generate items into the factory. They can be configured to generate specific, preset parts (i.e. set by the user through the FACTORY I/O interface), or a controller can decide what part is generated. This is done through a numerical code where each number corresponds to a part. It can also be configured the delay of generation, (i.e. the time between the order of generation and the actual generation) and the maximum number of parts allowed in the factory at a time. If this limit is reached, the Emitter does not emit anything even if signaled to do so.

Removers (orange arrow in the same figure) are the opposite of Emitters as they remove the parts from the factory once these reach them. It can be defined what parts a Remover can remove, as well as if the actual removal is controlled by a controller or if it set manually on FACTORY I/O.

2.1.1.3 Machining Center

The Machining Center (Fig 2.3) is a cell responsible for turning raw materials into lids or bases, at choice. Raw material that enters this cell at the entry bay (left) is picked up by the robotic arm and is placed in the manufacturing machine (center) which transforms the material into the desired part. After a pre-determined time (3 seconds for bases, 6 for lids) the robotic arm transports the newly made part to the exit bay (right). This cell is controlled through signals that can be sent through a controller, or manually inputted through buttons available with the cell, and can be stopped/started, and reset at any time. It also possesses a sensor at its entry that identifies if the part that arrived is, or not, a raw material and, if not, signals an error.

Figure 2.3: Cell Machining Center in FACTORY I/O

2.1.1.4 Assembler and Two Axis Pick & Place

The Assembler (Fig 2.4) is capable of mounting lids and bases together. In order to guarantee a correct fit, the lids and bases need to be pre-aligned by Positioning Bars, before the robotic arm picks them up and joins them together. The Two Axis Pick & Place (TAPP) can move assembled

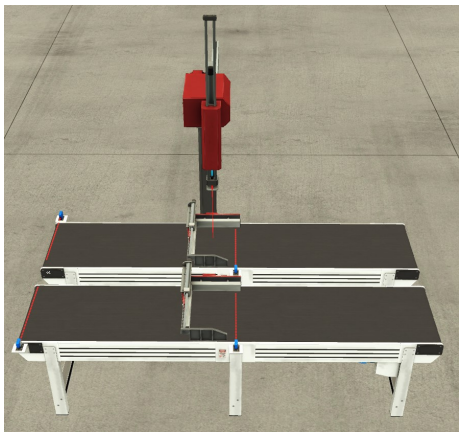

Figure 2.4: Cell Assembler in FACTORY I/O

items or parts from one of its conveyors to the other. In practice, this can be performed by the same group of equipment so the Assembler and TAPP are the same cell.

2.1.1.5 Sorter

The cell Sorter (Fig 2.5) possesses a vision sensor at its entry that is capable of recognizing what part is being detected (what type of part or what colour). It also has robotic arms (in the figure, the middle one is already turned) that can direct the parts towards the intended exit, on the sides of the main conveyor.

Figure 2.5: Cell Sorter in FACTORY I/O

2.1.1.6 Warehouse

The cell Warehouse (Fig 2.6) possesses racks where it can store parts, a crane to transport the parts to the intended rack, and entry and exit conveyors. In order to load/unload parts to/from the crane, special conveyors called Loading Conveyors are used, which are in every operation similar to normal Conveyors but are physically different. It can store up to 54 parts and a part, in order to be correctly stored away, needs to have a pallet to support it.

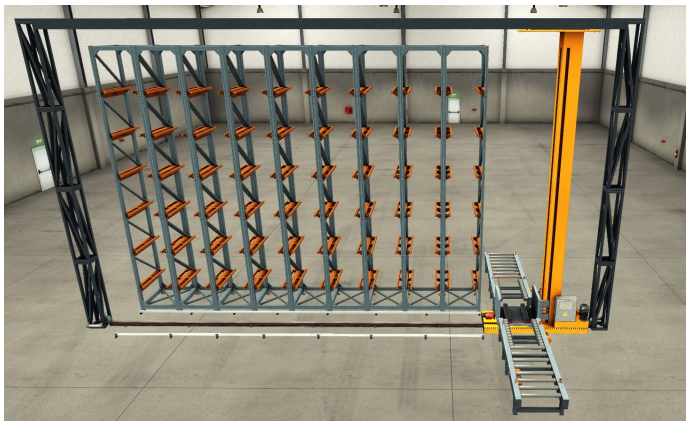

Figure 2.6: Cell Warehouse in FACTORY I/O

FACTORY I/O possesses a built-in feature called "Drivers I/O" that allows it to communicate with external machines, like PLCs, through the most used standards in the industrial environment.

FACTORY I/O has other features, and more information can be found in the user manual.[1]

2.1.2 CoDeSys

Controller Development System, or CoDeSys, is a development environment for programming controller applications according to the international industrial standard IEC 61131-3 [6]. While there are other tools that could be used to develop this layer, it it important to use one that is well prepared to work with the standard IEC 61131-3 and its definition of Function Block, which CoDeSys is. Furthermore, it supports OPC UA and is a free tool and, because of this, it is a good option for the development of the low level control of the Digital Twin.

2.1.3 IEC 61131-3

Before the existence of IEC 61131, each PLC manufacturer would have their own dialect for their machines, which a lot of the times meant that the transition from one PLC to another one required learning a new programming language (or at least the major differences between them). For example, arithmetic operations were only possible with specific manufacturer-made blocks, which were different between each other. This problem of lack of standardization made it very complicated to integrate different brands, provide support, and overall represented a big obstacle to the development of global and interconnected systems.[7]

IEC 61131 is an international standard for industrial programmable logic controllers and is divided in several parts. Part 3, regarding programming languages, is the one with the most interest for the scope of this project, as it defines five programming languages (Structured Text, Function Block Diagram, Ladder Diagram, Instruction list and Sequential Function Chart) that are more or less adopted by every major PLC manufacturer [8].

[1] https://docs.factoryio.com/manual/

While the standard is extensive and detailed, it is not in any way a rigid set of rules, seeing itself like a group of guidelines. [7]. This means that, while most PLC manufacturers adopt the standard, there are some differences between each manufacturer's systems even though they have all been certified by the standardizing organization [8].

As previously mentioned, the standard defines five different programming languages. Of these, only two are relevant to the scope of this thesis. Instruction List (IL), which is a language very similar to Assembly, is not very used anymore, mainly because of its extreme complexity and the existence of better options. It is maintained and kept in the standard for historical reasons. Ladder Diagram (LD), on the other hand, is still widely used, but it was not used in this work as it was never considered the best option for the specific situation. The main reasons why LD is still a very prominent language are its simplicity to understand and its resemblance with traditional electrical wiring diagrams. This is an advantage because, while a lot of projects need highly skilled engineers to develop, their maintenance can be done by people that are more used to electrical diagrams with a lesser level of academic education. However, this language is not very adequate for complex solutions like the one needed for the completion of the proposed work. Function Block Diagram (FBD) is a graphical language that is based on the connection of Function Blocks and also Functions. It is similar, in a way, to a lot of digital electronics circuits and can be good to program logical algorithms. Since this type of circuits can quickly become very complex and difficult to read for more extensive programs, and because Structured Text can do everything FBD can, this language was also not used during the development of this work. The two most important languages will now be presented.

Structured Text (ST) is a high-level text-based language that is similar to Pascal. It is indicated for complex arithmetic functions, table manipulation and work with tables and text [8] but it is a well-rounded language and entire programs could be written based solely on it. An excerpt of code in ST is shown in Figure 2.7.

```
FOR I:=0 TO 2 DO
    IF (blocks.B[index[i]-1].block_id <> 0) AND (blocks.B[index[i]-1].INST = 0) AND (index[i]=5) THEN
        IF pers.COMUNIC_MES_OK THEN
            blocks.B[index[i]-1].INST:=1;
        ELSE
            blocks.B[index[i]-1].INST:=0;
        END_IF
```

Figure 2.7: An example of ST code written in CoDeSys

Sequential Function Chart (SFC) is actually not a programming language, but a graphical tool for representing state-based and sequential algorithms. It defines three major elements that organize the control program:

- Steps
- Actions

- Transitions

Steps represent the state of the process control. Each step can have one or more **actions** associated with it that represent a set of instructions that are executed during that state. Actions can be thought of as outputs of a specific state of the control process. Finally, **transitions** are the conditions that need to be met for the state of the control process to change.

SFC is very similar to the standard Grafcet [9], in which is based, and one could think these as interchangeable. However, a key difference between these two languages is their nature. While Grafcet is a language mainly used for modelling the behaviour of discrete systems, SFC is a programming language and its actions can be implemented in any language defined by the standard (with the exception of SFC itself)[10](Figure 2.8).

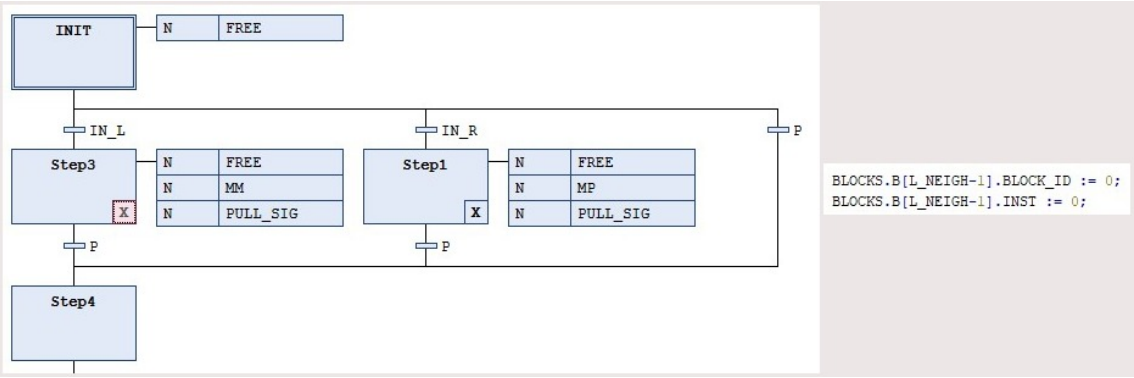

Figure 2.8: An example of an SFC diagram (left) with the action of Step 3 written in ST (right), with CoDeSys

2.1.3.1 Program Structure

An extremely important concept that is defined in IEC 61131-3 is the concept of Program Organization Unit (POU). POUs came to simplify and unify the usage of the previous block types, one of the main goals of the entire standard [7].

There are three main types of POUs defined in the standard:

- Programs
- Function Blocks (FB)
- Functions

As the name indicates, a POU is an independent structural part of the program application. As an example, a very simple program (read: functional program, not to be confused with the POU Program) would be a POU of the type Program, that would call a Function. The standard defines a very large number of FBs and functions, but there is also the possibility of creating custom POUs. These POUs will be summarized below.

2.1.3.2 Programs

A Program is defined in the standard as followed: "... a logical assembly of all the programming language elements and constructs necessary for the intended signal processing required for the control of a machine or process by a PLC system." A Program is the main POU and includes assignments to I/O, global variables and access paths [7].

2.1.3.3 Functions

A Function is a POU that can be coded in any programming language within the standard and yields the same output every time it is called (i.e. has no memory). This is the most basic of all the POUs and it is used in a similar way to functions in a lot of other programming languages (for example C). It is not possible to call POUs of higher complexity (FBs and Programs) and it cannot call itself (but it can call other Functions).

2.1.3.4 Function Blocks

Function Blocks, in contrast to Functions, can have internal memory. This means that the output of a FB call may not always be the same if the input is similar [8]. This is possible because FBs can store values inside their own variables, which also enables multiple outputs.

FBs as defined in the standard follow the three pillars of Object Oriented Programming: Encapsulation, Inheritance and Polymorphism [11].

- **Encapsulation** treats a FB like a black box, where the only visible things to the outside are the inputs and outputs. In this way, it becomes easy to protect code and keep it clean and extensible.
- **Inheritance** is a mechanism by which a FB derives from another FB, inheriting all properties, methods and variables from it. Using this, it is possible to reuse and expand on an existing functionality without re-writing the same code, and it allows the override of methods and properties of the base class (which relates closely to Polymorphism).
- **Polymorphism** is a way to generalize different "actions" that have a base in common. For example, three actuators that, at the high level, are controlled in the same way (using the same methods), but the actual FBs that define them are quite different. Applying Polymorphism, it is possible to develop a control flow that is only concerned with this generic actuator type instead of each one of the three different kinds, simplifying the development process.

In order to use a FB in a POU that allows the use of FBs, an instance of the FB must be declared. This allows the creation of multiple variables of the type FB that have individual and unique values but are all created from the same "template". This is extremely useful for the purposes of a modular approach, as it allows the creation of several similar units in a very quick way. An example of a FB is shown in Figure 2.9.

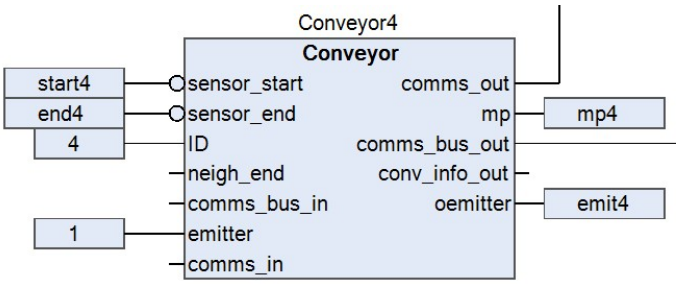

Figure 2.9: A Function Block named **Conveyor**, instantiated with the name "Conveyor4" with inputs and outputs connected to variables, constants, or other FBs (not shown)

2.1.4 OPC Unified Architecture

OPC Unified Architecture (OPC UA) is a platform independent service-oriented architecture that integrates all the functionality of the individual OPC Classic specifications into one extensible framework [12].

Before the creation of OPC UA, each communication standard existed and served their purpose, but they were not connected, which meant that there was no relationship between types of data across systems. Like IEC 61131, OPC UA was born out of the desire to unify and standardize all existing Component Object Model (COM)-based applications, and to simplify the integration of applications like MES, Human Machine Interface (HMI) or Supervisory Control and Data Acquisition (SCADA) systems [13], [14], [15].

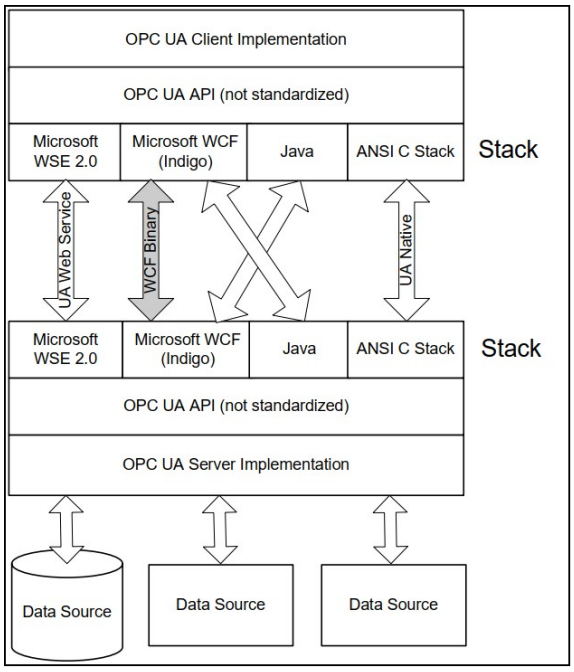

Figure 2.10: An OPC UA communication with APIs in different languages (from [14])

The interface for both OPC UA clients and servers is done via a *communication stack*, that decodes and encodes message requests and responses. This communication stack is accessed by

an Application Programming Interface (API), that is not standardized by OPC UA. This means that, while the messages themselves should be according to the standard, the way to process them can be different from machine to machine (Figure 2.10).

2.1.4.1 Concrete data model

In the Concrete Data Model, the concept of a *node* is the base for the model. Each node may have *attributes* and/or *references*. While attributes describe the node in which they are integrated, references define the relationships between nodes (Figure 2.11), that can even be located in different OPC UA servers.

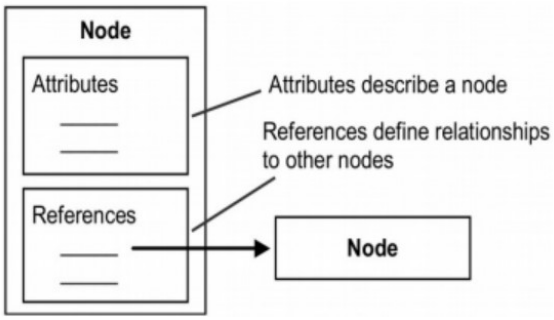

Figure 2.11: An example of the core concepts of the Concrete Data Model (from [16])

OPC UA is a very deep standard. While it is important to have a general idea of how the standard is defined, its vast theory does not greatly concern the main objectives of this document. Both CoDeSys and FACTORY I/O provide a very simple interface for OPC UA, and a superficial knowledge on the communication model is sufficient.

2.2 State of the art

The development of Digital Twins for factory plants is not a new concept. While there is no, to the author's knowledge, software freely available that has all the desired characteristics, there are other paid software developed that are important tools in the professional work with Digital Twins. It is important to have an overview of these solutions to better understand the motivation for the development of this work.

2.2.1 Existing software overview

Most of the existent programs for the simulation of factory plants are quite similar in nature and features, differing in some small aspects. Some of the most used are **Tecnomatix Plant Simulation** [2] by Siemens, **CIROS Studio** [3] by VEROSIM, **FlexSim** by FlexSim Software Products [4] and

[2] https://www.plm.automation.siemens.com/global/en/products/tecnomatix/
[3] https://www.verosim-solutions.com/en/industry/ciros-studio/
[4] https://www.flexsim.com/flexsim/

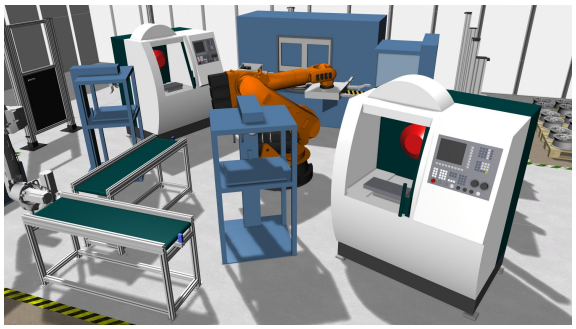

Figure 2.12: A factory plant in Ciros Studios

Simio [5] by Simio. In all of them it is easy to find information regarding Digital Twins and their integration with Industry 4.0, and all of them are prepared for systems simulation and statistical analysis. It is placed great emphasis in the easiness of creation of custom processes (some even allow graphical drag-and-drop models, which require very little programming from the user) and important aspects of factory management like throughput analysis and bottleneck detection are also present. An overview of each of these four solutions will be now be given.

2.2.1.1 Ciros Studio

Ciros Studio is a modular 3D factory simulation software. What this means is that there are multiple packages developed for Ciros which can be combined to provide the features that the user is most interested in using. Some of these packages are:

- Basic Package: Allows for the simulation of electrical, geometrical and mechanical models. It adds functionalities like electrical connections, simple CAD functions and functional kinematics.
- Visualisation Package: Possesses features such as user-defined sectional views, textures from graphics files and visualization of point clouds from laser scans.
- Controller Package: Enables communication through a lot of standards, including OPC UA.

There are many more with a wide range of utilities. For instance, one of the packages allows for programs programmed with SFC and its exportation to PLCs according to IEC. Ciros Studios' graphics can be seen in Figure 2.12.

2.2.1.2 FlexSim

FlexSim is a 3D simulation software well suited for model analysis and optimization. Even though it can simulate complex factory layouts, it is also built with beginners in mind. Its interface (Fig 2.13) allows for drag-and-drop modelling, suitable for users with no background in coding, and the graphical interface is rather simple. It is a very useful tool for the analysis of data and simulation

[5] https://www.simio.com/

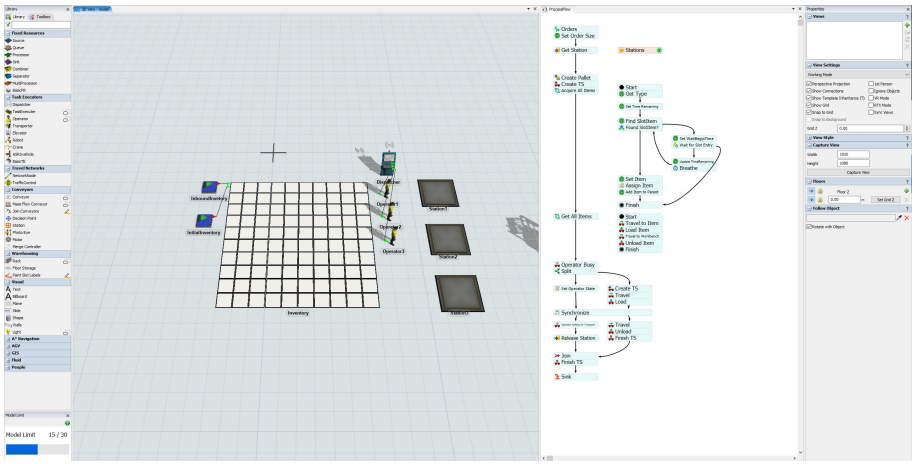

Figure 2.13: FlexSim environment, with its simulation graphics (left) and a program flow (right)

of "what-if" scenarios, ranging from the beginner level to the most professional applications, as it can be read in case studies available at FlexSim's website [6]. According to reviews [7], although it can be easy to start simulating, the learning curve is somewhat steep, and the more complicated simulations can require some learning time from the user.

2.2.1.3 Tecnomatix

Tecnomatix is a powerful simulator with many functionalities of interest in the area of factory simulation and data analysis. According to reviews, it has a lot of capabilities compared to other software and it is one of the strongest software around, but it can be a bit hard for beginners to start simulating, as most of the parameters need to be customized. Even though it depends on the equipment to simulate, the graphical interface can become hard to follow, visually (Fig 2.14).

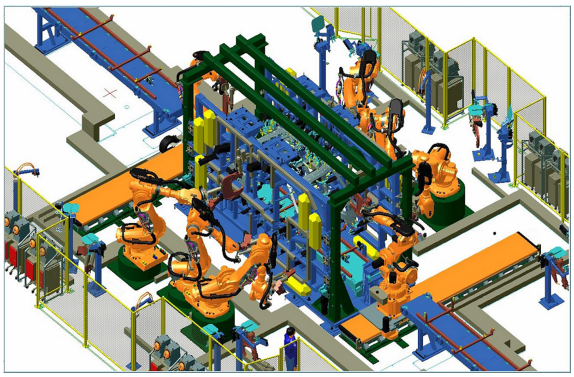

Figure 2.14: Factory plant in Tecnomatix[8]

[6] https://www.flexsim.com/case-studies/competitive-advantage-manufacturing-flexibility/
[7] Reviews were retrieved from https://www.g2.com/
[8] https://oneplm.com/tecnomatix-2/

2.2.1.4 Simio

Simio is a popular software with many functionalities similar to the ones previously discussed. One of its assets is its easy generation of 3D modelling from a 2D model, and it easily allows the visualization of a model in both perspectives. According to reviews, it can be somewhat complicated for new users to learn to use, given its complexity (which, understandably, also grants many powerful functionalities).

Simio has an academic program which can grant its software for free to universities for a period of time, which is also something to have in consideration.

2.2.2 Critical Discussion

As mentioned in the Introduction, one of the motivations for the development of a new solution with FACTORY I/O is the existence of licenses for this software at FEUP. Also, as seen previously, the existing software is powerful and complete, not well prepared for the intended purpose of this work which is, mainly, education, as they are complex with a difficult learning curve and provide a lot of features that, in the courses that will use this work, are not of interest. In this, FACTORY I/O provides a very simple 3D graphical interface and very simple means of connecting it to a PLC (or in this case, a softPLC like CoDeSys) and is overall more user-friendly and simple to use and to quickly grasp what is going on in the factory for the user. It is a factory simulator only, without any overhead of other functionalities, which is, in this case, an important asset.

This work is not, however, only useful for FACTORY I/O. While it is true that the code developed in CoDeSys will be especially adapted to FACTORY I/O elements, the logic underneath the architecture can be, with some adaptations, applied to other simulators similar to FACTORY I/O.

Chapter 3

Architecture of the solution

This chapter will discuss in a bit more detail the problem presented at the beginning of the chapter 1 and the general architecture of the solution developed. It starts with the analysis and establishment of the requirements of the problem followed by the approach taken to deliver the needed functionalities and meet the required objectives. Finally, the pillars on which this architecture is built upon are discussed.

Figure 3.1 contains the global architecture of the entire system and the hierarchy between the different applications for a better understanding of the chapter.

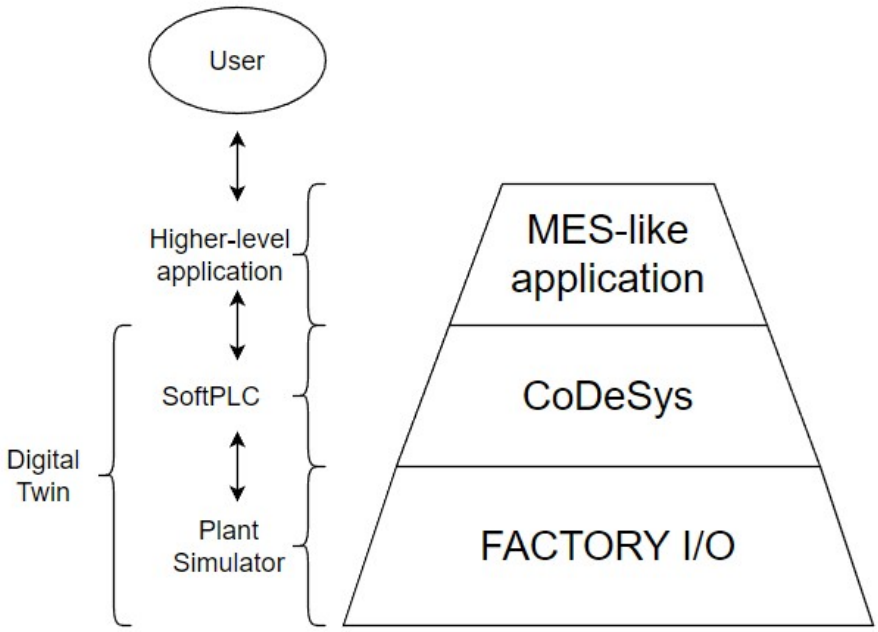

Figure 3.1: Global architecture of the system

Table 3.1: Requirements defined

ID	Requirement description	Group
1	The solution should provide support to the most relevant light-load pre-made scenes in FACTORY I/O - Two Axis Pick & Place (TAPP), Sorting Station, Machining Center and Assembler; to be used as cells in any plant	Plant
2	The solution should include support for individual conveyors to be placed in any logical position given normal restrictions	Plant
3	The solution should include two heavy-load parts - Turntable and Warehouse	Plant
4	There should be no restrictions to the position and connection of the equipment that would invalidate a logical plant in an industrial environment	Plant
5	The global architecture should be structured to permit the future development of additional features without too much work and high complexity	Plant
6	There should exist at least two different modes of controlling the factory: one with as little input as possible from the MES-like application, and another almost completely dependent on step-by-step instructions from the higher-level application	Interface
7	The nature of the data provided by the Twin regarding the process should be similar to the data usually processed by a MES-like application	Interface
8	The interface between the Digital Twin and the application above should be simple and intuitive to use while also being based on existing industrial standards	Interface

3.1 Requirement analysis

The requirements defined can be split, by their nature, in two major groups: requirements related to the plant itself (i.e. the setup of the plant to simulate in FACTORY I/O and CoDeSys), and the ones connected with the final user and the MES-like application that will communicate with it. These groups are referred to as 'Plant' and 'Interface', respectively. The requirements are summarily described in Table 3.1 along with an identification number for easier reference.

3.1.1 Plant requirements

All the requirements inside this group are, as mentioned previously, connected with the factory developed inside FACTORY I/O and its control in CoDeSys. It was defined that the most important and relevant kind of equipment available in FACTORY I/O, considering that the main objective of this work is its use by students who are more interested in developing projects similar to already existing projects in courses like Industrial Informatics (at M.EEC/FEUP) where the type of process resembles an assembly line, is composed of the light-load parts. Because it would be

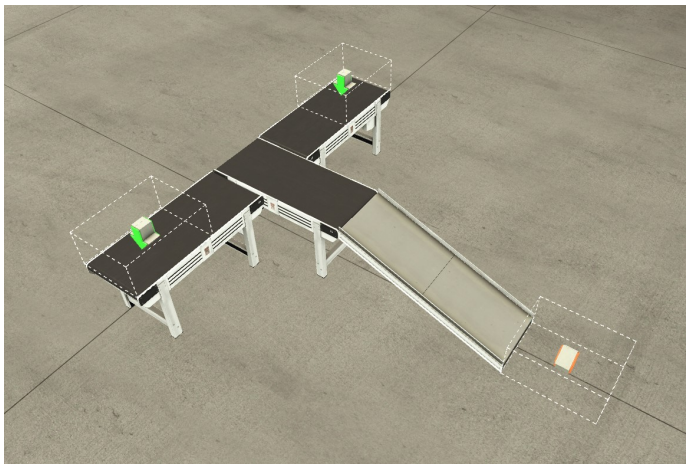

Figure 3.2: Perpendicular connection between light-load conveyors

nearly impossible to assure that every individual equipment from FACTORY I/O would be able to be used individually (i.e. without being previously grouped with other parts) due to its extreme complexity and lack of time, it was defined that the only equipment which would not grouped with any other machines would be the conveyors. The remaining equipment would be divided into groups with specific configurations and connections between the parts which would represent a cell, or a premade scene from FACTORY I/O that accomplished a specific task. For instance, the cell "Assembler" (which also acts as a TAPP) is nothing but a conjugation of conveyors, some position aligners and the equipment Pick & Place. Instead of developing general interfaces for these parts individually and then connecting them together, it was defined that this specific configuration (similar to a premade scene on FACTORY I/O) would represent a cell responsible for assembling lids and bases together (operating as an Assembler) and outputting them through whichever conveyor necessary (operating as a TAPP). This way, it only would only to be developed one interface with the other equipment for the cell , without accounting for all the possibilities if each individual part were to have independence.

The major difference between cells and individual parts (namely conveyors) brings us to requirement 2. While in cells there is a defined flow from input to output, conveyors (and turntables) are prepared to work in any configuration (i.e. any direction).

Because heavy-load conveyors are simpler than light-load conveyors (as there is no option to connect them perpendicularly to each other as seen in Fig 3.2 with light-load conveyors) the development of the former is assured by the development of the light-load conveyors. For the heavy-load parts to be of interest, it would need to be developed support for at least both the Warehouse and the Turntable, hence requirement 3. The Turntable is not strictly necessary to utilize the Warehouse efficiently, but it enables the option of much more flexible paths in the factory which could be used, for example, if the user wants to develop a path-finding algorithm or wants to develop anti-deadlock strategies, among other things.

Requirement 4 ensures that the solution developed can represent a typical factory plant according to the user's needs.

Finally, requirement 5 was defined with future work in mind. By requiring the solution to be modular and structured, it is assured that future expansions (if existent) of this thesis are possible without too much time required. Possible expansions could be more commands supported, other cells/parts supported, or different kinds of data generated.

3.1.2 Interface requirements

The requirements within this group are mainly related to the interface between the Digital Twin and the high-level application (which, for simplicity purposes, will from now on be referred to as "Application") to be developed by the final user. When defining these requirements, it was taken into account the final use of this work and its possible objectives, which are, in part, the ability to process different parts in the factory through the employment of different strategies defined by the Application.

In order to increase the flexibility of control from the Application, there are two extremes when it comes to types of control that should be provided (requirement 6). They are both related to how much input the Digital Twin needs from the Application in order to process a part and are:

- After a part is generated, it should not need any additional inputs from the Application in order to be correctly processed throughout the factory.
- Every step the part takes inside the factory, from simple movement from conveyor to conveyor to transforming actions (like assembling) should be ordered directly by the higher-layer.

The first type requires much less micro-management and can be useful to develop systems that only process statistical data, while the second type could be useful if the user wants to take more control of the factory itself. This work supports both options as well as modes of control that could be placed in the middle of these two extremes. This will be explained further in the subsection 3.2.3

Requirement 7 was born out of the fact that the Digital Twin developed is expected to be working under a higher-level application similar to a MES - maybe not a full-on MES, but a simpler academic MES with similar objectives. To be able to manage the factory efficiently, a MES needs both information related to the state of the plant and statistical information of various parameters, and that is the kind of information that the Digital Twin should provide. This requirement can, therefore, be split into several, more specific, sub-requirements. The Digital Twin:

7.1 must store the number of parts that each equipment processed.

7.2 should store the duration that each part spent on the factory.

7.3 must store the number of lids/bases produced from raw materials, as well as the number of parts assembled together for each equipment capable of doing these tasks.

7.4 must be able to provide information about the state of an equipment at any instant (if it is occupied, what part is occupying it, etc.)

7.5 should store the information of all the parts each Remover removed.

7.6 should report to the Application if any error is detected in the factory.

This information is the minimum required for the Application (and therefore the user) to have something useful to work with. As a typical theme throughout this work, the Digital Twin can be expanded to gather other types of information, but this is the basis that any academic user might need.

Finally, the interface ought to be as simple and intuitive as possible, for obvious reasons. Nevertheless, it should be based on industrial communication standards (like OPC UA), given the context of this work.

3.2 Development Approach

The previous section dealt with what needed to be accomplished in order to fulfil the objectives of this work. The remaining question that needs to be answered is the how. Besides putting together the plant on FACTORY I/O, what else does the user need to do before the Digital Twin can become functional? How to make that additional work as easy and as less time-consuming as possible?

The answer lies on the concept of a Function Block, explained in chapter 2. The base idea is to have a library (i.e. group) of different classes represented by FBs, each representing one type of equipment in FACTORY I/O, that can be instantiated quickly and as many times as needed in order to represent the factory virtually. By representing each equipment by a FB on CoDeSys (Fig 3.3) that to operate only relies on its inputs and outputs, the virtualization of a FACTORY I/O scenario created by the user becomes modular and methodical.

All the additional work required from the user, besides building the plant which is always necessary, is to, on CoDeSys, connect the FBs together and setting up the prepared interface (based on OPC UA) between the Digital Twin and the Application, thus greatly reducing the amount of work normally needed for a task like this.

In order for this concept to work, each FB needs to have defined interfaces, both to exchange data with its peers and to communicate with the Application. On the other hand, each interface should also be easily expandable and easy to work with. These interfaces are supported by the use of pre-defined (developed) structures which the entities in this system use to exchange data with each other. The big advantage of a structure compared to other types of data is that new variables can be added to it without changing the way the variables already existent inside the structure interact with and are accessed by the FBs. This proved to be extremely useful throughout this work and it remains an asset for future implementations.

Finally, in order for the Digital Twin to do anything, it needs inputs from the Application. These inputs are divided into two categories: orders, and commands. In their nature, these two

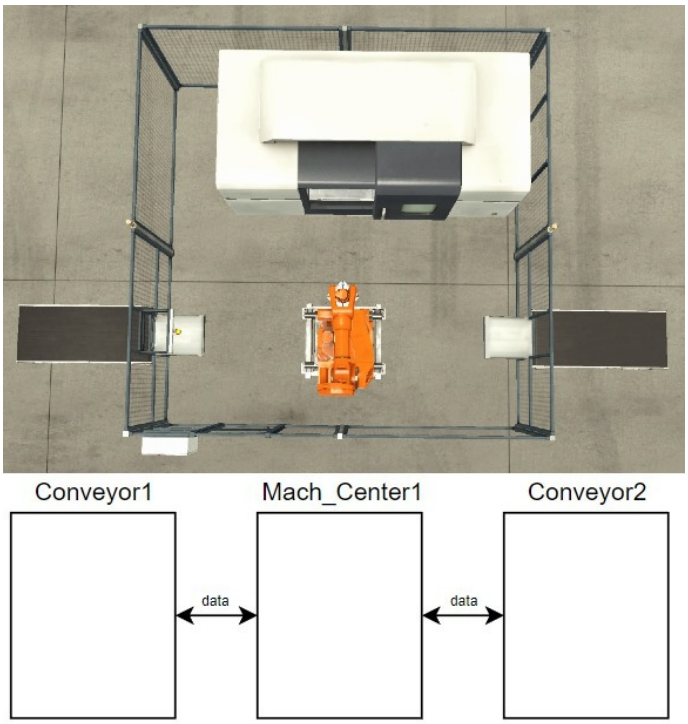

Figure 3.3: Simplified representation of three equipment together, where the rectangles represent Function Blocks

are very similar, as they represent actions for the plant to perform. The big difference between them is that while commands are finished as soon as they are read, orders take some time to finish, once started. Orders are, therefore, actions that implicate physical movement in the factory, like the transportation of parts, while commands deal mainly with information.

3.2.1 Types of information

There are three main types of information carried by structures in this system, each representing one pillar of the system:

1. Information of the state of a Function Block at any given time
2. Information representing commands/orders
3. Information of the state of a part

The first type concerns only FBs and represents the physical plant. From the outside of that FB, this data is merely informative and indicates everything that that FB is doing. This is the type of information that, after correctly interpreted, guarantees that all the different equipment functions properly together as a whole.

The plant, on the other hand, does nothing by itself if not commanded by a higher-level application. At the same time, that same application cannot properly manage the factory if it does not

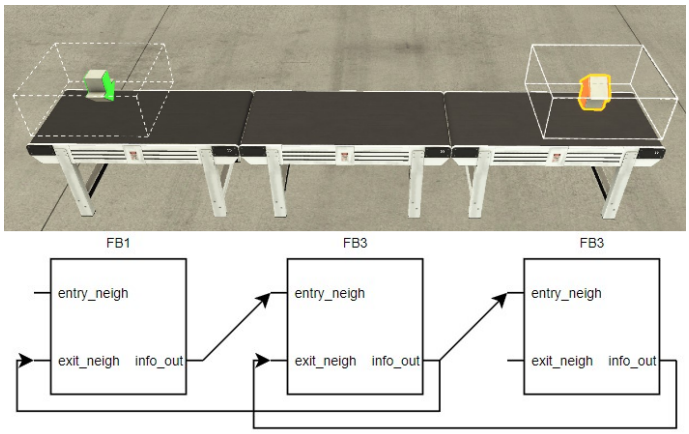

Figure 3.4: Three conveyors connected sequentially (from left to right) with their (simplified) representation as Function Blocks.

get feedback from it. This is where type two comes in. This structure can hold all the necessary information to send orders/commands to and receive responses from the plant and it essentially is what triggers all the actions in the system.

At last, but not least, the information of all the properties of a part is of utmost importance for tying up together the first two types, as everything in this factory revolves around parts, as without their existence the entire system is pointless. This information is what describes to the outside world (from the perspective of a part) the history of that part since its creation, its current state, and its future. In order for this information to follow the part throughout the factory, every FB contains one structure of this type that is filled and cleared according to the presence of a part.

These three structures, and the information they represent, are connected to everything this system deals with and how it functions.

3.2.2 Communication Interface

It was explained that FBs interact with each other and the Application through structures. It remains to be discussed in what way can the FBs be connected to each other and to the Application in order to efficiently exchange data.

In order to decide what actions to take when processing a part that is currently occupying the equipment which it represents, a FB needs to have access to the state information of its neighbours, but only of its neighbours. For a conveyor to decide whether to transfer a part to the next conveyor or not, it does not matter the state of the machine three conveyors away (at least for the level of complexity that this thesis is aiming at). This means that direct connections with the neighbouring FBs is enough. An example of this type of connection between three FBs is depicted in Fig 3.4. In the figure, **entry_neigh** and **exit_neigh** correspond to the inputs for the neighbours' state information and **info_out** to the output. As it can be seen, FB1 and FB3 are not connected directly in any way in the same way they are not connected directly in the factory.

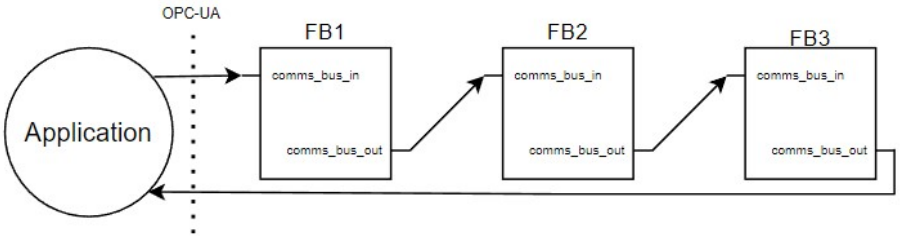

Figure 3.5: Communication line using both FBs and the Application. Other inputs and outputs are not pictured, for simplicity

Orders/Commands, on the other hand, can be sent from the Application to any FB, and Function Blocks should also be able to send commands to other Function Blocks. In this case, it would be impractical to connect every FB to each other. Instead, the solution is based on connecting all relevant inputs/outputs of the FBs sequentially in order to create a line of communication, similar to a data bus. An example can be seen in Fig 3.5. A command generated, for instance, on FB1, will go through each FB in the line until it reaches its destination and it is carried out.

This approach needs to be refined, however. Because, as indicated by the arrows in the figure, this line is unidirectional, a command or order, to go from FB2 to FB1, would need to travel through the entire line, including the Application. This is highly undesirable, as the communication between the Application and the Digital Twin is independent from commands that FBs might want to send to each other. To tackle this issue, a special FB was created (referred to as **Manager** in Fig 3.6) with the sole purpose of serving as a middle-man between the Application and the rest of the FBs. As shown in Fig 3.6, the communication line still exists but the Application is not a part of it which grants the necessary independence between the Function Blocks and the higher-level application.

Even though this type of command-based communication between FBs was not used for the final solution (FBs only send to and receive commands from the Application and not each other), it is a nice thing to have to be able to implement new functionalities if needed.

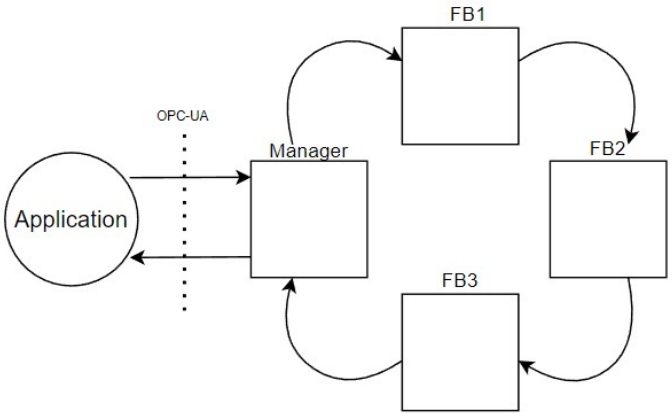

Figure 3.6: Information flow in the system with a FB responsible for communications

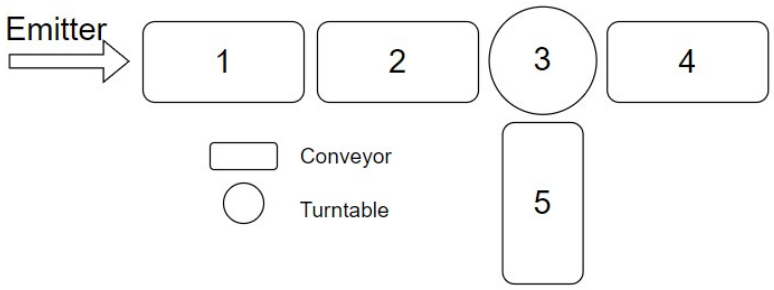

Figure 3.7: A scheme of a factory with different routing alternatives

3.2.3 Parts control

The control and processing of a part in the plant is an interesting topic to discuss. There are a couple of options, ranging from the two extremes mentioned when requirement 6 was presented. The solution ended up being a hybrid in-between which can be used as flexibly as possible between both these types of control.

There are two things that the Application can control when it comes to a part in the factory: path (where it goes), and what each cell does with/to the part. The path is simply what conveyors and cells (each identified by a different number) the part goes through. This path can be generated automatically by the Digital Twin by only indicating the final equipment, or it can be ordered one equipment at a time by each new command. These are the extremes, but there is also a middle ground - it can be decided one sequence at a time. Instead of indicating only the next equipment, the Application can indicate the next sequence of conveyors/cells that the part should go through and, once that sequence is done, indicate the next one. This "sequence" can be composed of solely one equipment, so ordering the next step, the next route until a certain point, or the entire final sequence, can be done in the same way.

Consider the representation of a factory layout in Figure 3.7. If a part is placed on conveyor 1 and the Application decides it should go to 4, it can either indicate the final conveyor (4) and the part starts moving automatically, or it can indicate conveyor 2, wait for the part to reach it, and then indicate conveyor 3. After the same process, finally indicate conveyor 4. This can be useful if an evaluation of the state of the factory is needed at every step, but what if the control policy is such that new orders are only needed on intersections? In this case, the first path defined would be 2-3 and then 4.

When a part reaches a cell, other types of actions can be taken. These actions only make sense when a part is present, so they are intrinsically connected to it. Nevertheless, the same logic used in routing can be used here: The action that should be taken once a part enters each cell can be already embedded on the part when it enters the cell, or can be assigned on the spot. As an example, a Machining Center can be considered: the action "turn the raw material into a lid" can either be part of the information carried with the part, or it can be given to the cell once the part arrives.

These two aspects together and the flexibility with which they are developed guarantee many options of control to Application, and the user.

3.3 Conclusion

In this chapter it was given a detailed view of the architecture of the solution developed. The two main characteristics of the proposed architecture are flexibility, by employing Function Blocks that rely only on their inputs and outputs to perform their tasks, and expansiveness, through structures of data that are easily utilized and built upon.

With this architecture, a Digital Twin can be set up for an arbitrary plant quickly and almost intuitively by connecting FBs together, with an interface for a MES-like application ready to be used as soon as the connections are done.

The concrete implementation of the architecture presented will be discussed in the next chapter.

Chapter 4

Implementation

In this chapter a comprehensive look at the way the different aspects of the developed Digital Twin work will be taken. Before each Function Block is explained, a thorough overview of the most important elements of the implementation, like orders developed and algorithms relevant to all FBs, will be done. Finally, the most concrete details of the implementation will be explained.

4.1 Preliminary information

4.1.1 Orders and errors

As explained in chapter 3, every action in the system has its roots in commands or orders, as without them the Digital Twin would not do anything. In order to better understand what each FB needs to be able to accomplish, it is important to first understand what orders (and correspondent errors) are supported by the FBs. Table 4.1 contains all the already functional orders (in this chapter, in order to avoid constantly referring the pair orders/commands, the act of sending information will be referred to as simply "orders") in the Digital Twin. There are two main types of orders:

- Orders that retrieve information from the factory
- Orders that send information to the factory by altering variables in the interface structures

Because there are orders specific only to certain machines, not all orders are read by every type of equipment and column three ("Read by") contains which equipment can interpret which order. To note that the orders indicated as readable by "all" are not readable by Removers. In fact, the only order directed to Removers is order 5. Finally, orders 14 and 15 are reserved, as they exist only for an internal initial setup of the factory which does not concern the user, and those ids should not be used for future orders, which can be developed by the user in the same methodical way the already existing orders were developed.

The orders' descriptions are mostly self-explanatory, but some of them are better understood in the context of the correct FB, which will be done later in the chapter.

Table 4.1: Orders developed

ID	Order	Read by	Description
1	EMIT	Emitters	Emit a part with the appointed properties
2	NEW_PATH	All	Change the target's part's **routing_mode** and indicate the next path
3	NEW_CONFIG	Sorters	Change the Sorter's sorting configuration
4	ASK_INFO	All	Retrieve the target equipment's current state
5	ASK_REMOVED	Remover	Retrieve an array with all the parts removed by the Remover
6	OUT_WH	Warehouse	Indicate the part(s) to be removed and stored in another location (if desired)
7	IN_WH	Warehouse	Indicate the store position of the part waiting for orders in the Warehouse
8	NEW_PART	Mach_Center	Change the property **part_type** of the target's part
9	NEW_PROPS	All	Change the properties **op**, **final_base**, **final_lid**, **routing_mode**, **store_position** and **path** of the target's part
10	NEW_OP	Mach_Center/ Sorter	Change the property **op** of the target's part
11	NEW_FINAL	All	Change the properties **final_base** and **final_lid** of the target's part
13	NEW_PRIOR	Turntables	Change the priorities of the Turntable's entry conveyors
14	RESERVED		
15	RESERVED		

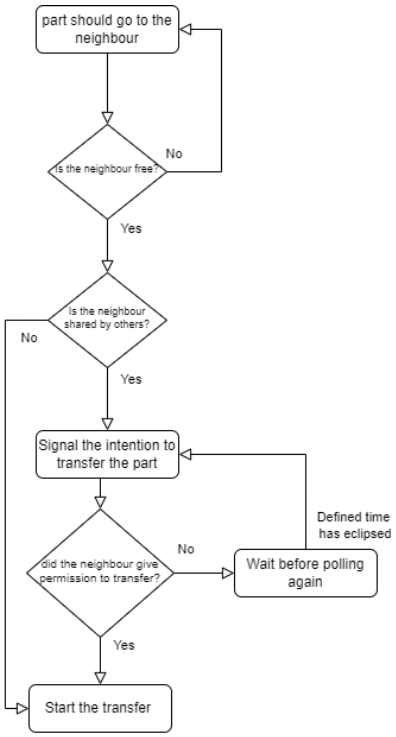

Figure 4.1: Decision-making algorithm for the transfer of parts between equipment

Each order has an associated status which indicates if the order is running (i.e. being performed), waiting to be dispatched, finished, or if an error occurred, with the status depending also on the type of error. Table 4.2 contains the different order status developed for this work. These are carried with the order, as explained previously, and are the way to the FBs to give feedback to the user on what it is happening in the factory. Most of them are simple responses to specific orders and are self-explanatory, but it is worth discussing status 2 and 4. Here, the distinction between the concepts of "order" and "command" is important. Only orders can have the status **FINISHED** while commands can only have **PROCESSING**. By their nature, commands are finished as soon as they are processed (admitting that they are valid) but they are labeled as **PROCESSING** instead of **FINISHED** to distinguish them from finished orders. This was made for a specific use of these status when managing feedback which will be explained in detail when discussing the FB **Orders_Manager** - section 4.2.

4.1.2 Decision-making algorithms

4.1.2.1 Part transfer

The most common decision-making process that every equipment has to go through is the process of transferring parts to neighbours. Figure 4.1 shows this process. This happens every time an equipment is occupied with a part that has the next step of the routing sequence defined, and it is time to carry it out.

Table 4.2: STATUS available

ID	STATUS	Description
0	NONE	The default value. It indicates that the order has not started processing
1	PROCESSING	The order is being processed in the factory, or the command has been read successfully
2	WRONG_PATH	The next conveyor/cell in the path sequence is not a valid ID
3	INVALID_ORDER	The order cannot be read by that type of equipment
4	FINISHED	The order has finished successfully
5	STORAGE_FULL	The position chosen (in the Warehouse) to store the part is already occupied
6	STORAGE_EMPTY	The position (in the Warehouse) chosen to remove the part from is empty
7	NOT_RAW_MATERIAL	The part at the entry (in the Machining Center) is not a raw material
8	OF_NOT_RECOGNIZED	The operation indicated is not valid

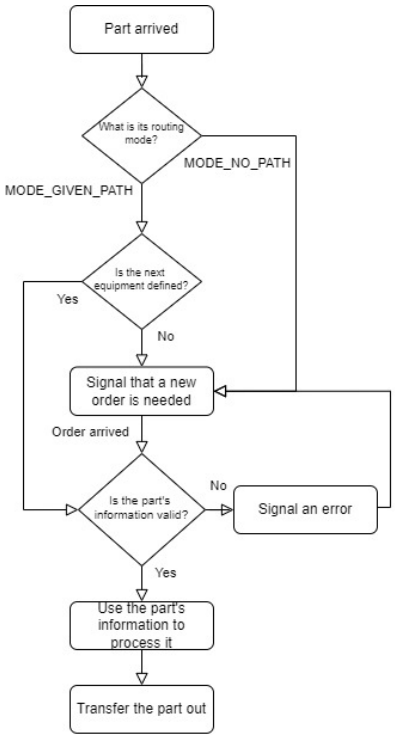

Figure 4.2: Decision-making algorithm for the processing of parts

4.1.2.2 Part processing

As explained in the previous chapter, the are two modes that each part can be defined to have once inside the factory, which grant different levels of control to the Application. These are called MODE_NO_PATH and MODE_GIVEN_PATH:

- MODE_NO_PATH is the mode where a cell that receives the part assumes that there is no information of its interest with the part so it waits for a command to change that information, and only then processes the part.
- MODE_GIVEN_PATH is the mode where any equipment that receives a part tries to move it towards the next equipment in the routing sequence defined for that part. If the equipment that is occupied with the part is a cell, it tries to process the part according to the information already stored with the part.

The algorithm for the processing of parts is shown in Fig 4.2. To note that, if the equipment is a conveyor, it skips the processing of the part, as conveyors do not process parts, and just transfers it out.

4.1.3 Sensor setups

To be able to detect the parts moving through the factory, each conveyor/cell needs sensors managed by their respective FBs. While cells have these sensors in fixed positions (normally, sensors

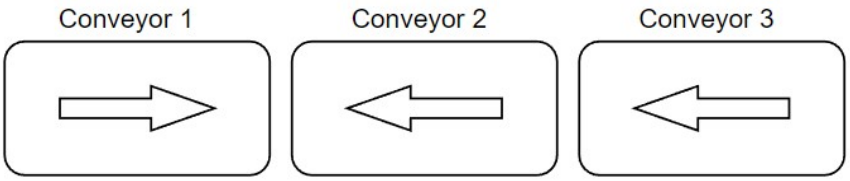

Figure 4.3: Representation of conveyors positioned with different orientations

to detect the entry of the part and sensors to detect the exit from the cell), conveyors can have the sensors in two distinct setups:

1. One sensor at the start and one sensor at the end of the conveyor
2. One sensor in the middle of the conveyor

Setup 1 is the most efficient, both in the number of inputs/outputs used and in the transfer of parts from conveyor to conveyor. While in setup 1 the conveyor that is transferring the part can stop moving and be free to receive another as soon as its end sensor goes from true to false (which, in practice, means that a part moved to another conveyor), in setup 2 the conveyor needs to wait for the sensor of its neighbour to detect the part. This means that the belt will be turned on for some unnecessary extra time and that each FB will need inputs to connect the neighbours' sensors. Nevertheless, this saves the placement of some sensors and could be useful in certain situations. This work supports both these options.

4.2 Function Blocks' implementation

Each Function Block's specific implementation will now be exposed in more detail. To note that not all of the inputs/outputs of each FB will be mentioned. Those that do go unmentioned are not relevant to the understanding of the specific workings of that FB and might take part in proceedings that affect every FB (which will be explained in a different section) or are only important to understand from a factory-building (i.e. user) perspective. The explanation of how to actually use each FB will be made in the next chapter.

Before entering the most concrete explanations, it is important to clarify some nomenclature used in this work regarding orientation of equipment. As explained before, each conveyor can be configured to transport parts in two directions. These are called positive and negative directions. Figure 4.3 represents an example of a possible configuration of three conveyors, with different orientations.

The direction is defined relatively to each conveyor and in the figure it is defined by the arrows, with an arrow pointing towards the positive direction. In this work, this orientation defined the "entry" of an equipment and the "exit". Table 4.3 shows what are the names of the neighbours from each conveyor's perspective. To note that this also applies not only to conveyors but to all equipment in the factory.

Table 4.3: Nomenclature of the equipment from each conveyor's perspective

Conveyor	entry neighbour	exit neighbour
1	NONE	2
2	3	1
3	NONE	2

It is important to keep this in mind throughout this chapter as these definitions are the core to many explanations that will be given.

4.2.1 Orders_Manager

The Function Block **Orders_Manager** (Fig 4.4) is responsible for receiving the orders from the Application and forwarding them to the rest of the FBs, as well as for receiving the responses from the FBs and making them available to the Application. It does not have any independent actions of itself, with the exception of two orders actively sent by it at the startup. These orders are the ones referred to as reserved (14 and 15) when presenting the developed orders.

The **Orders_Manager** uses an array to lay out the responses regarding the orders' status to the Application. Currently, the **Orders_Manager** is filtering all communications from the FBs and, although storing internally all feedback, only places in the array that the Application can access orders with the status FINISHED. This is the reason why commands do not have this status, so that the Application knows that as long as no error has been detected, the command was processed correctly. This way, the Application has an array with only the errors and the finished orders easily accessible. This is not strictly necessary, as the filtering could be done in the Application as there are other ways to go about this subject, and this can be easily changed to work differently if the user so wishes, although it requires some tweaks to the code.

Besides all inputs and outputs necessary for the communication with the Application and other FBs, it has one input **read_sig** which is used by the Application to clear the array where the

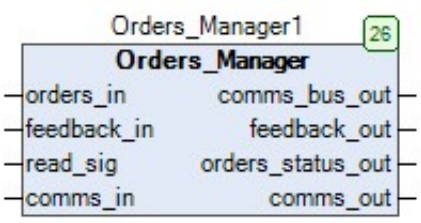

Figure 4.4: Function Block **Orders_Manager**

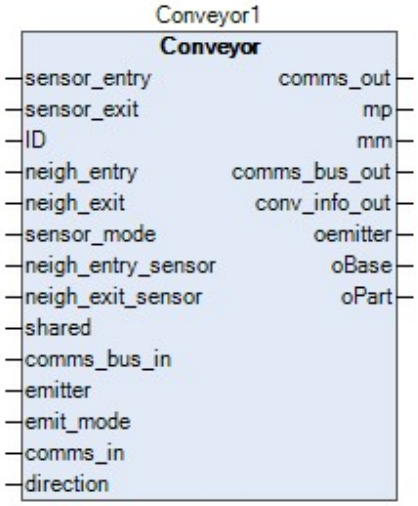

Figure 4.5: Function Block **Conveyor**

feedback from the FBs is stored.

4.2.2 Conveyor

Conveyors (Fig 4.5) are the most flexible equipment of all developed as they can have emitters on top of them, be configured to move parts in both directions and, in the case of light-load conveyors, have multiple conveyors connected to one, perpendicularly. This last property was the reason for the development of another FB called **shared_interface** which will be presented after the discussion of this FB.

Because the transfer process is similar whether the part is coming from the entry neighbour or the exit neighbour, in the next part it is assumed that the part is coming from the entry neighbour and is defined to go to the exit neighbour. It is also assumed that this conveyor's entry is not connected to any other conveyors, as this is the situation of a shared conveyor which is discussed in the **shared_interface**'s section. A simplified diagram of this process is presented in Figure 4.6 and below there is important complementary information of what is happening in each state (represented by a circle).

EMPTY : Wait for the entry neighbour to signal the intention to transfer a part.

RECEIVING : Turn on the motor in the positive direction and wait for the part to arrive, according to the sensor setup.

OCCUPIED : Copy the part's information from the entry neighbour. The conveyor is now occupied. If the next equipment in the routing sequence is valid (which means the entry neighbour or the exit neighbour), start the transfer process.

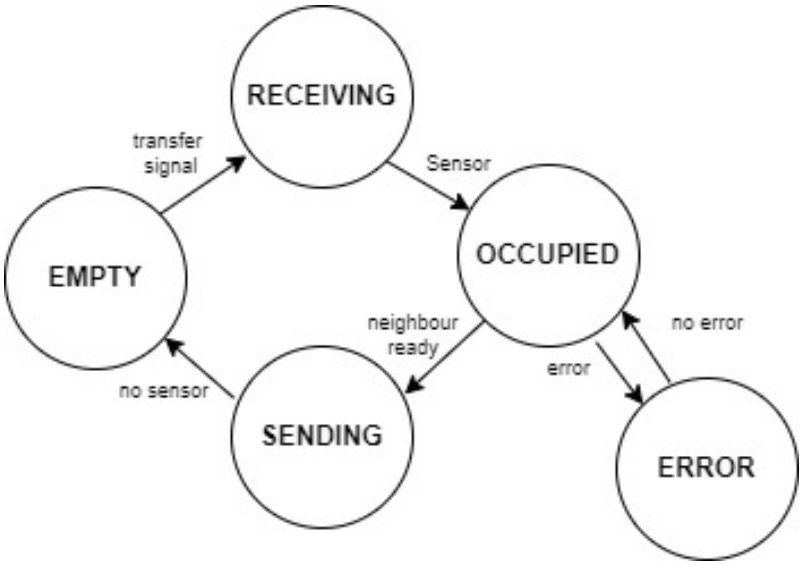

Figure 4.6: Simplified representation of the process of transferring parts between conveyors

SENDING : Turn on the motor in the positive direction and wait for the part to leave the conveyor by reading the falling edge of the correct sensor. Once this happens, clear the part information in the conveyor.

ERROR : Wait for an order to change the path sequence to a valid value.

Conveyors can also have emitters placed on top. This does not change much to the normal workings of this FB and, because of this, there is no FB representing Emitters, with the **Conveyor** having the functionalities of both equipment and becoming an Emitter when necessary. However, this does create the need to be prepared to have a part appear on top of the sensor instead of going through the normal transfer process. If the FB receives an order to place a part and it is labeled as an emitter (which is done by setting a specific input to 1), the FB can either copy the new part info from the order and go to the state **OCCUPIED**, proceeding normally from there, or it can compute the part's path, if a variable that indicates the final conveyor of the sequence is a valid value (and, after that, continue with the same decision making process as previously explained).

The route computed by the FB is the shortest path possible between a source and a target using Djikstra's algorithm ([17]). This is one of many path-finding algorithms that could be used, but this one was chosen for its simplicity, given that its relatively low efficiency at very high node counts is not important for the sizes of the paths to compute. In order to use this algorithm, a virtual graph representing the factory layout needs to be created (an example can be seen in Fig 4.7) which is done automatically at startup. In the solution developed all connections in the graph were considered to have the same cost, but for very complex management techniques this could be changed in future work. As an example, the cost of going through a specific conveyor could go up with the time that that conveyor is occupied by a part, forcing the algorithm to find a "quicker" alternative.

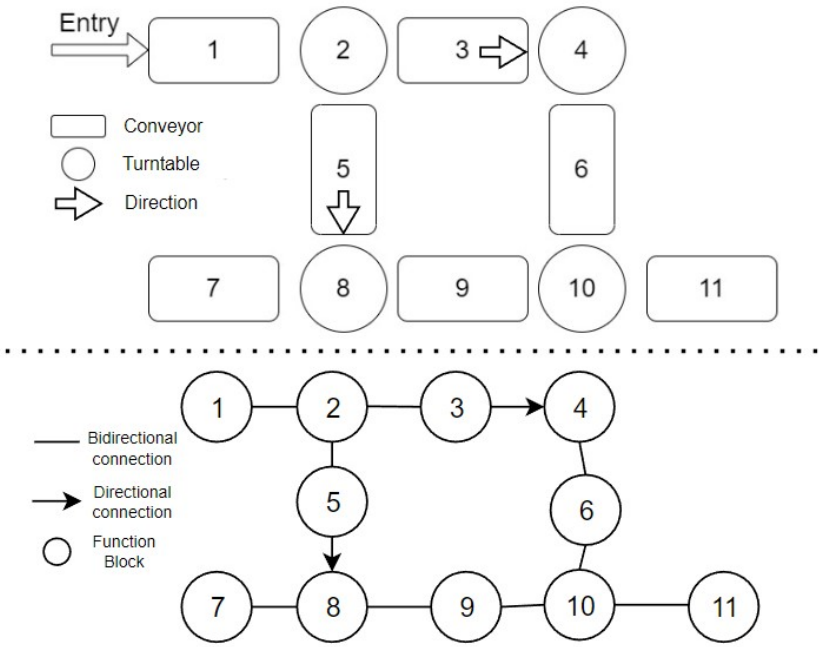

Figure 4.7: Example of a factory and its conversion to a graph

4.2.3 Shared_Interface

The Function Block **shared_interface** is a special FB that, like the **Orders_Manager**, does not represent an equipment in FACTORY I/O. It is used to interconnect multiple (up to four) conveyors perpendicularly to the same conveyor. This FB is necessary because each FB **Conveyor** only supports one equipment's state input for one entry neighbour. More could have been added to the FB, but since this type of connection is more situational, and to not overload the FB **Conveyor** with unused inputs, this FB was created. It is also useful for possible adaptations of other shared equipment, which is more easily done in a different FB than on the FB **Conveyor**. **Shared_interface** is shown in Figure 4.8 and an example is shown in Figure 4.9.

This FB contains four pairs of inputs/outputs used to connect the neighbours that will be connected at the start of the shared conveyor, and one pair for the shared conveyor itself. The

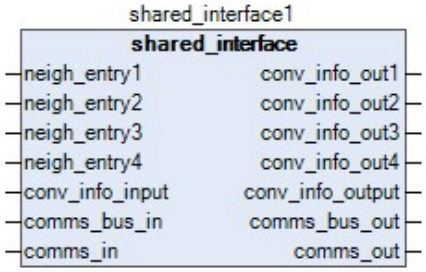

Figure 4.8: Function Block **Shared_Interface**

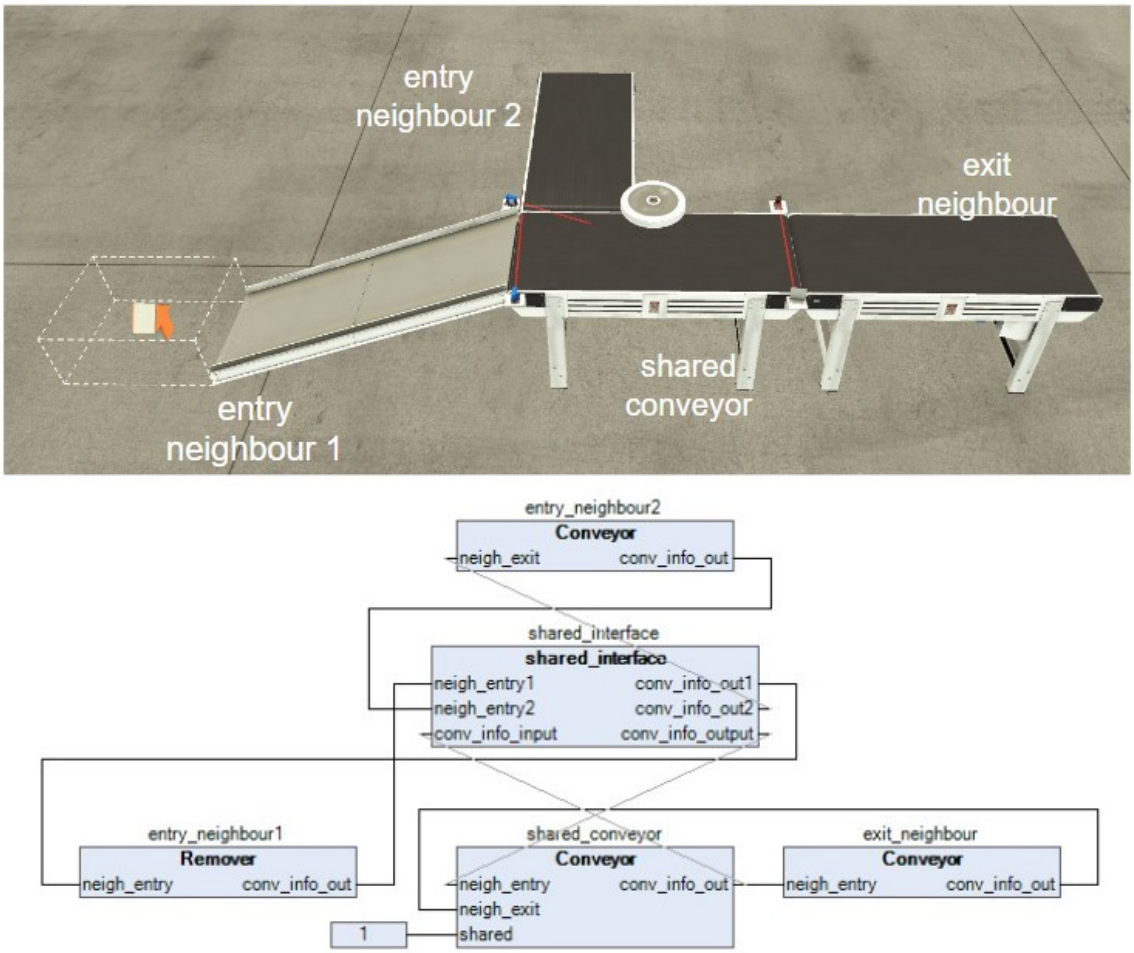

Figure 4.9: A shared conveyor connected to other equipment, labeled from the perspective of the shared conveyor (top) and the correspondent connections (not all inputs/outputs are represented), bottom

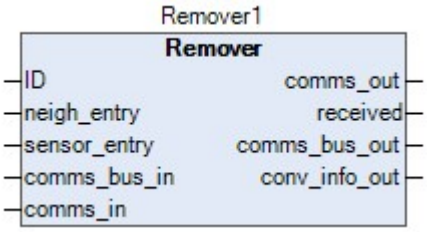

Figure 4.10: Function Block **Remover**

functioning of this FB is as follows:

1. Wait for a signal from any entry neighbour while outputting the neighbour 1's information to the shared conveyor. This is needed because the shared conveyor might want to transfer parts to its entry neighbour and for that it needs this neighbour's information.

2. If the signal detected is the only one detected, output that neighbour's information to the shared conveyor. If more than one signal is detected, give permission for transfer the part to the equipment with higher priority (which is, by default, **neigh_entry1** followed by **neigh_entry2**, **neigh_entry3** and **neigh_entry4**).

3. From now on, the transfer process is the same as discussed previously. Return to step 1 after this is done.

It should be kept in mind that this FB is representing the shared conveyor. Because of this, it needs to be constantly updating the information of that conveyor and feeding it to the equipment that is connected to it. What this means is that, from the perspective of the equipment that shares the conveyor, there is only one neighbour connected and the transfer processes are not any different.

4.2.4 Remover

The Remover (Fig 4.10) is the simplest FB in the entire factory as the equipment only receives parts and physically removes them from the factory. There are some important details regarding the use of this FB when building the factory that will be discussed in the chapter 5 but, from an implementation perspective, there is not much to discuss as the part transfer process is similar to that of normal conveyor. It is important to note that this work does not support Removers on top of Conveyors, and because of this every Remover must be preceded by a ramp.

4.2.5 Machining_Center

The Machining Center (Fig 4.11) is used to transform raw materials into lids or bases and, because it only has one entry and one exit, is a fairly simple FB. This is the first FB that really operates differently with each routing mode as it can take advantage of the properties embedded with the part to process it. The steps taken by this FB are the following:

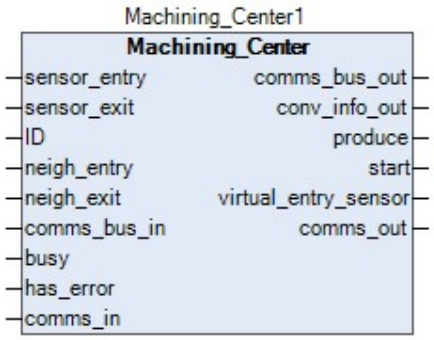

Figure 4.11: Function Block **Machining_Center**

1. Wait for a signal from the entry neighbour.

2. Once the entry sensor goes from true to false, a part is inside. Check if the next equipment is defined in the routing sequence (and if it is valid) and if that equipment is free. If so, transform the part into the correct material and go to step 4 afterwards. If not, signal that an order has been finished (or that there is an error, depending on the situation).

3. Once a new order that indicates the next equipment in the route sequence arrives, transform the part and go to step 4.

4. Start the transfer process and return to step 1.

Because the exit of this machine in FACTORY I/O is inclined (Fig 4.12), which means that a part once freed by the robotic arm falls to the next conveyor, the transformation of a part only begins once the neighbour at the end of the cell is free to receive that part. This could be avoided by stopping the robotic arm mid-air after transformation and waiting for the exit neighbour to be free but, from a real-life point of view, it would be highly undesirable.

4.2.6 Sorter

The Sorter (Fig 4.13) is a cell that can be used for part transportation only or to sort parts according to two properties: colour (blue, green or metal) and shape (lid, base or raw material). The type of sorting can be changed by the correct command at anytime while the system is running, for

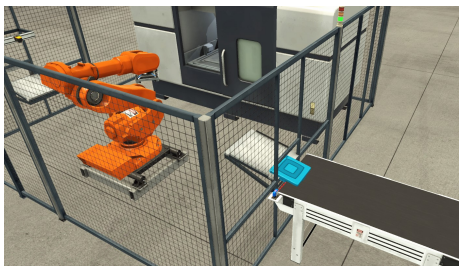

Figure 4.12: Exit of a Machining Center

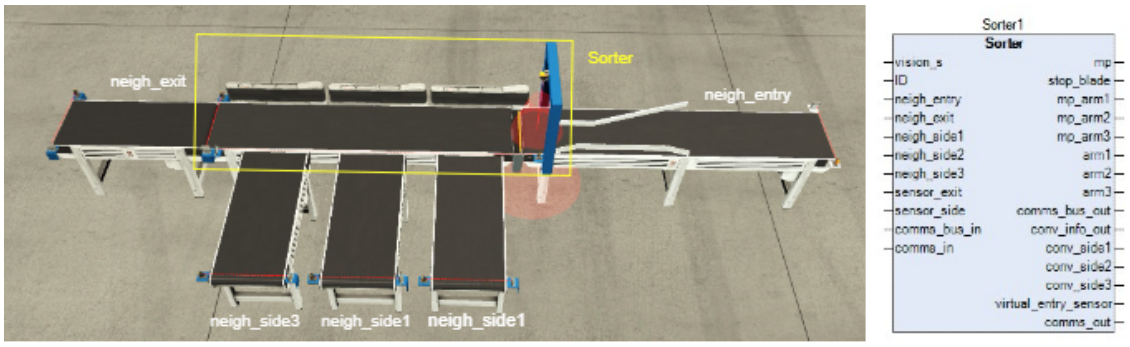

Figure 4.13: Sorter connected to dummy conveyors (left) and the FB **Sorter** (right)

maximum flexibility. Because the detection of these properties is done automatically by a vision sensor which outputs a number according to the property detected, the steps taken by this FB to process a part are very similar to those of a conveyor, with the difference that the output can be one of four different options (at maximum, as not all of the exits need to be occupied).

As usual with cells, it makes a difference which routing mode is present in the part. If it is MODE_GIVEN_PATH, the part is automatically sorted according to the type of sorting set. If it is MODE_NO_PATH, the cell waits for an order to either indicate the next equipment, or to indicate to use the vision sensor. This is done via changing the part property "op" to the keyword BY_COLOUR or BY_SHAPE, according to the intended filtering. Once the next equipment in the routing sequence is set and valid, all this FB has to do is signal the relevant neighbour, turn the robotic arm to direct the part to one of the side neighbours, if applicable, and transfer the part according to the transfer process already described.

To simplify the management of all the information, this FB possesses pairs of inputs/outputs for each possible exit/neighbour which has the disadvantage of adding a lot of pins to the FB.

4.2.7 Assembler

The Assembler (Fig 4.14) is by far the most complex FB of all those developed and it is the only FB that is given two different ids, one for each conveyor. Figure 4.15 contains this cell as well as the captions for possible neighbours and its conveyors. It is highly flexible and, with the exception of moving parts in the negative direction, can do every combination of part assembling and part outputting through whichever of its two conveyors. This means that a part can be assembled in any of its two conveyors and transferred to any of its exit neighbours. It can also receive a part and output it through any conveyor without any assembling done. In fact, this FB works more like the coordinator between two conveyors with each conveyor having its own specific information and behaving, from the perspective of the neighbours, as a normal conveyor.

The main problem with this FB is the deadlock avoidance and the coordination between the two conveyors. Lets consider a situation where all parts that go through this cell have the routing mode set to MODE_GIVEN_PATH and that conveyor 1 received a part that needs to be assembled. Since, to create an assembled part, a lid and a base are needed, this conveyor is now full and

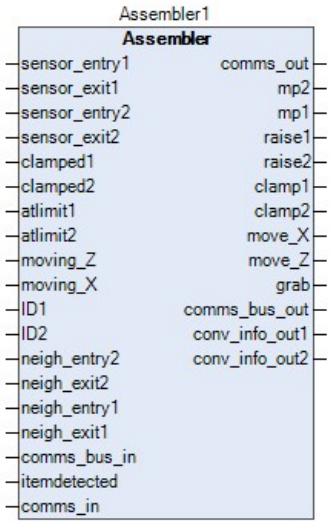

Figure 4.14: Function Block **Assembler**

blocked. Conveyor 2's actions are now limited, as if it receives a part that needs to be transferred to neighbour 1, a deadlock situation occurs. This conveyor should only receive the part needed to complete the assembly, or parts that should continue their path without any assembling, through conveyor 2's exit neighbour. Because of this, upon receiving a request for a transfer from the entry neighbours, this FB reads its entry neighbours' part information before signaling them to transfer parts and this only happens when no deadlock is bound to occur.

On the other hand, this is the only FB that the user needs to pay close attention to when using the routing mode MODE_NO_PATH. Since this is a mode that is meant to give more control to the Application, this FB assumes that if the part asking to enter the cell has this mode it should be accepted immediately regardless of its destination. This also makes sense, because in this mode it is not guaranteed that the part will have the information necessary to decide whether to assemble it or transfer it out. After entering the cell, the conveyor will wait for a command to change the parts' information and once it is valid, process the part accordingly

Finally, if both start conveyors signal the intention to transfer their parts at the same time, it is always given priority to conveyor 1, which means that if the part in the entry neighbour 1 needs to

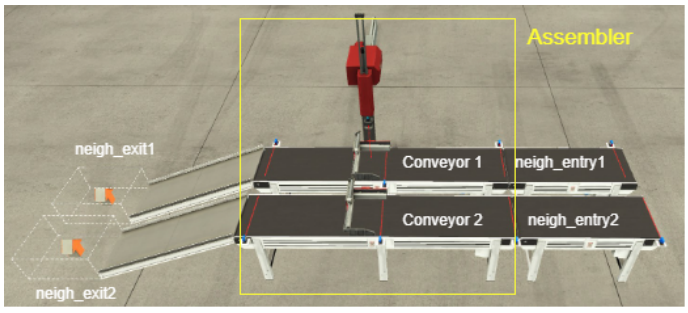

Figure 4.15: Assembler with dummy neighbours

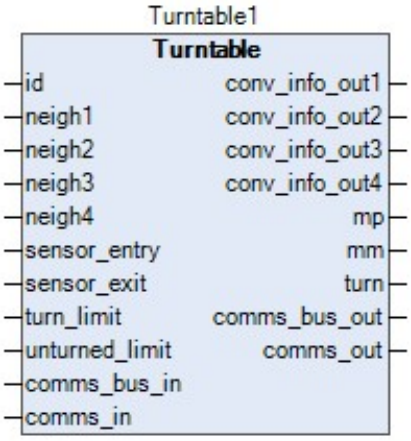

Figure 4.16: Function Block **Turntable**

exit the cell through conveyor 2, part 2 will not be allowed to enter the cell until part 1 is processed.

The transfer process itself is similar to all the other FBs, and the assembling process follows a simple preset sequence of actions, so there is not much to discuss there.

4.2.8 Turntable

Turntables (Fig 4.16) are ready to receive a part from any direction and output it through any of the neighbours. In figure 4.17 it can be seen that all the neighbouring conveyors do not have a specific name (exit or entry). This is because, since it is possible for any of those conveyors to receive or send parts, it does not really matter which label they are given, from the perspective of the Turntable.

Upon receiving signals requesting to transfer a part (as, by default, this is a shared equipment and because of this conveyors wait for confirmation before sending the part in), the FB processes them by priority. This priority is defined in an internal array, and can be changed at anytime by the

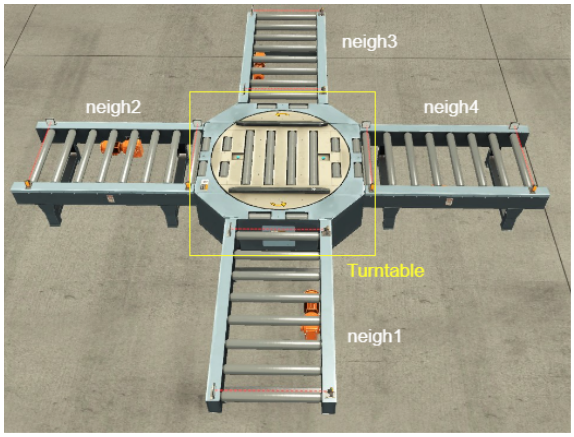

Figure 4.17: Turntable connected to dummy neighbours

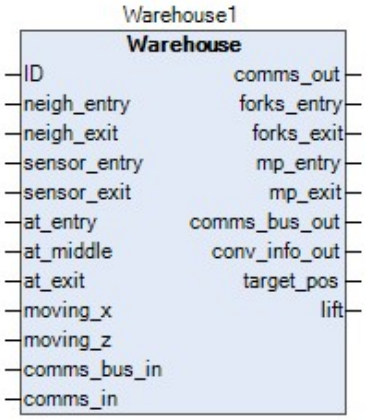

Figure 4.18: Function Block **Warehouse**

Application, through the correct order (NEW_PRIOR). The process of receiving and dispatching parts is, as usual, the same as the other FBs.

4.2.9 Warehouse

The Warehouse (Fig 4.18), besides the entry and exit conveyors, is composed of racks and a stacker crane, used to transport parts to and from the racks. To control this crane, the FB only needs to output a value between 0 and 54 (the maximum number of racks) and the equipment will automatically go to the correct location, with 0 being the reset position and each of the other positive values one storage rack.

This FB can receive parts while it is already "full" (with one part on one of its conveyors). This is done to allow the rearrangement of parts in the Warehouse while one part is waiting for the exit neighbour to be free to receive a part in the exit conveyor. In order to coordinate this information, two FB state information's structures were created, one, as usual, as an output available to the neighbours, and one internal, containing the information of the part in the crane (all the parts stored in the racks are stored in one array and are not relevant to this situation). To the neighbours, this FB is free as long as the crane is not full. The steps taken by this FB are the following:

1. Wait for a transfer signal or removal order. If a signal comes in, go to 2. If a new order arrives to remove parts, if no error is detected, do it. After it is finished, go to 4.

2. Transfer the part in. Once it is in, check its destination. If it is the racks, go to 3. If it is the exit neighbour, start the transfer process and go to 4. If there is nothing, signal end of order and wait.

3. Store the part in the correct rack. Return the crane to the starting position.

4. Go to 1.

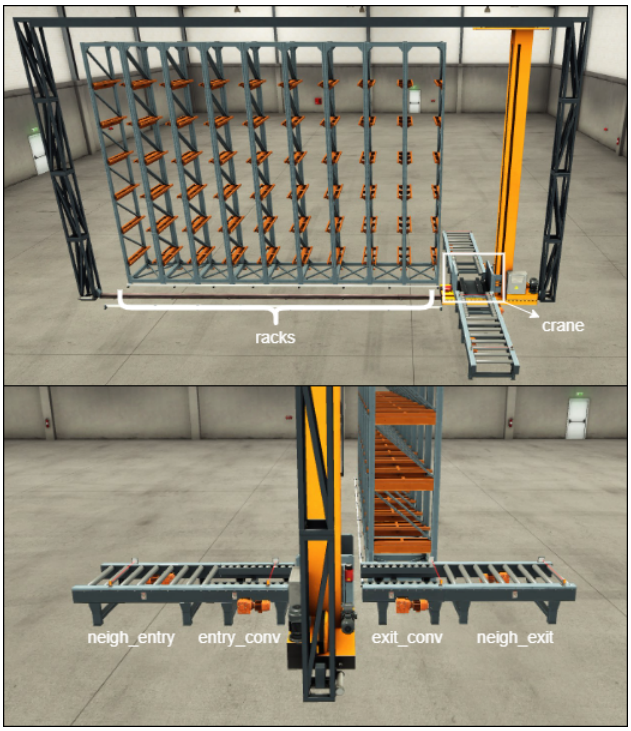

Figure 4.19: Main components of the Warehouse (top) and a side view of the Warehouse connected to dummy conveyors (bottom)

4.2.10 Communication between Function Blocks

All of the equipment in FACTORY I/O can receive and send away parts. It has already been explained what type of information each FB needs in order to accomplish that and the decision-making process that is used to successfully transfer parts without collision.

It was also mentioned previously that there is one input and one output in each FB destined for the establishment of a communication bus (these are referred to as **comms_bus_in** and **comms_bus_out**). To make this concept work in practice, another pair of inputs/outputs was added, called **comms_in/comms_out**. These connections are secured by the structure **comms** with only one variable inside, called **comms.free** (as an important side note, to differentiate regular variables from variables belonging to a structure, the following syntax will be used: <structure>.<variable>). The reason why a structure was used instead of a singular variable is, as usual, explained by the need to think about the future expansions of this work. These structures indicate if the FB is ready to receive another communication and are connected to the previous neighbour of a FB in the communication bus (Fig 4.20). The processing of communications inside a FB is done as follows:

1. Wait for an order.

2. Interpret the arrived order and output the response accordingly. This means that if the order is targeted at that FB it is read and outputted with a different status, and if not, it is simply copied to the output connected to the next FB in the communication bus.

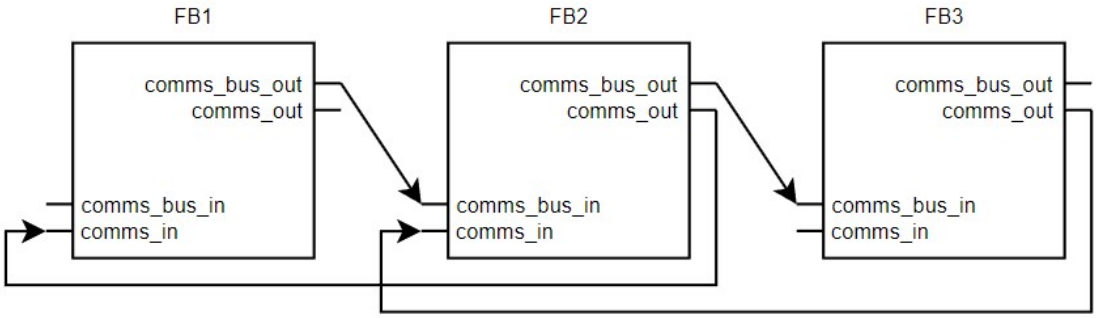

Figure 4.20: Communication connections between three FBs

3. Wait for the communication's neighbour to take the order in.

4. Go back to step 1.

4.2.11 Data structures

4.2.11.1 *equip_info*

This structure (Fig 4.21) was mentioned in the previous chapter and it is the structure that contains all the important information regarding what is happening in a certain equipment at any instant. Every FB has access to this information from its neighbours in order to decide what to do, and it is also useful for the Application to get information on the state of the factory. Each variable will now be explained, as well as the situations where they play a role.

- **id**: An identifier for the FB. This is defined by the user when putting the plant together in CoDeSys (see chapter 5 for more details) and must be different for every FB.
- **free**: This variable is set to true whenever the equipment is free to receive a part. It is set to false otherwise.
- **shared_resource**: Set to true if the equipment is shared by two or more other equipment.
- **ack_free**: This is a special variable that is used by a certain conveyor or cell only if it is a shared resource (defined by the previously discussed variable). It is set to true when, after receiving a request to receive a part, the equipment is ready to take that part in and the neighbour can start the transfer process. This is necessary to impede two equipment from thinking at the same time that their common neighbour is free, thus transferring two parts to the same location and causing a collision.
- **part**: This is a data structure of the type *part* which contains all the information regarding the part currently on that equipment. If no part is going through, this structure is filled with the keyword 'EMPTY' in the relevant positions. More detailed information is given during the explanation of the structure *part*.

```
TYPE equip_info :
STRUCT

id : INT;
free : BOOL;
shared_resource : BOOL;
ack_free : BOOL;
part : part;
exit_sig, entry_sig : BOOL;
entry_neigh, exit_neigh : INT;
waiting_for_order : BOOL;
received : BOOL;
total_parts, lids_prod, bases_prod, assembled : INT;

END_STRUCT
END_TYPE
```

Figure 4.21: Structure *equip_info*

- **exit_sig** & **entry_sig**: These are the variables used to inform the relevant neighbours according to the direction the part is moving (neighbour connected at the exit or at the entry of the equipment, respectively) that a part should be transferred. As explained previously, this does not directly mean that a part will be transferred immediately, as the target neighbour might be shared and it might give preference to another part waiting to be carried in.

- **entry_neigh** & **exit_neigh**: These are variables that store the identifier of the entry neighbour and the exit neighbour, respectively. This information is crucial in two specific scenarios: 1.Upon receiving an **exit_sig** or **entry_sig**, a conveyor or cell needs to know if that signal is meant for itself or another equipment and 2.When automatically setting up the virtual factory layout. This is used when searching for the shortest route between two FBs.

- **waiting_for_order**: This variable is set to true every time the equipment is not performing any order. This can be when it is empty and no part is being transferred, when a part is currently inside but there is no order to perform and finally when an error is detected and it is waiting on a command to correct it. This variable has no use for other FBs but it can be useful for the Application.

- **received**: This is only used by Removers and it is set to true for a short period of time whenever a part is removed from the plant. The reason why this is necessary will be explained in 5.

- **total_parts** & **lids_prod** & **bases_prod** & **assembled**: These variables store some relevant statistical data such as the number of total parts that passed by a certain conveyor (or cell), the number of lids and bases produced (only on Machining Centers) and the number of parts assembled (only used by the Assembler), respectively.

```
TYPE part :
STRUCT

ID : INT;
time_generated : UDINT;
time_removed : UDINT;
process_time : ARRAY[0..49] OF UDINT;
PATH : ARRAY[0..99] OF INT;
path_index : INT;
type_base, type_part : INT;
final_lid, final_base : INT;
final_conveyor : INT;
store_position : INT;
op : INT;
routing_mode : INT;
order_id : INT;

END_STRUCT
END_TYPE
```

Figure 4.22: Structure *part*

4.2.11.2 *part*

The data structure *part* (Fig 4.22) is the structure that represents a part in the factory. It contains all the relevant data to process the part throughout the plant.

- **ID**: An unique identifier. It is defined by the user upon creation of the part by an Emitter.
- **time_generated**: This is a parameter measured in seconds that indicates the time when the part was created. It is computed by the difference between the measured time upon creation and the time that the system that hosts the Digital Twin is on. This is done through the CoDeSys function *SysTimeGetMs* that retrieves the system time.
- **time_removed**: A similar parameter to the previous one but it indicates the time when the part was removed.
- **process_time**: Being an array, it indicates the time spent in each equipment that the part went through. Because of the way this was implemented, the values for each equipment are only valid once they are definitive (which means that while a part is occupying an equipment, the process time of that equipment is not the correct one and should not be consulted).
- **path**: The routing sequence for the part is stored here, by storing the identifiers for the FBs that it has to go through. Once the part is removed, this array stores the entire route that the part went through. While inside the factory, this information by itself is unreliable as this structure can contain equipment **ID**s that the part did not reach yet, so if for whatever reason this information is needed, the variable **path_index** needs to be taken into consideration as well. As with every array in this work, it is good practice to finish the sequence should with a '-1' to indicate when the relevant information to be read is over, if possible.

- **path_index**: This integer, starting at 0, keeps track of the next target equipment for the part, by storing the correct position to be read in **PATH**. If the information of the current FB is needed, the previous position in **PATH** should be read instead. For instance, if the array **PATH** has the values 1-2-3 and **path_index** contains the value 1, this means that the part is currently at conveyor 1 and that the next conveyor in the calculated route is 2 followed by 3.
- **type_base** & **type_part**: It is stored here the information regarding what type of part is this structure representing. It is an integer for easier processing, but each value translates into a FACTORY I/O item. The complete list for this correspondence can be found on Appendix A. **type_part** stores the part itself, like a lid or a box, while **type_base** stores the supporting base, like a pallet.
- **final_lid** & **final_base**: Contains the information of what should be, once removed, the final lid and base of the part. This is used, for instance, to issue orders for Assemblers as it indicates whether a specific lid or base needs to be assembled or if it is already on its final state and can proceed with its path. It is only used by light-load factories and because of this there is no danger of ambiguity with the term "base".
- **final_conveyor**: This variable is used when issuing an order of movement and the route is going to be automatically calculated by the plant.
- **store_position**: This data is only read by Warehouses and it stores the position on the Warehouse where the part should be stored away.
- **op**: Operations to be performed on the part are placed here. This information is used by Machining Centers, when deciding on whether to produce a lid or a base from a raw material, and by Sorters, which utilize this value to decide whether to sort the part using the vision sensor or not.
- **routing_mode**: As the name indicates, the routing mode, previously discussed, is defined here.
- **order_id**: Contains the identifier of the order currently being executed by the part.

4.2.11.3 *Orders*

A list of the variables defined in this structure can be seen in Figure 4.23. This is the structure that carries the information between the Application and the FBs.

- **id**: An identifier for the order. This can be any integer and it is assigned by the Application. It was considered to make the Twin responsible for filling this parameter, but it is important for the higher-level application to know what order corresponds to each ID and that is more easily done if this identifier is defined by the Application.
- **order_type**: An integer that indicates the order or command to be performed by the receiver.

```
TYPE orders :
STRUCT

id : INT;
order_type : INT;
target_group : ARRAY[1..100] OF INT;
new_data : BOOL;
order_status : INT;
remove_part : ARRAY[1..54] OF INT;
new_storage_pos : ARRAY[1..54] OF INT;
sort_by : INT;
sort_order : ARRAY[1..3] OF INT;
priority_order : ARRAY[1..4] OF INT;
feedback : ARRAY[1..100] OF  equip_info;
removed_parts : ARRAY[1..1000] OF part;
part_info : part;
layout : layout;

END_STRUCT
END_TYPE
```

Figure 4.23: Structure *orders*

- **target_group**: An array of integers that keep the order destination(s). There are three different ways this can be used. An user might need to send an order to one target only (for example, to order a specific conveyor to move) and thus should fill the first position of the array with the id of the target recipient. On the other hand, it might be necessary to send a command to various equipment at the same time and that is why this parameter is not simply a single integer. On the other hand, if the user wants to send a command to the entire plant, it can simply fill the first position with the keyword 'ALL' and that command will be passed on to every FB. This can be useful to, for example, ask the entire factory at once the current availability of each equipment (by using the correct order for that purpose, ASK_INFO).

- **new_data**: This boolean signals when the order is ready to be read. It is read only on its rising edge.

- **order_status**: It is stored here the order's status . It is, similarly to other variables, an integer that represents a code, for convenience.

- **remove_part**: This array is only utilized when issuing an order to a Warehouse to remove one or more parts from storage. If the same position of the array in the brother array **new_storage_pos** is also filled, the removed part, instead of being removed from the Warehouse, is stored in the place defined by **new_storage_pos**. Otherwise, the part is removed through the normal exit conveyor of the Warehouse. As an example, if the values stored are 1-2-3 and 2-0-1 respectively, the Warehouse will remove from the cell the part stored inside position 2, and reallocate the other parts to the new positions.

- **new_storage_pos**: It is used only in conjunction with **remove_part**, as explained previously.
- **sort_by**: It is read only by Sorters and it indicates whether the Sorter should sort parts by colour or shape. It is used in combination with the property variable **sort_order**.
- **sort_order**: This array defines, for a Sorter, which exit the part should be directed to, according to the detected property. The first position of the array corresponds to the side conveyor closer to the vision sensor. It should be filled with the appropriate keyword, "BLUE, GREEN, METAL" or "BASE, LID, RAW", depending on the filtering defined by **sort_by**.
- **priority_order**: As explained, Turntables can receive up to three requests for part transfers at the same time (theoretically it can receive four, but that would implicate a deadlock). If the user wishes to change this priority while the factory is running, it is in this array that that new priority is carried. The priorities goes from higher to lower, from position 1 to position 4 of the array.
- **feedback**: This is where a response from the Function Blocks to a specific information request order is placed. It contains data of the type **equip_info**.
- **removed_parts**: Only used by Removers, to retrieve all the parts removed by that equipment.
- **part_info**: This variable is used to carry the order's information to be written on the part of the target FB.
- **layout**: This is only used by the FBs in the initial setup, when creating the virtual layout used to compute the shortest path between two points.

4.3 Interface with the Application

The interface with the Application is done using the FB **Orders_Manager** through its inputs/outputs which are *orders* structures. The structures which are connected to these inputs/outputs are accessed by the Application via OPC UA. The FB has an input **orders_in** to receive the orders from above, and two outputs: one named **feedback_out** which holds the responses from the other FBs to orders sent by the Application, and one called **orders_status_out** which is a buffer that accumulates the information of the finished orders, until it is cleared out by the rising edge of the input **read_sig** (as mentioned in the section of this FB).

The communication between the Application and the **Orders_Manager** is simple. The input **orders_in** is only read by the FB once the variable **orders_in.new_data** changes from false to true, so the Application just needs to write the necessary variables of the intended order to the input structure and once everything is ready, set the variable **orders_in.new_data** to true.

Each order has its own mandatory fields, but every order needs at least one target (by writing to the **target_group**), one **order_type** and one **id**. An example of this communication is shown in the next chapter.

4.4 Conclusion

In this chapter a much more concrete view of the implementation was given. Not all details were discussed, but the most important aspects of the workings of each FB were laid out and explained. In the following chapter, a step-by-step look at the final usage of this work will be taken.

Chapter 5

Tests and validation

In this chapter it will be explained how to actually use the developed work, from the perspective of the final user. Even though the previous chapters were important to understand the inner workings of the system, it is still not clear how to connect all the inputs/outputs of the FBs, what sensors should be used in what situations and how to connect FBs in an exceptional (i.e. not common) configuration. This, and how the system was tested to guarantee that it is in fact viable to work with a MES-like application, will be the theme of this chapter.

5.1 Proof of concept

Throughout the development of this work, each new functionality that was added to a FB was tested immediately after, in the possible scenarios for that specific feature. This "method" assured that the basic objective of each FB was correctly developed and that that FB could be integrated with others in a factory.

What could not be properly tested before the completion of all FBs was the overall user experience, from the perspective of the development of the MES-like application that will control the Digital Twin. In order to validate the interface with the Application developed and to better understand how it could be improved, a proof of concept was developed: a factory was constructed in FACTORY I/O and its virtualization in CoDeSys done, with all FBs represented. After that, an extremely basic and simplified MES-like Application in C/C++ was developed[1], to automate even more the interaction with the Digital Twin since, without this application, the testing was done by manually inserting values directly in CoDeSys. By developing this software, insight was also gained on what could be improved in the Digital Twin and in the way it made data available.

The developed light-load factory is shown in Figure 5.1. The layout was chosen with the idea of creating a straightforward and reasonable flow of parts throughout the factory that both put the factory under stress (i.e. with a lot of parts inside) and used all cells. For this, three Emitters were created (labeled with yellow numbers) and Removers (red numbers) were placed in logical locations to simulate every possible situation.

[1] https://github.com/AndreSousa2417/DigitalTwinWithFactoryIO

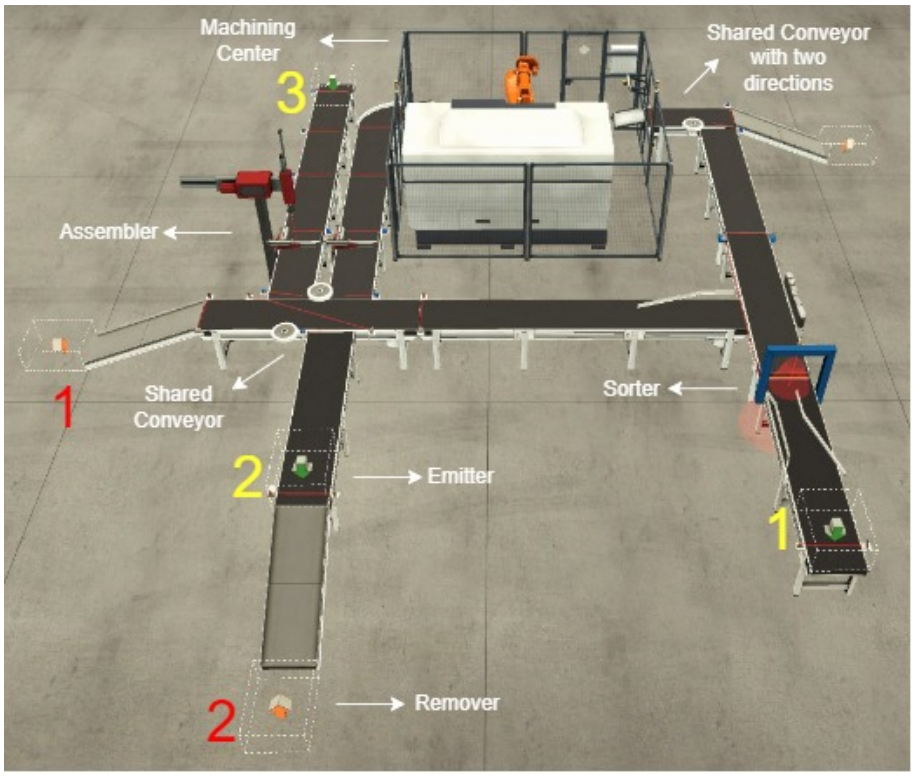

Figure 5.1: The light-load factory constructed for testing

The Application, on the other hand, was programmed to follow the following actions, cyclically:

1. Read the array that keeps the orders' status (and clear it, by writing the **Orders_Manager**'s input **read_sig** to 1).

2. If the Machining Center is waiting for an order with a raw material at its entry, send an order for the raw material present in that cell to be transformed into a lid with Remover 1 as destination.

3. If the conveyor at the end neighbour of the Machining Center is waiting for an order, send an order for the lid that occupies it to be assembled, and set the routing path towards the Remover 1.

4. Ask for the state information of all FBs in the factory

5. Send an EMIT order. The specific order goes through the following, in a loop:

   (a) Order Emitter 1 to create a raw material with the Machining Center as destination, and routing mode defined as MODE_NO_PATH.

   (b) Order Emitter 2 to create a base to be assembled with Remover 1 as destination and mode as MODE_GIVEN_PATH.

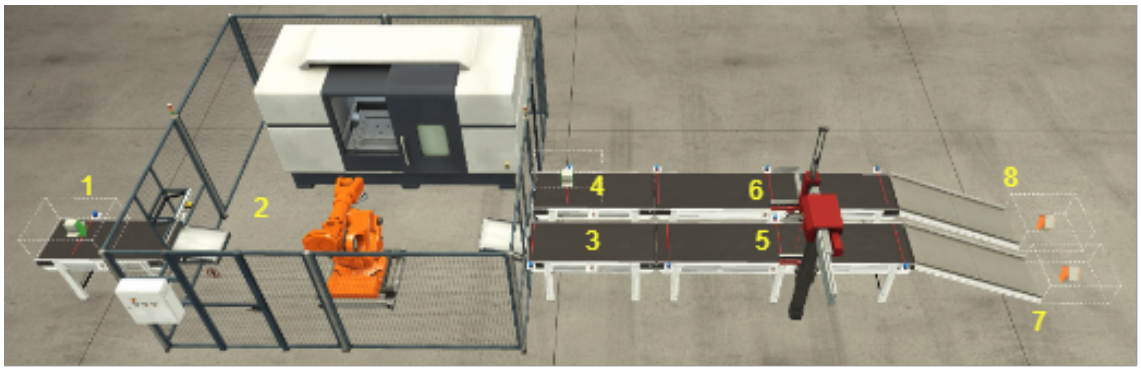

Figure 5.2: Factory plant for this application (each equipment's ID is written in yellow)

(c) Order Emitter 3 to create a lid with Remover 1 as destination.

(d) Order Emitter 3 to create a raw material with Remover 2 as destination.

To note that some of these orders assume information, for instance, that the Machining Center is waiting for an order with a raw material at its entry. In a real Application, additional steps would have to be taken to properly have this information, for example, read the information gathered in step 4 to know which part is waiting. Because the idea was not to create a real full-on Application but to show the concept and test the interface, this is not an issue.

Since the time that an order takes to be completed in the factory is much larger than the cycle time of the Application developed, the factory becomes full quite quickly. This was great for the purposes of testing as it became possible to observe possible deadlocks or situations that were not contemplated when developing each FB individually. Also, because a C/C++ Application can send commands/orders much faster than a human manually inputting information, it also enabled the better testing of the communication both between FBs and between the **Orders_Manager** and the Application.

5.2 User's Perspective

In the previous section it was shown what was made to test the work produced, in order to validate it. In this section an (extremely basic) example of a possible utilization of this Digital Twin from the user's perspective will be described, step by step, from the very beginning, to give a better idea of the steps that need to be taken until there is a Digital Twin ready.

Lets assume, as a starting point, that the user wants to develop an Application for an industrial plant with the following objectives:

- Development of an algorithm to decide where to produce lids or bases according to the number of already assembled parts.
- Development of a management algorithm to manually control every step of the assembly process.

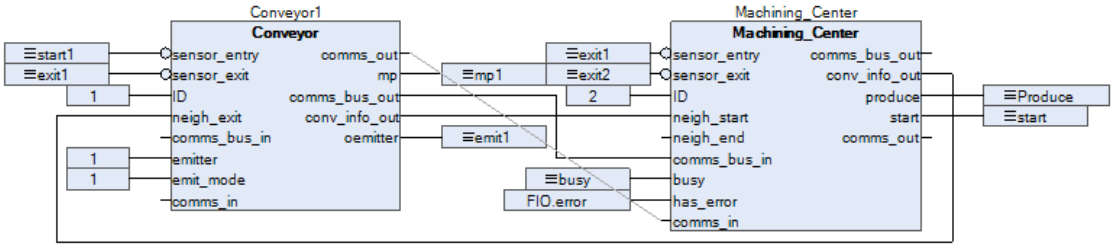

Figure 5.3: First two FBs connected

In order to have a factory with the needed functionalities, a possible layout shown in Fig 5.2 can be put together in FACTORY I/O, with the idea of having Emitter 4 producing lids or bases that may or may not need assembling, and have to decide what transformation to order to the Machining Center, once a raw material arrives at its entry.

Now that the factory layout in FACTORY I/O is done, it is time to create its counterpart in CoDeSys. Firstly, it is necessary to create the variables that will be shared with FACTORY I/O (during the development of this work this was done via OPC UA, but a reasonable alternative would be, for instance, Modbus) and that will represent motors and sensors (In this example, this was done through a list of global variables called FIO). Once that is done, it is necessary to start connecting FBs together. The connections between Conveyor 1 and the Machining Center would be the ones shown in Figure 5.3.

The logic used for the names of the sensors was to finish the name with the equipment ID and start with the position of the sensor regarding to that equipment. To note that, because every FB except Assemblers use negative logic, and in the FACTORY I/O layout sensors with positive logic were used, it is needed to negate their signals. Another thing to take notice is that, since the exit sensor of Conveyor 1 is the entry sensor of the Machining Center, these inputs are fed the same signals.

After doing this procedure to all equipment the plant would look something like the one showed in Figure 5.4. It should be noted how similar the FBs and the FACTORY I/O layouts are, even visually, which is very helpful both when building the Digital Twin, and possible debugging afterwards, in case the user made a mistake making the connections. Now, it is needed to create the FB **Orders_Manager** in order to close the communication line and interface with the

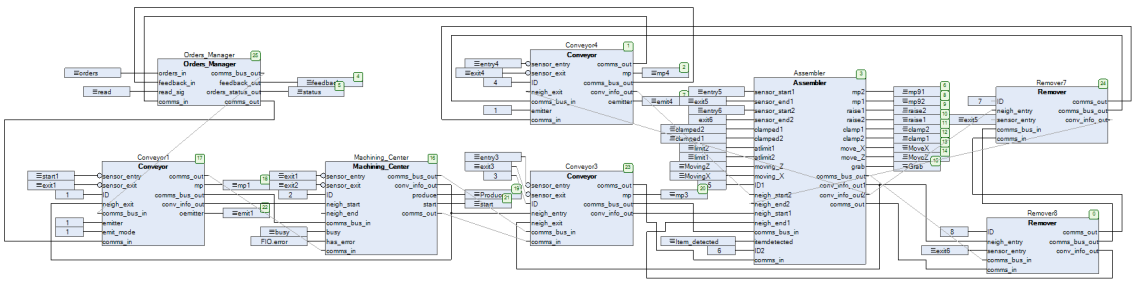

Figure 5.4: All equipment's FBs connected

Application. To interface with the Application, a list of global variables called was created with four variables: **orders**, **read**, **feedback** and **status**. As each name indicates, they are used to send orders in, send the read signal in, retrieve feedback from FBs and access the status array, respectively. These connections to the **Orders_Manager** are depicted in Figure 5.5. These variables will now be shared over OPC UA, in order to be accessed by the Application, which is easily done with CoDeSys.

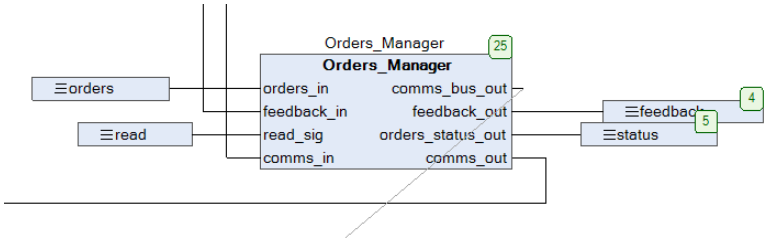

Figure 5.5: **Orders_Manager** as an interface

The Digital Twin is now completed and ready to be used. For example, to create a raw material in Conveyor 1 with destination Remover 7 but be able to decide if that part is turned into a lid or base only when it is inside the Machining Center, the order sent from the Application could be the following (only the variables of interest for this order are shown):

- id = 1
- order_type = 1 (EMIT)
- target_group[1] = 1
- part.id = 1
- part.path[0] = 2
- part.path[1] = 3
- part.path[2] = 5
- part.path[3] = 7
- part.path[4] = -1
- part.type_part = 3 (BlueRawMaterial)
- part.final_base = 2 (BlueProductBase)
- part.final_lid = 1 (BlueProductLid)
- part.routing_mode = 1 (MODE_NO_PATH)

and to retrieve all the parts that were removed from Remover 6 (which is useful to decide on whether to transform raw material into lids or bases):

- id = 2
- order_type = 5 (ASK_REMOVED)
- target_group[1] = 6
- target_group[2] = -1

To note that, even though there are keywords (i.e. aliases) that represent numbers in CoDeSys, these variables are still integers. This means that, to send the orders through OPC UA, all fields must have their numerical representation rather than the more user-friendly keyword. A good way to solve this is to, in the Application (and in the case of C/C++), create a list of #define with all the correspondences before writing any additional code. This way, when developing the orders, it is much easier to understand what is being done.

5.3 Restrictions and general set up information

This section contains a list of general restrictions and important things to know when connecting the FBs together and when using this Digital Twin.

- Parts when generated must not be assigned the ID 10000 as it is reserved.
- When an Assembler mounts a lid and a base together, the final assembled part keeps the properties of the lid with the exception of the variable **time_generated** which takes the value of the older part.
- Every FB, with the exception of the Assembler, uses negative logic for the sensors.
- In Assemblers, the inputs **entry_sensor** are not destined for the sensors at the start of the cell, but to the sensors positioned at the start of the position aligners.
- If the user wants to use the vision sensor placed at a Sorter as the exit sensor for the Sorter's entry neighbour, it should connect the Sorter's output **virtual_entry_sensor** to the neighbour's **sensor_exit**. This is necessary because the vision sensor does not output a boolean value which is needed for an input of this type.
- When placing a Remover, and if its neighbour is using a sensor in the middle of the conveyor (rather than one in the end and one in the start) the Remover's output **received** should be connected to the neighbour's input **neigh_sensor_exit**. In FACTORY I/O it is also needed to place a sensor at the start of the ramp (of the Remover).
- Conveyors can be configured to move in both directions or only one. The FB needs to have this information, and has an input called **direction** for this purpose. It should be set to 0 if the conveyor can only move parts in the positive direction, 1 if only in the negative direction, and 2 if both directions.

- When sending a NEW_PROPS command, the parameters that are not to be changed in the part should be filled with the keyword DONT_TOUCH. It is good practice to do this with every order that can change more than one variable, but not all variables are to be changed with that order.

Chapter 6

Conclusion and future work

As more and more engineering sectors evolve towards automation, intelligent and more adaptable systems are demanded. As seen in this document, companies are stepping forward in the development of software capable of quickly simulating factory plants and easily making available data needed to analyse and improve existing industries - software capable of representing Digital Twins.

The proposed work consisted of picking up this concept and adapting it to a tool that is well prepared for academic purposes at the FEUP. As a starting point, it was chosen to create a Digital Twin with FACTORY I/O specifically, because this simulation software, although licensed, is already available at the University through paid licenses, and use CoDeSys as a softPLC since it is a powerful IEC 61131 automation software which is completely free to use. The choice of these software also mirrored the other important aspects of this work, knowing that it would be used in an academic environment: they are simple to learn, yet complete and complex enough for the development of work of interest in the courses of FEUP.

A critical aspect of this work, and the main problem it seeks to address, is the extremely time consuming task of building a plant in FACTORY I/O and developing all the needed controller logic afterwards. This is aggravated by the fact that, in the majority of the situations, tweaks in the factory layout mean big changes in the controller logic, which implicates more time lost doing something that is not the objective of that work. This is a relevant issue, not only on academic grounds, but on real industrial applications, as it is seen by companies' efforts to develop software capable of embodying Digital Twins in order to improve the efficiency of their factories.

In order to solve this problem in a way suitable for learning, the solution developed is based on a modular approach, with each equipment in FACTORY I/O being represented by a class in CoDeSys. Each class has representation in Function Blocks, which can easily be connected graphically on CoDeSys, in a configuration almost alike the physical factory. This approach of developing each equipment's internal logic in a self-contained, modular way, greatly improves the flexibility of this work and allows for both future expansions by other developers, and enormous time gains when compared to developing each factory from scratch.

To validate the developed work, several basic tests were carried out throughout the development phase, for each individual Function Block. In the end, to take the role of the final user, a complete factory was developed, which gave to the author the experience of building both a full factory and its logic (which confirmed the easiness of this procedure) and insight on what could be improved in the way a MES-like application communicated with the Digital Twin.

Even though all the laid out requirements are supported in this work, there is room for future improvement. The most obvious step towards the expansion of this work is the development of support for the rest of the equipment that FACTORY I/O provides, in order to completely cover this simulator's capabilities. The other is to improve on the interface with a higher-level application, which, in the current form, does not utilize functionalities provided by OPC-UA, like events, and requires the MES-like application to poll the Digital Twin periodically for information. It could also be of interest to develop support for FACTORY I/O's operators/warning devices which, through its lights, buzzes and displays, could provide a visual interface to the user in FACTORY I/O itself.

Throughout this work it was motivating, to the author, to know that this work, besides its academic component, might be actively used by other colleagues in the upcoming years at M.EEC, like other works by colleagues that the author had the opportunity to take advantage of himself.

Appendix A

Lists of Codes

Table A.1: Identifiers for each FACTORY I/O item

INT	**Item**	**INT**	**Item**
0,11	None	9	GreenRawMaterial
1	BlueProductLid	10	Assembled
2	BlueRawBase	12	SquarePallet
3	BlueRawMaterial	13	StackableBox
4	MetalProductLid	14	SmallBox
5	MetalProductBase	15	MediumBox
6	MetalRawMaterial	16	LargeBox
7	GreenProductLid	17	PalletizingBox
8	GreenProductBase	18	Pallet

Table A.2: Identifiers for each order

INT	**Item**	**INT**	**Item**
1	EMIT	8	NEW_PART
2	NEW_PATH	9	NEW_PROPS
3	NEW_CONFIG	10	NEW_OP
4	ASK_INFO	11	NEW_FINAL
5	ASK_REMOVED	13	NEW_PRIOR
6	OUT_WH	14	FORM_LAYOUT
7	IN_WH	15	STORE_LAYOUT

Table A.3: Identifiers for each order status

INT	**Status**	**INT**	**Status**
0	NONE	5	STORAGE_FULL
1	PROCESSING	6	STORAGE_EMPTY
2	WRONG_PATH	7	NOT_RAW_MATERIAL
3	INVALID_ORDER	8	OP_NOT_RECOGNIZED
4	FINISHED		

Table A.4: Identifiers for other terms

INT	**Item**	**INT**	**Item**
0	TO_BASE	12	GREEN
1	TO_LID	13	METAL
6	MODE_GIVEN_PATH	14	LID
7	MODE_NO_PATH	15	BASE
8	BY_COLOUR	16	RAW
9	BY_SHAPE	-1	EMPTY
10	DONT_TOUCH	-2	FROM_VISION
11	BLUE	-7	ALL

Appendix B

Inputs/Outputs descriptions

Inputs/outputs with the same designation from FB to FB are explained only once, as they should be connected by the same logic regardless of the FB.

B.1 Orders_Manager

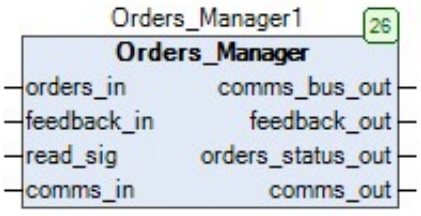

Figure B.1: Function Block **Orders_Manager**

B.1.1 Inputs

- **orders_in**: Orders/commands from the high-level application.
- **feedback_in**: Connect **comms_bus_out** of the last FB in the communication line here.
- **read_sig**: A rising edge of this input clears the output **orders_status_out**.
- **comms_in**:Connect **comms_out** of the last FB in the communication line here.

B.1.2 Outputs

- **comms_bus_out**: Connect to **comms_bus_in** of the next FB in the communication line.
- **feedback_out**: Outputs the responses from the FBs to a command/order.
- **orders_status_out**: Array with the status of the finished orders.
- **comms_out**: Connect to **comms_in** of the next FB in the communication line.

B.2 Conveyor

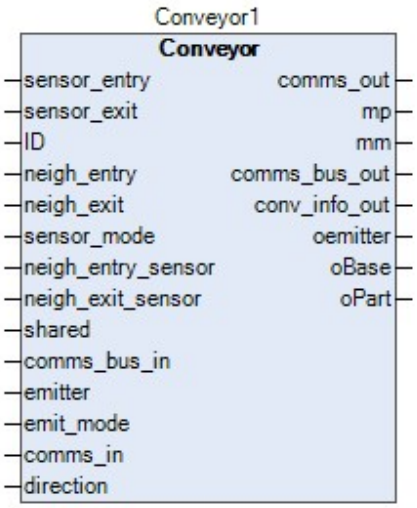

Figure B.2: Function Block **Conveyor**

B.2.1 Inputs

- **sensor_entry**: Signal of the entry sensor from FACTORY I/O.
- **sensor_exit**: Signal of the exit sensor from FACTORY I/O.
- **ID**: ID of the FB.
- **neigh_entry**: Connect **conv_info_out** of the entry neighbour of this FB here.
- **neigh_exit**: Connect **conv_info_out** of the exit neighbour of this FB here.
- **sensor_mode**: Defines the sensor mode for this FB, 0 for a sensor in the exit and entry of the conveyor, 1 for a sensor in the middle only.
- **neigh_entry_sensor**: Connect the sensor of the entry neighbour of this FB here, in case the sensor_mode is set to 1.
- **neigh_exit_sensor**: Connect the sensor of the exit neighbour of this FB here, in case the sensor_mode is set to 1.
- **shared**: Set to 1 if this is a shared equipment, 0 if not.
- **emitter**: Set to 1 if this conveyor has an emitter on top, 0 if not.
- **emit_mode**: Set to 1 if the type of part to be emitted is defined in the controller, 0 if the part is defined in FACTORY I/O directly.

- **direction**: Set to 0 if the conveyor moves only in the positive direction, 1 if only in the negative direction, 2 if both.

B.2.2 Outputs

- **mp**: The command signal to move the conveyor in the positive direction.
- **mm**: The command signal to move the conveyor in the negative direction.
- **conv_info_out**: Connect to **neigh_exit** and **neigh_entry** of the exit and entry FBs, respectively.
- **oemitter**: Signal to turn on the emitter.
- **oBase**: Information of the type of base to emit.
- **oPart**: Information of the type of part to emit.

B.3 Shared_Interface

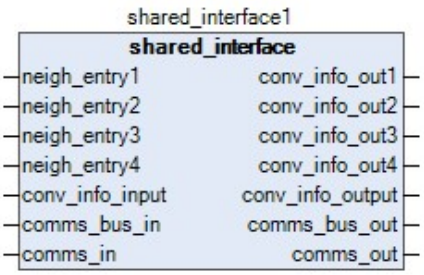

Figure B.3: Function Block **Shared_Interface**

B.3.1 Inputs

- **neigh_entry1**: Connect **conv_info_out** of the FB that is connected directly to the shared conveyor here.
- **neigh_entry2**: Connect **conv_info_out** of one FB connected perpendicularly to the shared conveyor here, if applicable.
- **neigh_entry3**: Connect **conv_info_out** of one FB connected perpendicularly to the shared conveyor here, if applicable.
- **neigh_entry4**: Connect **conv_info_out** of one FB connected perpendicularly to the shared conveyor here, if applicable.
- **conv_info_input**: Connect **conv_info_out** of the shared conveyor here.

B.3.2 Outputs

- **conv_info_output**: Connect to **neigh_entry** of the shared conveyor.

B.4 Remover

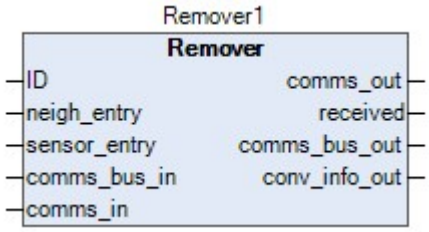

Figure B.4: Function Block **Remover**

B.4.1 Inputs

All inputs have been discussed already in previous FBs.

B.4.2 Outputs

- **received**: Connect to **neigh_sensor_exit** of the neighbour, if its sensor mode is set to 1.

B.5 Machining Center

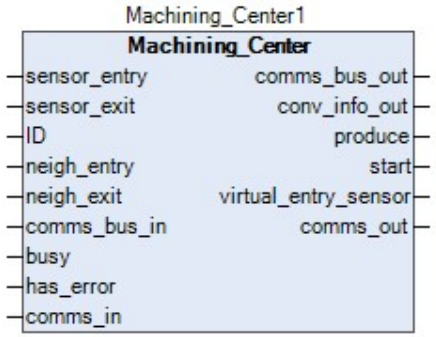

Figure B.5: Function Block **Machining_Center**

B.5.1 Inputs

- **busy**: Connect the "busy" signal from the FACTORY I/O equipment here.
- **has_error**: Connect the "has error" signal from the FACTORY I/O equipment here.

B.5.2 Outputs

- **produce**: Outputs a command signal to the equipment in FACTORY I/O to determine whether to produce lids or bases.
- **start**: Command signal to start the operation of the equipment in FACTORY I/O.
- **virtual_entry_sensor**: Represents an entry sensor in case a neighbouring FB needs it to operate.

B.6 Sorter

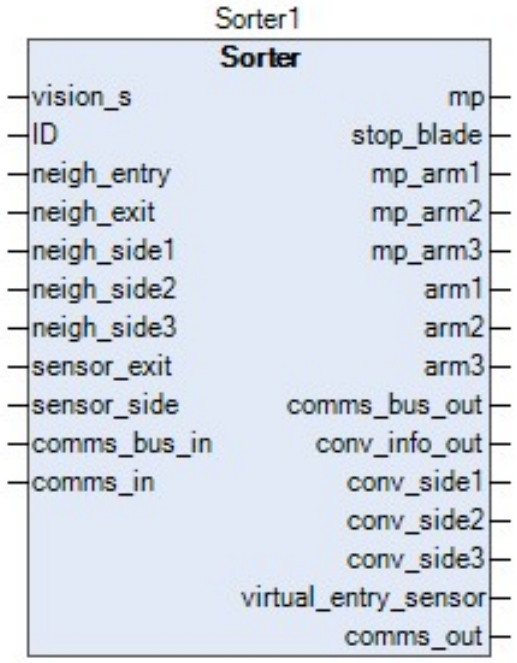

Figure B.6: Function Block **Sorter**

B.6.1 Inputs

- **vision_s**: Signal from the vision sensor.
- **neigh_side1**: Connect here the **conv_info_out** of the side neighbour closest to the vision sensor.
- **neigh_side2**: Connect here the **conv_info_out** of the middle side neighbour.
- **neigh_side3**: Connect here the **conv_info_out** of the side neighbour farthest to the vision sensor.

B.6.2 Outputs

- **stop_blade**: Command signal to the stop blade that is part of the Sorter.
- **arm1**: Command signal to turn the conveyor arm closest to the vision sensor.

B.7 Assembler

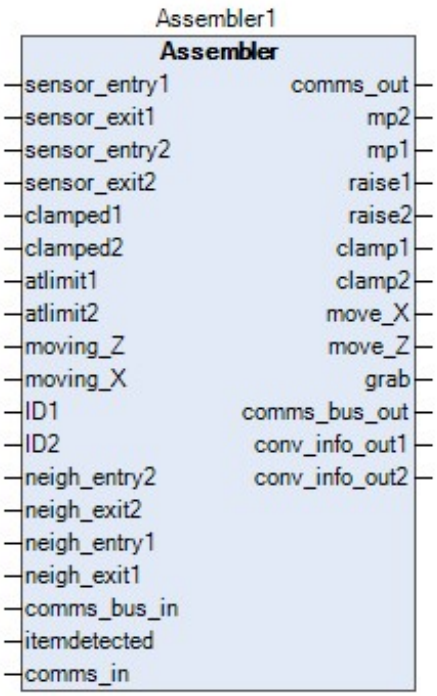

Figure B.7: Function Block **Assembler**

In this FB, every input/output name ending with a 1 represents the conveyor closest to the assembly robotic arm, while a 2 in the end of the name represents the conveyor farthest from the robotic arm (i.e. ID1 and ID2). All inputs/outputs have the same names as their FACTORY I/O counterparts, when applicable.

B.8 Turntable

B.8.1 Inputs

- **turn_limit**: Signal from FACTORY I/O indicating that the turntable has finished turning.
- **unturned_limit**: Signal from FACTORY I/O indicating that the turntable is not turned.

B.8.2 Outputs

- **turn**: Command the turntable to change position (from turned to not turned or vice versa).

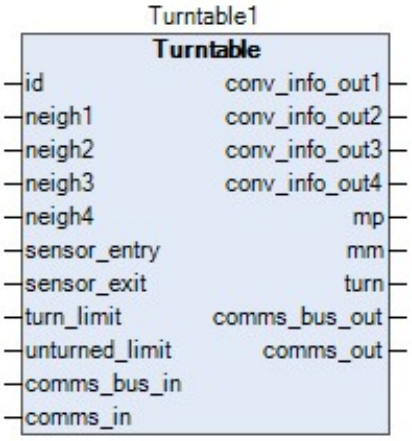

Figure B.8: Function Block **Turntable**

B.9 Warehouse

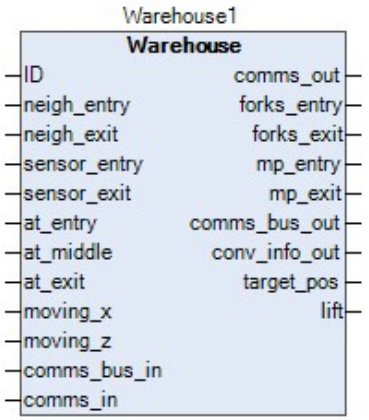

Figure B.9: Function Block **Warehouse**

B.9.1 Inputs

All inputs have the similar designations to their FACTORY I/O counterparts, or have been discussed already in other FBs.

B.9.2 Outputs

All outputs have similar designations to their FACTORY I/O counterparts, or have been discussed already in other FBs.

References

[1] R. Rosen, G. Von Wichert, G. Lo, and K. D. Bettenhausen, "About the importance of autonomy and digital twins for the future of manufacturing," *IFAC-PapersOnLine*, vol. 48, no. 3, pp. 567–572, 2015.

[2] M. Shafto, M. Conroy, R. Doyle, *et al.*, "Modeling, simulation, information technology & processing roadmap," *National Aeronautics and Space Administration*, vol. 32, no. 2012, pp. 1–38, 2012.

[3] E. J. Tuegel, A. R. Ingraffea, T. G. Eason, and S. M. Spottswood, "Reengineering aircraft structural life prediction using a digital twin," *International Journal of Aerospace Engineering*, vol. 2011, 2011.

[4] S. Haag and R. Anderl, "Digital twin – proof of concept," *Manufacturing Letters*, vol. 15, pp. 64–66, 2018, Industry 4.0 and Smart Manufacturing, ISSN: 2213-8463. DOI: https://doi.org/10.1016/j.mfglet.2018.02.006. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2213846318300208.

[5] R. Games. "About." (2006), [Online]. Available: https://docs.factoryio.com/ (visited on 01/02/2022).

[6] *Codesys*, Dec. 2021. [Online]. Available: https://en.wikipedia.org/wiki/CODESYS.

[7] M. Tiegelkamp and K.-H. John, *IEC 61131-3: Programming industrial automation systems*. Springer, 2010.

[8] D. H. Hanssen, *Programmable logic controllers: a practical approach to IEC 61131-3 using CODESYS*. John Wiley & Sons, 2015.

[9] I. E. Commission *et al.*, "Grafcet specification language for sequential function charts," *International Standard, IEC*, vol. 60848, p. 94, 2002.

[10] L. A. Bryan and E. A. Bryan, *Programmable controllers: theory and implementation*. Industrial Text Company, 1997.

[11] G. Barteling and R. Sadeghi, *Plccoder.com*, Dec. 2021. [Online]. Available: https://www.plccoder.com/.

[12] *Unified architecture*, Sep. 2019. [Online]. Available: https://opcfoundation.org/about/opc-technologies/opc-ua/.

[13] W. Mahnke, S.-H. Leitner, and M. Damm, *OPC unified architecture*. Springer Science & Business Media, 2009.

[14] S.-H. Leitner and W. Mahnke, "Opc ua–service-oriented architecture for industrial applications," *ABB Corporate Research Center*, vol. 48, no. 61-66, p. 22, 2006.

[15] T. Hannelius, M. Salmenpera, and S. Kuikka, "Roadmap to adopting opc ua," in *2008 6th IEEE International Conference on Industrial Informatics*, IEEE, 2008, pp. 756–761.

[16] M. Sousa, "Opc ua," *Industrial Informatics*, 2021.

[17] *Dijkstra's algorithm*, Jun. 2022. [Online]. Available: https://en.wikipedia.org/wiki/Dijkstra's_algorithm.