Faculdade de Engenharia da Universidade do Porto

Automação de Linha de Fabrico Flexível do DEEC

Daniel André da Silva Petim Batista

VERSÃO PROVISÓRIA

Dissertação realizada no âmbito do
Mestrado Integrado em Engenharia Electrotécnica e de Computadores
Major Automação

Orientador: Professor Doutor Mário Jorge Rodrigues de Sousa

Junho de 2001

© Daniel André da Silva Petim Batista, 2011

Resumo

A automação industrial é actualmente uma área de grande importância de engenharia sendo aplicada nas mais variadas industrias de produção industrial. Alguns dos inúmeros exemplos são a industria automobilística, industria química e industria alimentar.

Neste campo tem-se vindo a presenciar grandes esforços na normalização, sendo a Comissão Electrotécnica Internacional a entidade que lidera o processo de produção e publicação de normas neste domínio. O IEC 61131 é uma das normas publicadas por esta associação e estabelece um conjunto de características eléctricas mecânicas e lógicas que os Autómatos Programáveis (Programmable Logic Controllers) devem seguir. A componente 3 da norma estabelece um modelo de programação que define três unidades de organização de programas e cinco linguagens de programação. Os fabricantes dos PLCs têm vindo a adaptar as suas ferramentas de programação a esta norma, no entanto apresentam algumas inconsistências e formas de impossibilitar a portabilidade do desenvolvimento nessas ferramentas.

Devido a esses factores a empresa LOLITECH decidiu criar um ambiente de desenvolvimento integrado de código fonte aberto para PLCs, permitindo aos utilizadores escreverem programas em conformidade com a norma IEC 61131-3, e gerar código ANSI-C correspondente, através de um compilador intern, possibilitando a sua execução nas mais variadas plataformas.

Este trabalho apresenta o desenvolvimento de um algoritmo de controlo implementado na ferramenta Beremiz para uma linha de produção flexível que se encontra instalada no Departamento de Engenharia Electrotécnica e Computação. Pretendesse por uma lado validar a ferramenta Beremiz numa aplicação de controlo discreta, e por outro mostrar os aspectos de modelização e concepção de soluções na área da automação industrial que recorrem à norma IEC 61131-3.

Abstract

Automation fills the needs of almost any production based industry that aims to maximize productivity. Car making, chemistry and food industry are some examples were being competitive is only possible with tools such as automation.

There has being an effort from the International Electrotechnical Commission (IEC) to create a standard in this field to provide a workable platform to any entity working in automation. IEC 61131 is one of the standards used to establish a set of electrical, mechanical and logic rules that Programmable Logic Controllers (PLC) should follow. Due to some inconsistencies in the programming model (IEC 61131-3) of the standard, companies like LOLITECH are trying to overcome those problems creating an open source package, which can be used by any user to program PLC's using ANSI-C code that can be executed in different platforms, and, at the same time, complies with IEC 61131-3.

This work aims to create a control algorithm made with Beremiz toolbox and test it on a flexible prodution plant located at the Department of Electrical and Computer Engineering.

Part of this work lies in the validation of the Beremiz toolbox. The other part tries to show different aspects regarding conception and modeling of applications in the industrial automation field that follow IEC 61131-3.

Agradecimentos

Aos meios pais por todo o apoio ao longo destes anos, e também à minha família.
Aos meus amigos que de alguma forma tentaram ajudar.
Ao Professor Mário de Sousa pela disponibilidade ao longo de todo o projecto.

Contents

List of Figures

List of Tables

Symbols and Acronyms

ADU	Application Data Unit
FBD	Function Block Diagram
FB	Function Block
GUI	Graphical User Interface
IEC	International Electrotechnical Commission
I/O	Input/Output
LD	Ladder Diagram
PLC	Programming Logic Controller
ST	Structured Text
POU	Program Organization Unit
PDU	Protocol Data Unit
IDE	Integrated Development Environment
SCADA	Supervisory Control and Data Acquisition
HMI	User Machine Interface

Chapter 1

Introduction

1.1 - Motivation

"Society in its daily endeavors has become so dependent on automation that it is difficult to imagine life without automation engineering. In addition to the industrial production which it is popularly associated with, nowadays it covers a wide number of areas. Trade, environmental protection engineering, traffic engineering, agriculture, building engineering, and medical engineering are but some of the areas where automation is playing a prominent role" [1].

The department of electrical and computer engineering (DEEC) at the Faculty of Engineering of the University of Porto (FEUP) has recently acquired, for one of their laboratories, a discrete flexible manufacturing line (see Figure 1.1). This acquisition intends to provide the students better means to learn the technologies of industrial automation.

This line may be divided into five main modules:

- An automated warehouse to store work-pieces
- Two plates for work-pieces machining (serial and parallel), each one with two drilling machines
- An assembly plate composed by a 3 axis-portal robot in which it is possible to pile work-pieces
- A plate that allows the load of work-pieces from outside into the factory, and the opposite (unload of work-pieces from the factory to outside)

All these modules are connected by conveyors, which task is to route work-pieces to the different modules.

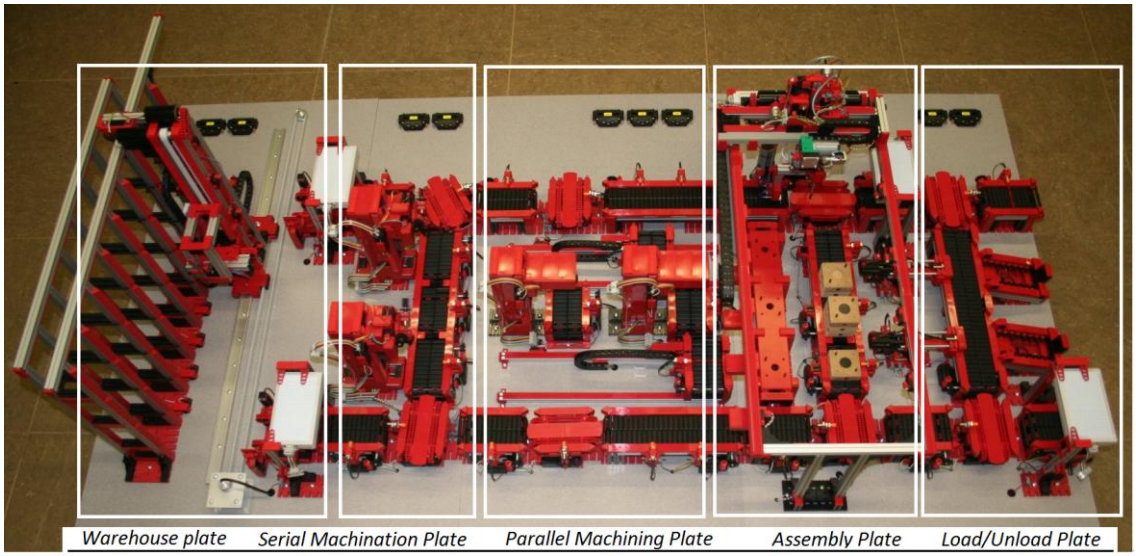

Figure 1.1 - Discrete manufacturing flexible line divided in five modules

The aim of this project is the development of a control application for the entire flexible manufacturing line described above using for that purpose the standard IEC-61131-3. The IEC 61131 standard can be briefly described as a general framework that tries to establish the rules to which all PLCs should follow, encompassing mechanical, electrical, and logical aspects. The third component, IEC 61131-3, deals with the programming aspects of the industrial controllers, defining the logical programming blocks and the programming languages [2].

Although most of the vendors adhere to this standard, they continue to lock the users into their product lines, and the code portability is still a problem between different vendors. Due to these reasons it was decided to use a free and open source IDE (integrated development environment) named Beremiz. This framework is strictly accordant with IEC-61131-3 standard and is a cross-platform software. Therefore except for programming the control algorithm, it is also expected to test and validate Beremiz as an automation framework using the flexible line.

This work intends to be later used for demonstration sessions of the flexible manufacturing line.

1.2 - Objectives

The outlined objectives can be divided into five groups in the following sequence:

1. Study of all modules and components of the flexible line
2. Detailed problem specification, i.e., define all the services that the user can request as well as all the different interactions that the production line has to perform after a request
3. Architecture problem modeling using an abstract layer tool
4. Programming the control algorithm according to standard IEC 61131-3

5. Test and validation of Beremiz as a automation framework
6. Development of a graphical user interface

1.3 - Document Structure

This document is divided into six chapters.

Chapter 1 presents the motivation behind this work and the objectives that have been defined.

Chapter 2 explains the main modules and components of the flexible manufacturing line.

Chapter 3 describes the technologies used in this project, starting with the communication network protocol used to connect the manufacturing line and the control application, followed by the IEC 61131-3 standard, where the main components that it describes will be presented. The last subchapters relates to Beremiz as IDE, describing briefly both "internal" (how it is implemented) and "external" (how it is present to the user) architecture.

Chapter 4 explains the concept of the problem and the services provided to the user, followed by the problem modeling architecture using UML as the abstract layer tool. Afterwards the implemented details on the control application according to IEC 61131-3 standard are demonstrated and matched with the UML architecture. Next follows an evaluation about Beremiz, concerning the main emerged problems, and the IDE evolution in the last months. The final subchapter presents the approach to the graphical user interface.

Chapter 5 relates the performed tests concerning the problem validation and overviews all of the work that has been done as well as future work.

Chapter 2

Flexible Line Description

2.1 - Overview

The flexible line considered in this project is a STAUDINGER physical toy model, which consists of five main modules as stated in Chapter 1 and the following components:

- parts to be processed (work-pieces)
- input and output stations
- material handling devices and transporters for transferring parts in and out of the robotized line
- machines to perform processing (drilling machines)
- one control device, (in this project a softPLC) to perform the control activities

The entire flexible line has only discrete sensors and actuators. Just to give an idea of the complexity which is normally associated with the number of hardware components, in total there are approximately 100 sensors (input signals) and 130 actuators (output signals).

This chapter starts with a brief description of the modules installed in the flexible line support, and the following subchapter deals with the disposition of the five main modules and the components that each module integrates.

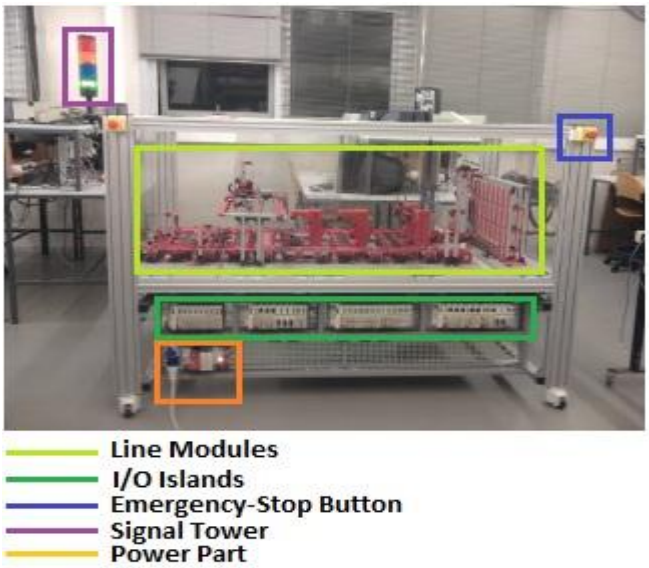

Figure 2.1 - Modules on the support of the manufacturing flexible line

Figure 2.1 shows the support for the modules installed in the flexible line. One of them concerns five switches (one for each separate module) installed on the left side at the bottom of the line support (not visible in Figure 2.1), to which it is possible to commute between two states, *remote mode* or *local mode*. In remote mode the line is controlled via network using the Modbus TCP or other fieldbus protocols. Hence, there's a group of islands installed under the support that are nothing more than distributed I/O modules (dark green rectangle in figure 2.1), each one composed of an Ethernet interface, a power source, several input/output digital cards associated with all sensors and actuators of the entire flexible line, and two counter modules to which the appropriate reference will be done in Section *assembly plate* of this chapter. In local mode the control is performed using the PLC wires which are directly connected to the different components.

Figure 2.2 - One of the four Islands installed in the flexible line

Another module is the embedded system ICNova AVR32 AP7000 (Linux based) installed under the line support, which runs an interesting interlock system working as a mask between the flexible line and the application control. This system prohibits contradictory orders when somebody is connected to the manufacturing line in remote mode. The architecture may be

described as a network switch where the four islands are connected as well as ICNova that in turn will be able to acknowledge all sensors and actuators of the flexible line. Inside the hardware (ICNova) runs a logical application related to a Modbus/TCP slave (on which the sensors and actuators are mapped) and a program responsible to prohibit the mentioned dangerous orders. As a practical example, if somebody sends a command to a conveyor ordering it to move in two different directions at the same time or a command saying to a rotary conveyor to rotate beyond its limit the interlock stops it automatically.

To advise the user about the line status there is a signal tower (purple rectangle in Figure 2.1) installed with four colors. The green one indicates that the line is ready to be used, in fact when the power is set up (using the power module (orange square in Figure 2.1), the islands will be initialized. This takes about 2-3 minutes, after this amount of time, the green light turns and remains on. The blue and the orange colors advise the user in case one interlock emerges, in that case the orange goes immediately off when the interlock disappears, and the blue one remains on for a couple of seconds after the interlock goes off. Note that without this interlocking system, simulation would be a crucial task once an incorrect synchronization between two components could damage part of the flexible line irreversibly. There are also four emergency buttons in the four corners of the support line which allow the user to immediately stop all the components in the flexible line. In this case the red light of the signal tower turns on.

2.2 - Modules Description

For better comprehension of the subsequent descriptions of the modules the upper view - regarding the components disposition of the entire flexible manufacturing line is being illustrated below.

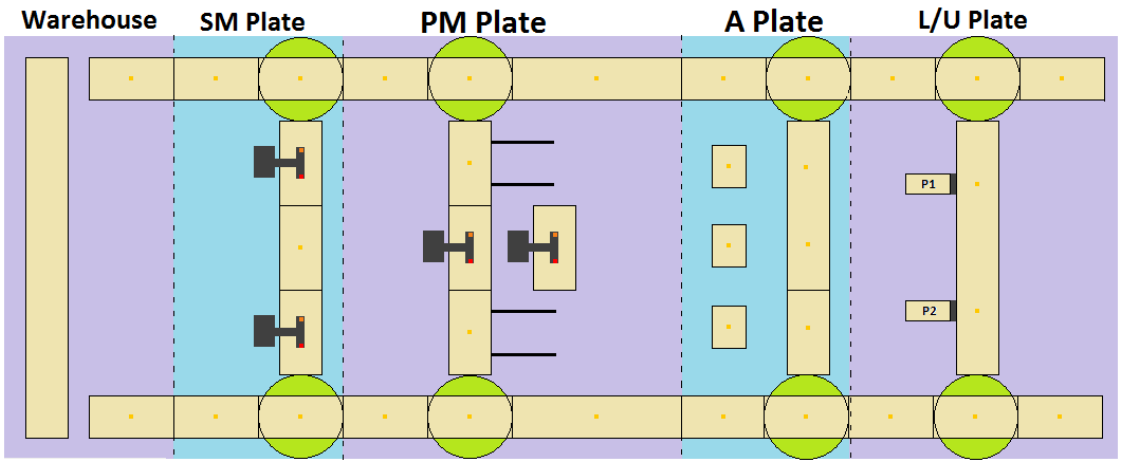

Figure 2.3 - Disposition of the components in the manufacturing flexible line

Figure 2.3 shows the modules referred earlier in chapter 1. The disposition from left to the right regarding the components that each module contains is as follows:

- **An automatic warehouse** composed of a stacker and two linear conveyors working as input/output stations
- **A serial machining plate** constituted of five linear conveyors (including the two conveyors attached to the drilling machines), two multi-spindle drilling machines, and two rotary conveyors
- **A parallel machining plate** composed of six linear conveyors, two rotary conveyors, two sliding conveyors and two vertical drilling machines
- **An assembly plate** constituted of four linear conveyors, two rotary conveyors, three work-tables, and a 3-axis Gripper which can move inside the entire plate (light blue zone in A Plate (Figure 2.3))
- **A load/unload plate** composed of five conveyors, two rotary conveyors and two pushers.

2.2.1. Warehouse

The warehouse is the place where the blocks can be stored. It is made of twenty four alveoli (storage place, each one stores one work-piece), distributed into three columns and eight rows. These alveoli are not provided with any sensors to recognize if a work-piece is present within them; therefore, it's the control algorithm that has the responsibility to recognize which blocks are distributed in the warehouse.

Figure 2.4 - Warehouse of the flexible line

In order to store and remove work-pieces from the warehouse, there is a stacker. This component can move in the three Cartesian axes. There are pressure sensors along the Cartesian axis where the stacker can move, allowing the stacker to reach a specific position/alveoli. The movement is performed at constant speed allowing the control to use discrete actuators to move the stacker. One interesting particularity is the discrete signal which commands high (if the command is true) or low (if the command is false) speed in X-

axis. In fact if high speed is always activated, the stacker may not stop exactly in the specific sensor along the x-axis, generating an interlock. Therefore, the correct way to control it is to move the stacker in high speed until it reaches the x±1 desired position and then switch to low speed until it reaches the required position (x position). Only the x-axis has this particularity (two speeds), the Z and Y axis movement have only two commands to move the stacker in each direction.

Another detail is the Z-axis sensor number. In fact, there are only three positions in the Z axis (three rows) to store work-pieces, but there are six sensors distributed along the z-axis. The reason for this detail is that, in order to store a block piece into the warehouse, the stacker needs to first rise to an upper position of the alveoli, and then step down until it reaches the next sensor below, engaging this way a work-piece in one alveoli. The same action but with inverse logic has to be done in order to remove a piece from the warehouse.

The blocks origin/destination which are going to be store/removed from the warehouse are two linear conveyors working as input/output stations. These conveyors are quite different from the others respecting the sensor type which is an optical sensor, and the mechanical structure which has an open slot where the stacker may engage, leaving the work-piece up in the conveyor.

2.2.2 - Serial and Parallel Machining Plates

The serial and parallel machining plates are composed of several linear conveyors, and a couple of rotary conveyors. The linear conveyors can move in two different directions in the same orientation, for this purpose, there are two discrete actuators (one for each direction). The movement speed is always constant and equal in the two different directions. For each conveyor there is one sensor located in the center of the respective conveyor. Note that due to this reason a conveyor should only transport one work-piece at a time. The rotary conveyor has the same behavior as the linear one, but it´s also possible to rotate it. To control the rotation, the conveyor has two more actuators, each for one rotation direction control (and again, the speed is the same in the two different orientations, existing for this reason just two discrete actuators, one for each orientation). The maximum rotation that a rotary conveyor can reach is ninety degrees, existing in the two extremes, two end limit position sensors. Apart from linear and rotary conveyors the parallel plate integrates two sliding conveyors. These are almost identical to the rotary ones (in terms of sensors and actuators), the only difference is the rotation which is executed as a linear translation.

These plates also integrate four drilling machines. In the serial machining plate there are two multi-spindle drilling machines that can perform machining operation on the work-piece, which have to be upon the simple conveyor attached to the machine. This conveyor is considered an integrated part of the milling machine and it is controlled in the same way as the other linear conveyors, having so the same commands and sensors. In this type of machine there are three integrated tools in a turret. The tool change is performed ordering the rotation of the turret, always in the same direction and speed until the desired tool is in machining position. Afterwards the machining itself can be performed on the work-piece using a specific digital command (start machining). A sensor is activated if a tool (one of the existing three) is in the machining position. Since this sensor doesn’t indicate which tool is in

the machining position, but only the presence of one tool, it is the responsibility of the control algorithm to keep in memory the number of rotations executed. Four additional commands are provided to move the machine forwards/backwards and downwards/upwards in order to reach the work-piece, and four respective digital end limit position sensors to inform the activation limits of those commands.

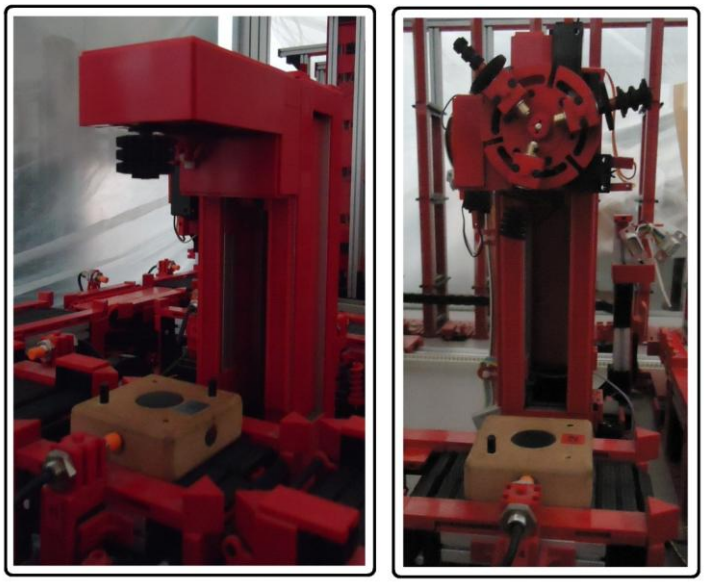

Figure 2.5 - Horizontal and multi-spindle drilling machines of the flexible line

The two additional milling machines (designated as horizontal milling machines) are situated in the parallel machining plate. These are significantly less complex than the multi-spindle milling machines, as they do not have a tool turret, just one fixed tool ready to operate. This machine does not operate forwards and backwards, but only contains actuators to move it downwards and upwards and the two respective discrete end limit sensors.

Note also the disposition in which these four machines are disposed. In the serial machining case if a work-piece enters in one side, it has to pass through both machines. In the parallel machining plate case, as there are two sliding conveyors, there is the possibility to choose between the two respective machines.

2.2.3 - Assembly Plate

In the assembly plate, it is possible to pile work-pieces forming this way a composed work-piece. The main component to perform such a task is a tri-axial gantry type robot. A composed block piece can have at most three "simple" work-pieces pilled in a certain order, and the gripper is able to grab and move this composed work-piece inside the robot operation zone, but just one (composed or simple work-piece) at a time. The robot structure contains pressure sensors distributed along the X, Y and Z axis aligned with the other components that are inside the robot operation zone. These components are conveyors and three assembly-tables. The conveyors work in the same way as the ones described in Serial and Parallel Machining Plate. The assembly-tables are no more than horizontal supports (having just one binary sensor to inform if a work-piece is present) on which will be possible to place and to

pile the simple work-pieces and afterwards transfer the composed work-piece to the output conveyor.

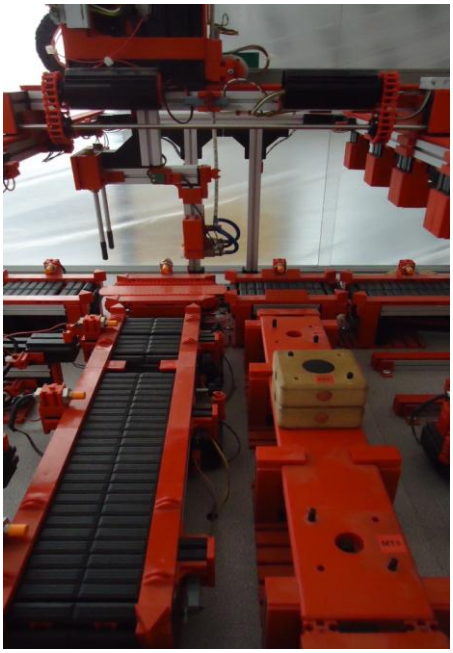

Figure 2.6 - Assembly plate module illustrating the robot gripper and the assembly-tables

There are six discrete actuators to move the gripper at constant speed inside the operation zone along the x, y and z axis (two for each axes, to perform the movement in each direction), and one more to give the order to engage the gripper, so as to grab the work-pieces. Only two sensors in Z axis are however provided to pile the blocks but it´s also necessary to know intermediate positions (not only up and down positions), since when the second or the third work-piece is going to be piled it´s necessary to release the gripper before it reaches the down position. Therefore, it is needed to use encoders to recognize intermediate positions. For this purpose, two channels of a counter module installed in the specific island -described in the overview chapter will be used.

It is possible to execute complex trajectories with the robot gripper while moving it on the X and Y axis at the same time. The composed work-pieces might be stored in the warehouse, as well as machined in the two machining plates.

2.2.4 - Load/Unload Plate

In this plate it is possible to unload work-pieces from the factory to the outside, as well as load work-pieces from the outside into the factory. The unloading of work-pieces is performed using two pushers, each one has a container with space for two simple or composed work-pieces. In order to load work-pieces inside the robotized line, two linear conveyors can be used to perform such a task, one located in the upper corner on the right, the other in the lower corner on the right (see Figure 2.3).

Attached to the two pushers there is only one linear conveyor. This is controlled in the same way as those described in *Serial and Parallel Plate*, but instead of one sensor in the middle of the conveyor, there are two (the length of this conveyor is quite bigger than the

others (see Figure 2.7)), one in each pusher, so that the programmer knows if a work-piece is aligned in front of each pusher.

Figure 2.7 - Load/unload plate showing the pushers and respective containers

To perform the extension and the retraction of one pusher there are two discrete command signals, so the position limits are signalized by two end limit position sensors (on both extremes of the pusher). To know if the pusher container is full, there is one optic sensor in each pusher container.

Chapter 3

Technologies

This chapter intends to explore the technologies used during this project. The first subchapter deals with the Modbus network protocol, with special incidence in the regularly designated Modbus/TCP implementation which was the communication method used between the control algorithm and the manufacturing line. Next, follows the analysis of the third component of the standard 61131 which was the "programming model" used to program the control algorithm. At the end, framework Beremiz is discussed as an IDE for PLC´s (Programmable Logic Controllers) approaching both internal (how it is implemented), and external (how it is presented to the user) structural design.

3.1 - Modbus

3.1.1. Overview

MODBUS is an application-layer protocol based on a client/server or request/reply architecture. It was published by Modicon in 1979 and is primarily used in industrial applications due to its implementation simplicity and robustness. The Modbus Messaging protocol is only a protocol and does not imply any specific hardware implementation.

The requests and responses are based in simple frames, designated as *Protocol Data Units* (PDUs), independent of the underlying communication layers. The specification defines three types of PDU´s:

- **Modbus requests** - the messages sent to the network by the clients in order to initiate transactions. These messages serve as indications of the requested services (function code and data) on the server´s side.

- **Modbus responses** - the response messages sent by the servers. These messages serve as confirmations (corresponding function code and data) on the client´s side.
- **Modbus Exception Response PDU** - the server returns a code that is equivalent to the original function code from the request PDU with its most significant bit set to logic 1 (original code + 128).

"The interaction between client and server (controller and target device) can be depicted as follows. The parameters exchanged by the client and the server consist of the Function Code (what to do), the Data Request (with which input or output) and the Data response (result) [1]". Figure 3.1 shows that behavior.

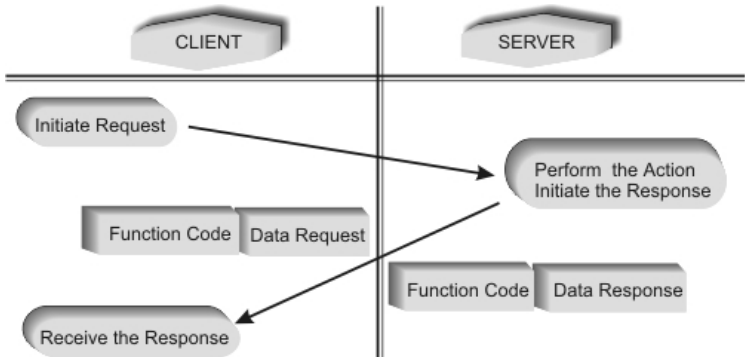

Figure 3.1 - Actions in a Modbus transaction without errors (Source: [1])

3.1.2. Services

The protocol specification defines a set of functions, where each one has a unique code. These codes are in range [1-127], being that the range [129-255] is reserved for exception response codes.

The specification also defines three categories of functions codes as follows:

- **Public** - This category includes well-defined public function codes, defined uniquely for all the users of the protocol. These functions are documented and verified by Modbus organization [3].
- **User defined** - The user-defined codes, assure the flexibility of the protocol that accepts that the producer can add new functions without any approval. These codes (ranges [66-72] and [100-110]) do not provide the guaranty of their uniqueness. In this case the producers have to publish their specification.
- **Reserved** - These functions are used by companies for legacy products and they do not represent public function's interest.

The two PDUs formats (request and response) are documented with its purpose, parameters and return values, being that sometimes the response can be an exception as

mentioned before (Modbus exception response). The details about the error are identified in a proper field, and it depends on the invocated function. In the specification [4] it´s documented the *Public* category functions where the related information of public requests, responses and associated errors can be found.

3.1.3. Data Model

Table 1 presents the basic data types in Modbus protocol referred by the document [4].

Table 3.0.1 - Basic used data types in Modbus protocol (source: [4])

Name	Type	Access
Discretes Input	1 bit	Read-Only
Coils or Discretes Output	1 bit	Read-Write
Input Registers	16-bit word	Read-Only
Holding Registers or Output Registers	16-bit word	Read-Write

There are distinctive differences between inputs and outputs, and between bit-addressable and word addressable data items. However these differences do not suggest that the application behaves in a particular way. So that considering all four tables as overlaying one another is very common, since this is often the most natural interpretation on the under consideration target machine.

For each one of the primary tables, the protocol allows individual selection of 65536 data items, and the operations of read or write of those items are designed to span multiple consecutive data items up to a data size limit which is dependent on the transaction function code .

The device application memory must be the location of all handled Modbus data (bits, registers) but the physical address in memory and data reference should be differentiated with each other despite the required connection between them [4].

3.1.4. Implementation TCP/IP

Modbus protocol has a variety of implementations and the most popular work on TCP/IP and asynchronous serial transmission, in this last case the most common physical fields are EIA/TIA-232 and EIA/TIA-485. Document [5] provides the implementation specification for TCP/IP as well as the functional descriptions for a client, a server and a gateway (device that guarantees the interface between an IP network and a serial bus).

The MODBUS protocol defines a frame (PDU) independent of the underlying communication layers as seen in Section 3.1.1. The mapping of Modbus protocol on specific buses or networks can introduce some additional fields on the Application Data Unit (ADU) which integrates the PDU. In case of TCP/IP implementation the specification [5] adds a dedicated header on the PDU designated as MBAP (MODBUS Application Protocol header). This header has 7 bytes length and is composed of the following four fields:

1. ***transaction identifier*** (2 bytes)- Identification of a Modbus request/response transaction;

2. ***protocol identifier*** (2 bytes)- It has the value 0 as the default number for Modbus (exists for the expectation of future expansions);
3. ***length*** (2 bytes)- Number of following bytes in a frame. Help to detect the limits in a frame;
4. ***unit identifier*** (1 bytes)- Identification of a remote slave connected on a serial line or on different buses;

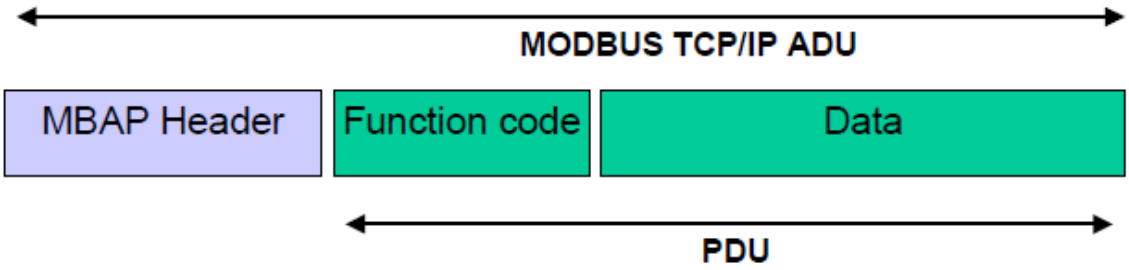

Figure 3.2 - Modbus/TCP Application Data Unit (Source: [5])

The topologies and configurations of a Modbus TCP/IP network are not defined in the specification, they just present two illustrative examples. It's perfectly feasible to build network topologies with more than one client, or even have devices which work simultaneously as client and server. This particularity is regarded as a big advantage when compared to other implementations.

3.2 - Standard 61131-3

3.2.1. Overview

"The IEC 61131 standard has its origins in 1979, when an international committee was formed with the intention of generating a standard for a common user interface for PLCs. In 1982, this committee came up with a draft that was so complex that it was then decided to divide it into five parts [6]." These parts concern both PLC hardware and programming system, being that the third part (IEC 61131-3) deals with the programming aspect of the industrial controllers and defines the programming model, composed of three program organization units and five programming languages.

Although the standard defines a set of rules, to which all the PLCs manufacturers should adhere to, this set is not being regarded as a set of rigid rules. In fact vendors can implement as much as they want from the enormous number of details defined in the standard, being that these aspects after being implemented have to be documented, proving this way the parts that they do or do not fulfill in the standard [7].

However the adherence of the standard has had a big acceptance in the last few years by manufacturers, bringing advantages for both manufacturers and customers. As related in [7] some of the main advantages of using the standard are:

- Gradually, the industrial equipment manufacturers (not just PLCs) are adopting it.
- Standardization of the equipment functional structure

- Programming languages standardization, i.e. unique software model, independent of the manufacturer, standard functions and functions blocks, and reuse of already developed software
- Allows development of structured code (higher quality and fewer errors).
- Existence of typed data, minimizing errors
- Support for complex data structures
- Provides the most suitable language for each type of problem, such as high and low programming languages, and textual and graphical languages

There is been a continuously increasing tendency of leaning towards the software market of the PC world, which PLC programming could not avoid being a part of, and thus they have gradually joined the software market trend. Standardization and synergy are basic and very significant factors in the process of reducing costs. Both manufacturers and customers benefit from IEC 61131-3 for it brings previously manufacturer specific systems closer together [7].

3.2.2. Building Blocks

Building blocks, referred to as POUs in the standard, fall in one of three types: Function (FUN), Function block (FB) and Program (PROG), in ascending order of functionality. As the name implies, they can be seen as the smallest independent software units of a user program.

The typical structure of POU follows in Figure 3.3:

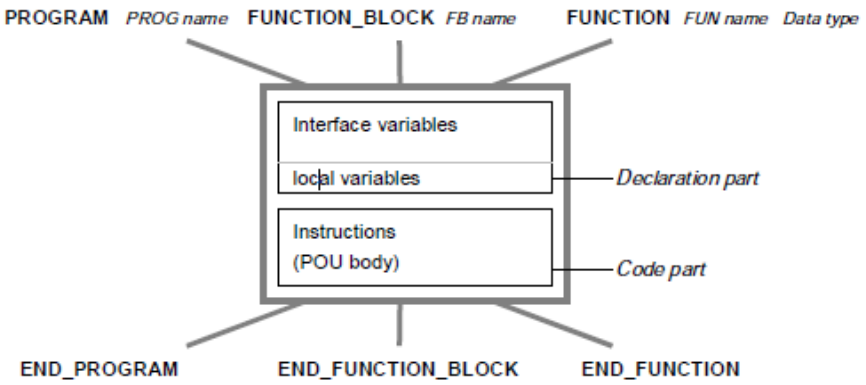

Figure 3.3 - The common structure of the three POU types (Source: [7])

As can be seen in the Figure 3.3, a POU consists of a variable declaration part and a code part. Variables may have local scope or be declared as input and/or output. Variables with local scope can only be accessed by the own FB, while variables declared as input or output are used to interface other POUs. The code part is specified using one of the languages defined in the standard and contains the instructions to be processed by the PLC.

The code part may contain calls to other building blocks, having three possibilities of invocation among the three POU types as follows:

- Program may call function or function block
- Function block may call function or other function block (recursion is not allowed in any of the three types of POUs)
- Functions may call other functions

Functions always produce the same result (function value) when called with the same input parameters, i.e. they have no "memory" and they can have one or more inputs (parameters), which may include output variables from other FBs, or even a result from another function. However functions have only a single direct output value (function result). When calling a function the parameters can or cannot be filled, being that, unspecified parameters are automatically set to their default values. Functions can only be programmed using four of the five programming languages, precluding SFC [6].

Unlike functions, function blocks have their own data record and can therefore remember status information (instantiation) beyond they may use all the five languages defined in the standard. After being created, each instance of a FB is independent among the several instances that were created for a same type of a FB. If a function block is called for the first time with unspecified parameters it behaves as a function (the parameter is set to its default value), otherwise the values of the previous call are retained. Therefore the function block parameters may be said to be persistent between calls. For FBs there are two types of local variables, designated as persistent or temporary, these as mentioned before can only be accessed within the FB itself. The temporary variables are initialized to their default values every time the function block is called, unlike persistent variables whose values remain unchanged between calls [6].

Programs (PROG) represents the "top" of a PLC user program, and this kind of POU is very similar to FBs, with the exception that programs can only be instantiated inside a *Configuration*, unlike FBs that can be instantiated inside programs and other FBs. *Configurations* will be better described in section 3.2.4 [6].

The IEC 61131-3 provides some standard functions and function blocks, commonly referred to as *basic building blocks*. These blocks are in a range of various types such as conversion type functions, numerical functions, arithmetical functions among many others. In reference [7] can be found all the functions and FBs that the standard establishes. Some of the standard functions can be used with different data types, but this particularity (overloading of data types) is only allowed for standard functions.

3.2.3. Data Types and Variables

3.2.3.1 - Data Types

The standard besides organization units also defines a set of predefined data types commonly designated in standard as *elementary data types*. Table 2 presents these data types that are along with their size and the respective default number that is assigned when the parameters in a POU are not specified (remember previous Section).

Table 3.2 - The elementary data types of IEC 61131-3 standard (Source: [6])

Data Type	Size (bits)	Default Number
BOOL	1	FALSE
BYTE	8	0
WORD	16	0
DWORD	32	0
LWORD	64	0
SINT/USINT	8	0
INT/UINT	16	0
DINT/UDINT	32	0
LINT/ULINT	64	0
REAL	32	0
LREAL	64	0
STRING	8 (per character)	' ' (empty string)
WSTRING	16 (per character)	" " (empty string)
TIME	(implementation dependent)	T#0S
TIME_OF_DAY	(implementation dependent)	TOD#00:00:00
DATE	(implementation dependent)	D#0001-01-01
DATE_AND_TIME	(implementation dependent)	DT#0001-01-01-00:00:00

Note in Table 2 how the standard supports data types to handle time and the passing of time. TIME_OF_DAY data type is used for absolute times in the day, the TIME data type is used for relative times, such as periods, offsets and the difference between two TIME_OF_DAYs [6].

Besides elementary data type, the standard provides the opportunity to create new data types and as these derive from the elementary data types they are referred to as *derived data types*. According to [7] these variables fall in one of the following types:

- **directly** - The most simple derivate data type. Creates an elementary data type with a particular initial value,

- **subrange** - Creates an elementary data type to which values within a specific range can be assumed. Their default value is the lowest value in the subrange, unless specified otherwise,

- **enumeration** - The variable can assume one value out of a specific list of names, using the first value as the default number. The names of the values in an enumeration may not be reused for other constructs such as variable names, so that there is no ambiguity,

- **array** - Several elements of the same data type are combined into an array. While accessing the array, the maximal permissible subscript (index) must not be exceeded. Arrays may be defined out of any data type, which excludes arrays of a function block type, as function blocks are not considered a data type. Multiple arrays of a particular data type form a multidimensional array type.

- **structure** - Several data types are combined to form one data type. Structures can also be applied to derived data types, i.e. they can be nested;

3.2.3.2 - Variables

Variables are declared together with a data type as placeholders for application specific data areas, as illustrated in Figure 3.4. Their declaration properties according to [7] are composed by means of:

- Properties of the specific (elementary or derived) data type,
- Information about additional initial values,
- Information about additional array limits (array definition),
- Variable type of the declaration block in which the variable is declared to (with attribute/qualifier).

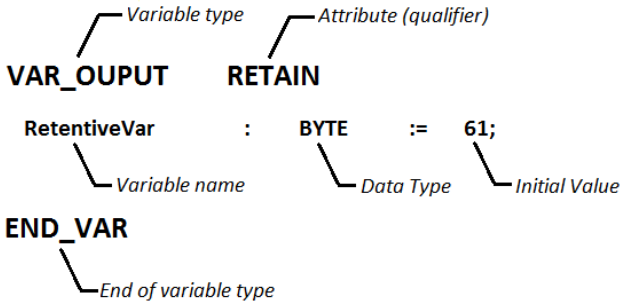

Figure 3.4 - Elements of a variable declaration with initial value assignment (Source: [7])

The variable declaration in Figure 3.4 has a byte data type, 61 as initial value, and a retain qualifier. This last particularity allows the battery-backed to store and keep the variable data value in case of a power-off/power-on cycle, otherwise (non-retainable case) the variable is reset to its default value after a PLC reset.

Note that the declaration of an instance name for functions blocks represents a special case of variable declaration. In fact a FB instance name is declared just like a variable, except that the FB name is specified in place of the data type.

Relating to inputs, outputs and flags when they are associated to PLC system´s processors and their I/O modules in the program, IEC 61131-3 gives special treatment to the IEC variable concept and it offers two possibilities to access it through the programmer. The one refers to *directly represented variables* and the other one to *symbolic variables*. In case of *directly represented variables*, a data type is assigned to a hierarchical address and in the *symbolic represented variables* case they can also be accessed "symbolically", this is, with a variable name [7].

The symbology to declare such variables is specified using the keyword AT. The address structure is done concatenating:

'%' + location + length + one or more integers separated by '.'

These direct PLC addresses are also called *hierarchical addresses*. The prefixes for location and length can be consulted in Table 3.

Table 3.3 - Prefixes for the location and length of *directly represented variables* and *symbolic variables* (Source: [8])

Type	Prefix	Meaning
Location	I	Input
	Q	Output
	M	Memory/Flag
Length	X or none	1 bit
	B	8 bits
	W	16 bits
	D	32 bits
	L	64 bits

As referred in the beginning of this subchapter, the third part of the standard 61131 defines five programming languages. Of these, three are graphical based languages, and two are text based languages as follows:

- **ST** - *Structured Text* (text based language);
- **IL** - *Instruction List* (text based language);
- **LD** - *Ladder Diagram* (graphical based language);
- **FBD** - *Function Block Diagram* (graphical based language);
- **SFC** - *Sequential Function Chart* (graphical based language);

All the five languages support the same data types, i.e a specific building block written in one of the five languages may be called in another different building block which is written in a different language. This final building block does not have to concern about data type converting between the two languages. Although the same syntax for two different languages may be defined almost in all situations, the languages are not completely interchangeable between them. For this reason there are languages that fit better than others in certain specific tasks [6].

The language Ladder Diagram (LD) comes from the field of electromechanical relay systems and describes the power flow through the network of a POU from left to right. According to [6], this language can be considered a historical artifact, since the first PLCs were competing with existing control equipment based on hardwired relay circuits, and therefore adopted a language similar to the electrical circuits in order to ease platform acceptance by the existing technicians.

This language can be seen as a set of connections between logical checkers (contacts) and actuators (coils) connected in serial or parallel which are connected between two vertical power rails. If a path can be traced between the left side of the rung and the output, through asserted (true or "closed") contacts, the rung is true and the output coil storage bit is asserted or true. If no path can be traced, then the output is false and the "coil" by analogy to electromechanical relays is considered "de-energized" (see Figure 3.5). Ladder logic has contacts that make or break circuits to control coils. Each coil or contact corresponds to the status of a single bit in the programmable controller's memory, although the standard has extended the language to allow the calling of other building blocks (functions or function block instances) and to handle referencing data in variables more powerful than the simple Boolean type, such as analog values, counters, and timers [6].

The language Function Block Diagram (FBD) comes originally from the field of signal processing, and it describes a function between input variables (on the right hand side) and output variables (on the left hand side). A function is described as a set of elementary blocks and the user must place boxes representing the building blocks (functions or function blocks) and then connect the inputs and outputs of these using connection lines

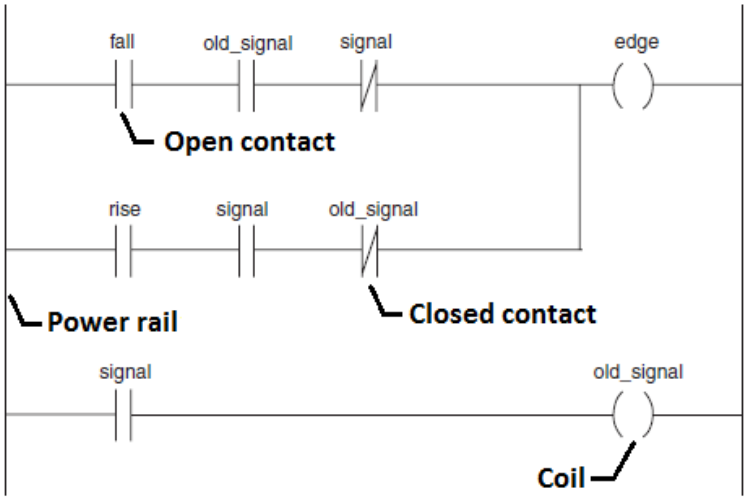

Figure 3.5 - Example of a LD program (Source: [6])

The input lines of a building block can only be connected to the outputs of other building blocks and the same for the outputs that can only be connected to inputs of other FBs. The data compatibility also has to be considered, i.e. it is not allowed to connect different data types. The standard specifies basic building blocks that implement the basic logic operations, counters, and timers, which make this programming language somewhat similar to designing a digital electrical circuit based on logic gates, counters, and timers [6].

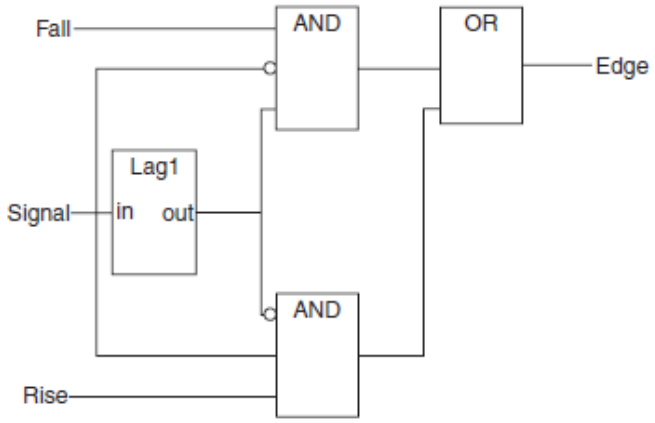

Figure 3.6 - Example of a FBD program (Source: [6])

The IL language is low-level programming language resembling assembly (see Figure 3.7). It is universally usable and often employed as a common intermediate language to which the other textual and graphical languages are translated in. Like LD, this language has also been

extended to handle the calling of functions and function blocks, and deal with complex data variables [6,7].

```
LD fall
AND old_signal
ANDN signal
OR (
LD rise
AND signal
ANDN old_signal
)
ST edge
LD signal
ST old_signal
```

Figure 3.7 - Example of an IL program (Source: [6])

ST is a procedural language consisting of a list of statements. Each statement is used to compute and assign values, to control the command flow and to call or leave a POU. ST is called a High-Level Language (opposed to Instruction List), because it does not use low-level, machine-oriented operands but offers a large range of abstractions statements describing complex functionality in a very compressed way [7]. According to [7] this brings advantages comparing to IL and this in turn brings its own disadvantages. Some of these advantages are:

- Very compressed formulation of the programming task
- Clear construction of the program in statements blocks
- Powerful constructs to control the command flow

The disadvantages are:

- The translation to machine code cannot be directly influenced by the user since it is performed automatically by means of a compiler.
- The high degree of abstraction can lead to a loss of efficiency (compiled programs are in general longer and slower).

As with other high level programming languages such as Pascal or C, ST supports interaction statements such as FOR, WHILE, and REPEAT as well as well known flow control statements such as IF THEN, ELSE and CASE. Both text-based languages (ST and IL) share the same syntax for declaring the interfaces to program building blocks and for the declaration of derived data types.

```
fedge := fall AND (old_signal AND NOT signal);
redge := rise AND (signal AND NOT old_signal);
edge := fedge OR redge;
old_signal := signal;
```

Figure 3.8 - ST example (Source: [6])

SFC (sequence function chart) is a *statechart machine* based language, inspired on Grafcet that in turn was originated in France in the 1970s and was later standardized by the IEC itself in IEC 60848. Due to this "inheritance", all the methodologies applied in Grafcet, have been integrated into the world of IEC 61131-3.

According to [7] SFC was defined to break down a complex program into smaller manageable units and to describe the flow control between these units. It describes operation sequences and interactions between parallel, sequential and concurrent processes. However it needs to use one of the remaining IEC 61131-3 languages to define the conditions referred as actions and transitions. These two conditions (actions and transitions) and steps (referred to as *states* in *statechart machines*) are the three fundamental concepts of this language. Between any two linked steps, there must be exactly one transition. Likewise, between any two linked transitions, exactly one step must be found. Transitions are only enabled when the immediately preceding steps (connected to the transition by directed links) are active. The evolution of a SFC (deactivation of the current steps and activation of all successor steps) depends on the firing of the transitions, and a transition can only fire when they are enabled and the condition associated with it is true. One or more actions may be associated with each step. Actions are only executed while the step is active. Qualifiers are associated to actions, and these define how the actions should be performed (set (P), non-stored (N), time limited (L)). Associated to each step there are two automatic variables, referred to as <*stepname.X*> and <*stepname.T*>. The first is related with the step activation (true if it´s active, false otherwise), the following gives the step activity time. These variables are often very useful in transitions definition [6].

3.2.4. PLC Configuration

In order to create a complete structure for the POUs described in section 3.3.2, the IEC 61131-3 standard defined the concept referred to as *configuration*. Each *configuration* is composed of abstraction layers defined as resources and tasks (see Figure 3.9). In fact, each resource can be seen as a CPU since there are complex control applications that may need more than just a CPU. This way, it is possible to define which POUs run in each CPU/resource. Another particularity that should be specified is the resource type which refers to a CPU model. Each vendor is expected to provide a library of resource types that it supports.

Inside the configurations and resources it is possible to define variables that are often designated as global variables. All the variables declared inside a configuration are seen in all POUs variables scope, whereas variables declared in one resource, are only seen by POUs that are inside this same resource. Nevertheless, from the point of view of the programs or function blocks, these variables are external to them, and must be declared in the program or function block type declarations as external variables before being accessed.

Each resource may be attributed one or more tasks. Tasks are similar to processes in common operating systems, but according to [6], the standard does not specify how these tasks should be implemented. The standard only specifies how tasks should behave.

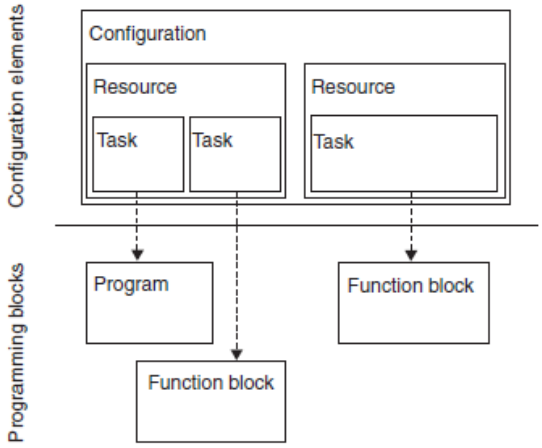

Figure 3.9 - The components of a configuration (Source: [6])

Tasks can be configured to run periodically, or run upon a rising edge of a specific variable. It is inside the tasks that FB and PROG POUs types are instantiated. The number of tasks a resource type supports, as well as the properties that may be applied to it (such as the period) depends on the implementation. Most simple CPUs support a single fixed task, while some may support a fixed number of two or three tasks. Currently, few CPUs permit the programmer/user to define large numbers of new tasks, which the standard allows [6].

3.3 - Beremiz

3.3.1. Overview

Beremiz is a free, cross-platform (it may run in different O.S.) and open source Integrated Development Environment (IDE) for developing of PLC programs in accordance with the third part of the standard IEC 61131, available for public use under the software license GNU GPL v2 or later.

This automation framework is the result of a long development effort, taking roots at LOLITECH in Saint-Dié-des-Vosges, France and at the University of Porto, Portugal. On the one hand LOLITECH, an enterprise created by the authors of the CANFestival project in 2005, decided to bet in Free and Open Source Software for automation. On the other hand, Professor Mario de Sousa, working for the "Faculdade de Engenharia da Universidade do Porto" developed the original IEC-61131-3 compiler, initially part of the MatPLC project. Therefore this project can be seen as combination of these two identities [9].

The main motivations to create a framework with such features assumed by the authors in [9] and [10] were the following:

- Identification of lack in open source solutions in this area;
- Despite normalization on the programming of PLCs, there are difficulties in porting the developed programs;

- The design and maintenance of applications developed for PLCs are directly dependent on the respective IDE associated with the hardware manufacturers;
- The learning process of the IEC 61131-3 standard typically involves the acquisition of expensive licenses, limiting or even precluding students of software;
- Operating safety may hardly be proven, as source code of PLC runtime and compilers are closed.

Beremiz was developed in a modular form, relying on the following sub-projects:

- **PLC Builder GUI** - Global vision of the projects
- **PLCOpen Editor** -Program editor according with IEC 61131-3 standard
- **MatIEC** -IEC 61131-3 -> ANSI-C compiler
- **CanFestival** - CANOpen framework for physical I/O interface
- **SVGUI** - Tool for development of HDMIs

The PLC Builder GUI and PLCOpen Editor are programmed in Python, using the module WxPython [11] for the library WxWidgets [12] (cross platform technologies). The two last referred sub-projects fall in the category that the authors refer as plugins.

3.3.2. PLC Builder GUI

The designated PLC Builder GUI gives a global perspective to the user, presenting three main parts (see Figure 3.10):

- Toolbar
- Plugins management area
- Log console - Textual information provided for the user

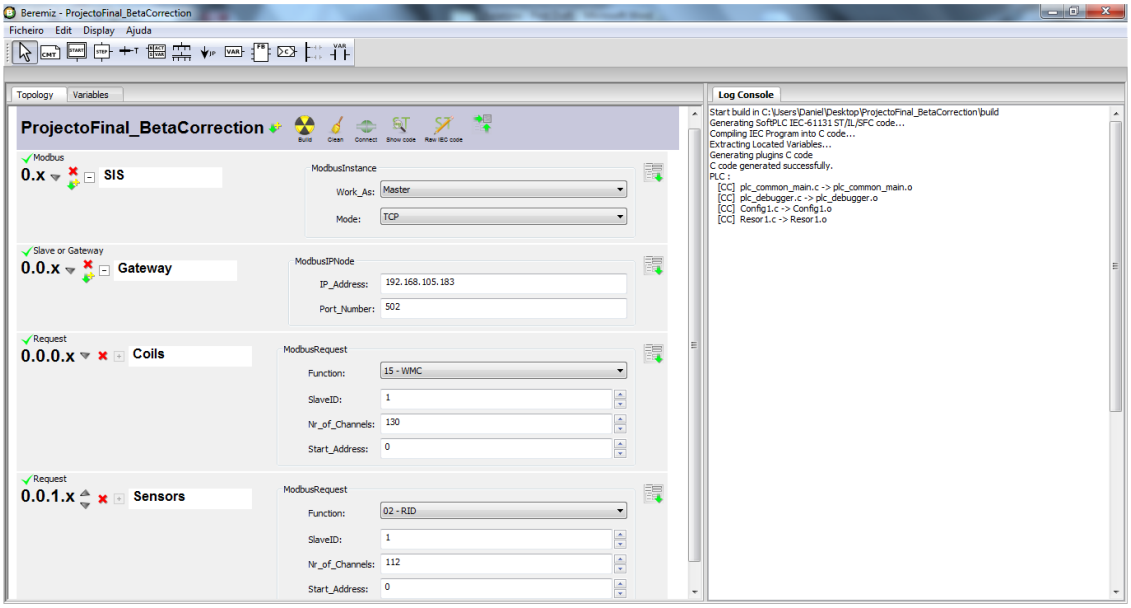

Figure 3.10 - PLC Builder GUI

The tool bar is dynamic, i.e. the user is able to edit at any time some of the project options. Some of the available options include defining the target platform adding, removing, or editing plugins, defining compilation options, compiling the project, viewing the IEC 61131-3 generated code (after compilation), transferring the program into the softPLC, and starting and stopping the execution of the control algorithm.

The plugins are added in a tree structure, so that it is possible to associate a specific plugin from another that is dependent of the first. Plugins present a similar tool bar, concerning in both association of new plugins and the way in which the parameters are presented to the user. According to [8] plugins are seen from the programming GUI point of view as classes that inherit from the same abstract class.

The log console presents information concerning the status results (debugging code of the IEC-61131-3 standard) when the user compiles a program. The error messages reflecting the user mistakes end up also in this space.

3.3.3. PLC Open Editor

The PLC Open Editor is the space where the user writes the programs in this framework. It is composed of several divisions as illustrated in figure 3.11.

In the vertical division on the left there is the possibility to select between two sub-panels (tabs), each one with one tree structure containing the several elements/variables that can be integrated. The first referred to as Types, is where the user can configure and create the three possible POUs (Program, Function Block, Functions), the derived data types, and aspects concerning the *configuration*. According to [6] the standard does not include syntax to declare derived data types in graphical languages, however Beremiz allows the programmer to define the new data types through the use of a graphical interface, using for this purpose pull-down menus and lists. This of course is not standardized.

The other sub-panel contains all the POUs instances and elements of a specific Program POU, declared in a specific task. Here there is a high level detail showing the actions and transitions of a SFC (if it exists), and the IDE also presents different symbols for different instance types (inputs, outputs and local variables).

The central division is where the user writes the code according to one of the five languages defined in the standard, and the framework includes the possibility to create/edit both graphical and textual languages. When the user selects one programming language the top vertical pane changes its appearance, screening the elements that compose the selected language.

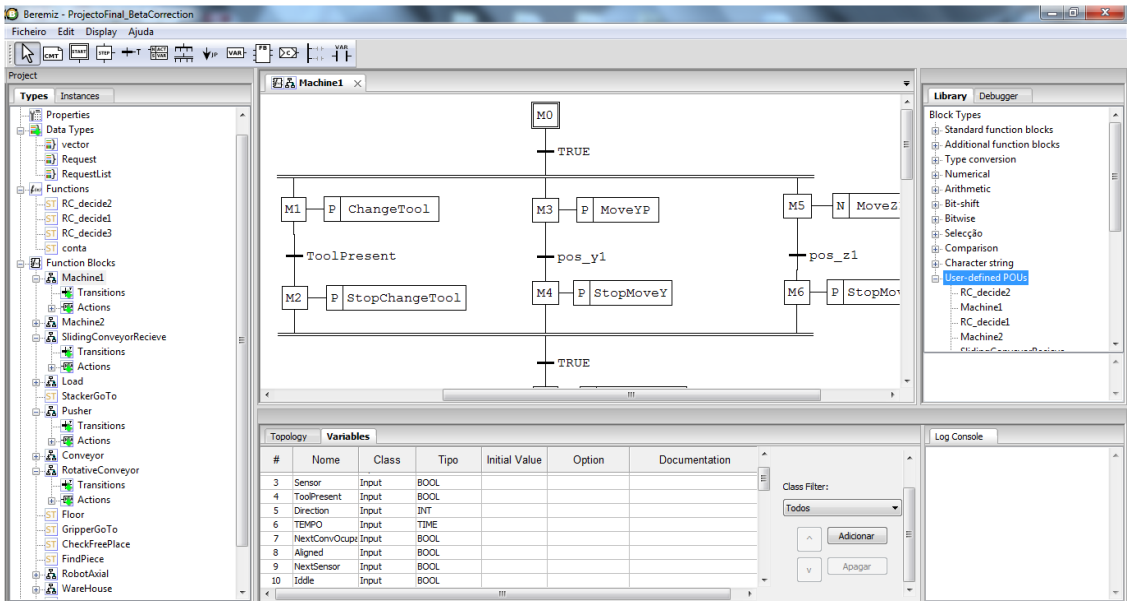

Figure 3.11 - PLCOpen Editor Window

The variables of a POU are located on the button division. It is possible to add and delete variables in this list as well as to edit the fields of each variable (name, type, location, initial value, retention and constant attribute). Variables may also be dragged and dropped in the central division in all languages when a user is editing a program, a simple but useful detail.

In the right division there are two more sub-panels which the user may choose from. The first concerns the standard functions and function blocks (basic building blocks), organized by scope of use, being that the last list is dedicated for the user developed POUs. In this sub-panel is also possible to drag POUs instances and drop them in center division. When FBs are dropped in the central division, they must be declared as the standard dictates. In the other sub-panel "Debugger", it is possible to drag and drop variables from the instances sub-panel, but this can only be done in running time. Thus, it is possible to monitor each separate variable of every POU. In fact, during running time, the *instances sub*-panel is able to monitor the status of all the variables declared in the respective POUs of a specific program. It is possible to monitor separated variables as described above (using the debugger sub-panel) or even to observe the status of an entire POU (if this is written using a graphical languages as SFC) in central division. For this purpose the user needs to double click on the specific POU. This particularity is only implemented in graphical languages and allows monitoring of variables in a whole program context.

Another particularity is that PLCOpen Editor is strongly linked to PLCOpen specification. This organization is not another standardization committee, but rather a group with a common interest wanting to help existing standards to gain international acceptance. Further information about this organization can be found in [13].

This concrete specification defines an XML grammar describing the five IEC 61131-3 languages. All automation programs written in this environment are saved into XML files, according to this grammar. It is then possible to exchange projects with other IEC 61131-3 editors that are in accordance with the PLCOpen. The data model follows the illustration showed in Figure 3.12. Basically, the structure relates to a XML file (*.xsd) that is used when a project is created, establishing all the relations between the objects that compose the project. These rules allow that PLCOpen Editor validates if a specific project in that format follows the referred base structure.

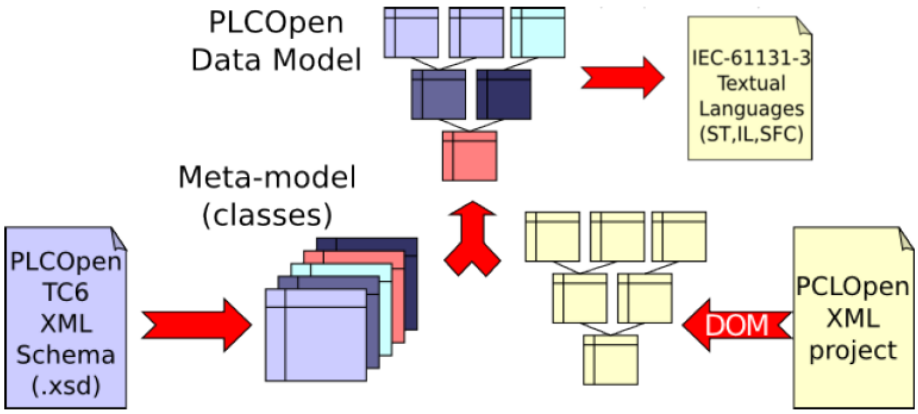

Figure 3.12- Inheritance of the data model in TC6 - XML Schema (Source: [9])

PLCOpen editor also integrates a responsible module to convert the graphical languages (FBD, SFC and LD) in their textual equivalent. Concretely, FBD and LD are converted in equivalent ST, while SFC elements have their own textual specifications which are defined in PLCOpen organization.

3.3.4. MatIEC 61131-3 Compiler

The textual conversion result of an IEC 61131-3 project referred in last subchapter is the consumed object by the MatIEC compiler in order to produce the equivalent C code. The organization of the compiler is illustrated in Figure 3.13. This compiler works through four main stages: lexical analyzer, syntax parser, semantics analyzer and code generator. The details of operation are explained in the official web site in [14] and more briefly in [15].

The lower block in Figure 3.13, shows how the compiler organizes the C code generated after compilation. All POU parameters and variables are accessible through nested C structs and located variables are declared as extern C variables [9].

The SoftPLC control algorithm is executed by initiative of a specific module referred to as Target Specific Code in Figure 3.13. It is responsible for managing the specific clock of the target platform and generating the cadential interruptions for the execution of the tasks.

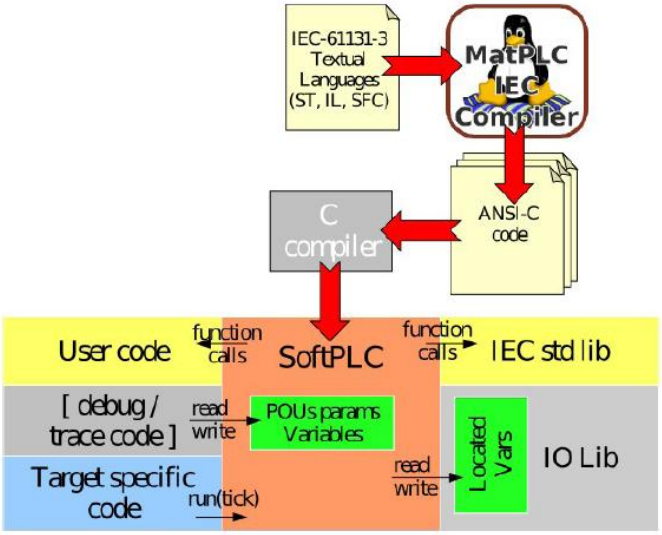

Figure 3.13 - Compilation global stages and generated code organization (Source: [9])

The program accesses to a set of Functions and FBs (created by the user or std lib), that are defined in a specific module and it receives as parameters the C structures mentioned above. MatIEC has also another module responsible for the consumption of the actual state of all the parameters of the program in runtime. Those parameters can after be presented to the user through the PLCOpen Editor as described in the previous Subsection.

3.3.5. Plugins

The plugins in Beremiz provide to the SoftPLC the possibility of communicating with the outside world. In fact all the control actions are closely associated to the necessary communication with logical or physical devices which are in direct contact with the process that has to be controlled. Sensors, actuators, and HMI are some examples of those devices.

All the plugins in Beremiz are composed of a user interface and a C component code that provides a set of services to the softPLC.

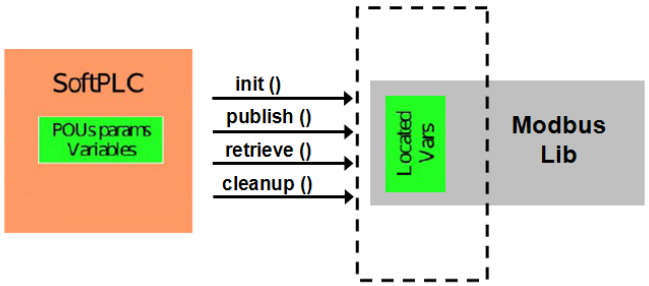

Figure 3.14 - Interface between the softPLC and a specific Beremiz plugin (Source: [8])

Briefly, and according to [8], the softPLC in Beremiz provides two services to the user: *Run* and *Stop*. The first one makes the initialization of the configuration, and also initializes the plugins that were added to the project as well as the routine that manages the cyclic execution of the control algorithm. This cyclic execution follows the behavior of a typical PLC (read inputs -> Execution of the control algorithm -> write the outputs), and this control

cycle beyond call the run_(config) routine, which runs the control algorithm itself, also uses exactly before and after the plugins routines named *retrieve _()* and *publish_()*. The *stop* service resort the plugins *cleanup_()* routines before completing the ongoing processes that disrupt the mentioned actions. Figure 3.14 illustrates the used routines of an interface between a SoftPLC and a specific Beremiz plugin.

Physical input and outputs variables are hierarchically organized in a plugin tree. Each plugin is associated with a range of IEC-61131-3 *directly/symbolic represented variables*. During build, these declared variables are dispatched in plugin tree according to their location, and consumed by plugins to produce corresponding C code [9].

Tabela 3.4 - Plugin utilization example and PLC variables association (Source: [9])

	Plugin IEC_Channel	Possible Variable Location
CANOpen plugin	0	
1st CANOpen	0.0	%IX0.0.3.323.1
Network	0.1	%IX0.1.3.323.1
2nd CANOpen	1	
Network	1.0	%IX1.0.3.323.1
HMI plugin	1.1	%IX1.1.3.323.1
1st Display		
2nd Display		

Chapter 4

Development

4.1 - Control Application Objectives and Services

The purpose of this work is the control development for the flexible line described in chapter 2, to be used afterwards in demonstration sessions. Therefore, an effort was made to simplify the available services to the user in order to enable an easy viewing by those who are watching/using the flexible line. The line is composed of five modules, each one performing concrete operations (see chapter 2). This way, an approach to the available services is based on these modules, and the following services to the user are defined:

- Machining of work-pieces using both serial and parallel machining plate
- Assembly of composed work-pieces using the assembly plate
- Work-pieces unloading using the load/unload plate
- Work-pieces loading using the load/unload plate

In order to give some dynamism and realism to the problem it was decided to consider four different types of work-pieces, distinguished by colours. Thus the work-pieces were labelled in the following colours: yellow, red, green and blue (note that these colours are easily distinguished by the user/observer of the line).

For the management of the work-pieces flux to be transported on conveyors of the flexible line, it was decided to define order execution paths. The upper conveyors sequence performs the work-pieces transport from left to right; the sequence of the lower conveyors performs the transport from the right to the left, and the movement within the plates is always performed from the top to the bottom. This can be visualized on Figure 4.1.

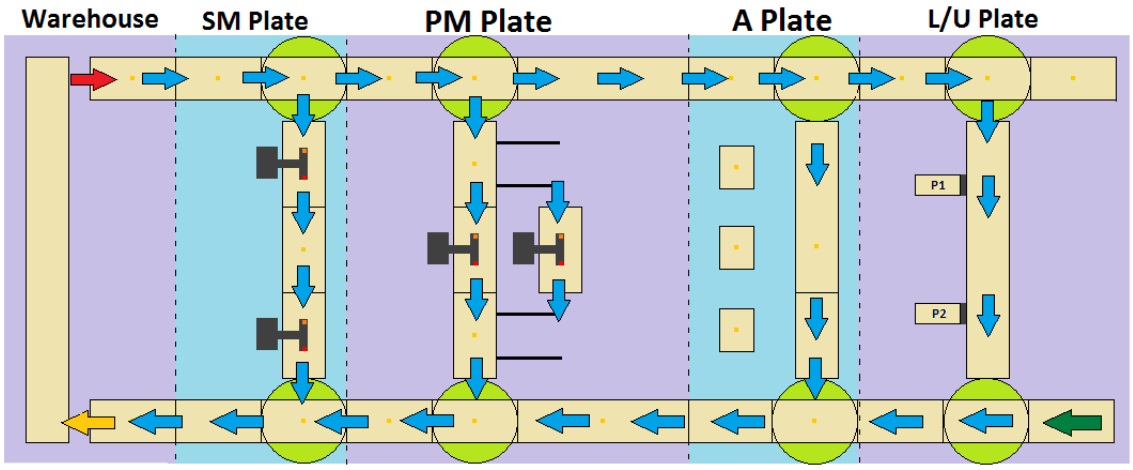

Figure 4.1 - Work-pieces flux on the flexible line

It is given that when the program starts, there are no work-pieces on the flexible line, neither in the warehouse. Because of this, the initial and only request that can be demanded is the load of work-pieces into the flexible line. According to Figure 4.1, the plate that performs the load of the work-pieces is to be found in the lower right corner (dark green arrow). For this request it's only required the colour of the work-piece. Therefore when the work-piece is placed on the corresponding conveyor, the presence sensor will be activated and the work-piece is automatically transported to the warehouse and later stored. This request can be called at any time and as long as the warehouse has one or more work-pieces available other requests can be satisfied.

For machining of work-piece requests, the user has to specify the quantity of work-pieces that he intends to machine and the corresponding work-piece colour. Naturally, if the amount of requested work-pieces is bigger than the existing number of work-pieces in the warehouse, the request is rejected, and the user is notified with an appropriate message. The selection of the plate for machining (serial or parallel) is not to be specified by the user when he performs a machining request. The control algorithm decides in which of these two plates the work-piece should be routed to. The implemented logic follows this approach: The work-pieces should move to the parallel plate if the upper sliding conveyor (see Figure 4.1) is not busy, otherwise the work-pieces should move to the serial machining plate. Another issue is, in which of the drilling machines the work-piece should be processed. It was decided that it has to be processed in both drilling machines; the same logic is used for the parallel plate. Different machines are used in both plates (see chapter 2). The serial plate machines are formed by three distinct tools. When the controlled algorithm starts, one of those tools is in machining position. This is considered tool number one (default tool). The user can at any time change this tool separately for both machines, specifying which one he intends to use to process the work-pieces. The same happens regarding machining time for the four machines installed in both plates, where it is possible to choose separately the time for the four different existing machines in the machining plates. After being processed the work-pieces are sent back to the warehouse.

For requests regarding work-pieces assembly, the user has to request the order in which the work-pieces are to be assembled, considering that the maximum number of simple work-pieces that a composed work-piece can have is three, and also specify the number of

composed work-pieces to be performed. Once more, before processing the request, the total amount of work-pieces necessary to attend this assembly is previously asked. If the number of the existing work-pieces is inferior to the number of the requested work-pieces, the request is rejected and an appropriate message is sent to the user. It is possible to perform assembly request with two or three work-pieces, outside this range requests are rejected. After being accepted in the assembly plate, the work-pieces are piled on the assembly-table situated in the middle of the existing three. After assembly work-pieces are sending and stored in the warehouse. It is possible to attend posterior machining requests and unload of composed work-pieces.

For unloading requests, the user has to request again the color of the work-piece he intends to unload, as well as the quantity. Once again before the request is attended, it is checked if the amount requested is not bigger than the amount available in the warehouse. If this happens, the user will again be notified with an appropriate message. In this kind of request it is possible to unload work-pieces to different pushers located in this load/unload plate (see chapter 2). The user does not specify this precise field when a request is made, as it is possible at any time to alter the pusher to where the work-pieces will be unloaded, in a similar way to what happens with the machining request times and with the change of tools on the same machining requests for the serial plate (remember that parallel plate machine present just one possible tool ready to be used). By default, when the program starts the selected pusher is the one that is found on the upper part of the plate (P1 on figure 4.1). After work-pieces are unloaded, these are routed to containers located in front of each pusher as described in chapter 2. These can store locally only two simple/composed pieces, but when a request is presented, if the selected container is full, the request is equally processed. However, when the work-piece reaches the selected pusher it stops in front of it. In this case an appropriate message will be sent to the user, referring this fact. If other work-pieces are sent meanwhile, these will be queuing in a sequence, in each conveyor, but always one work-piece for each conveyor. When work-pieces are removed from the full pusher container, the work-pieces in the queuing sequence (if that is the case), will be unloaded by sequence order. The procedure regarding the sequence of work-pieces in case a unit (pusher or machine, or even the stacker) is busy (full container of a pusher, or a machine that is processing a specific work-piece during sometime) is always the same, which is: each conveyor behaves as a single unit and therefore if a work-piece is being transported and at a certain moment the following conveyor is busy, the piece that is being transported will stop exactly on the previous conveyor. When the next conveyor is finally free, the waiting work-piece will continue forward – case of the output conveyor attached to the warehouse on which the work-pieces stop and wait for a posterior storage, or even for work-pieces that are being processed in machining plates.

For the work-piece storage, the stacker proceeds always the same way regarding the work-pieces organization inside the warehouse. The stacker organizes the work-pieces in rows always starting in the lower corner from left to right. When a row is full of work-pieces, the next row exactly above starts to be filled equally from left to right. In case the warehouse is full up, the stacker will stop storing work-pieces inside the warehouse until additional storage becomes available. In fact the number of available work-pieces is less inferior to the capacity of the work-pieces in the warehouse, however this issue was not overlooked. The stacker has

the same behavior (relating to work-pieces storage) when it is necessary to take out work-pieces from the warehouse to attend the requests.

The requests are organized in an array before being processed. This way, the flexible line will be able to attend and execute requests simultaneously. The requests are fulfilled as follows:

1. The user chooses among the three types of existing requests: machining, assembly and unloading (note that the loading request of work-pieces is different from the rest, being sufficient to specify the colour of the work-piece and place it on the appropriate conveyor for loading to begin),
2. afterwards, he should specify the colour or the sequence of colours of the work-pieces and the number of work-pieces to perform,
3. once the previous fields have been specified, the user should activate a boolean variable (rising edge of Add_Request variable) adding this way the request to the list of requests(an array with capacity of five requests),
4. Back to step one, if the user intends to add more requests, since the maximum number for the request list is five. In case the user intends to operate the request list, even if it is not full (less than five requests), he has to activate a boolean variable (rising edge of Start_Processing variable).

The requests start to be performed by the factory only after step four. While the requests list is not empty (all the requests processed), is not possible to continue with more requests. The requests are processed by arrival.

The warehouse stacker manages the input conveyor (interface conveyors to where work-pieces wait to be stored up) and the output conveyor (interface conveyor to where work-pieces according to requests are placed) as follows: If there is no existing requests to be processed from the list, the stacker turns to the input conveyor (in case work-piece input into the flexible line or even when the requests list is empty but there are work-pieces still running in the flexible line), otherwise it is given priority to the output conveyor. Meanwhile in case that there are still requests being processed and more than three work-pieces waiting to be stored, the warehouse stacker gives priority to this later situation, and moves to store work-piece (in case there are exactly three work-pieces waiting), proceeding afterwards to processing requests. This approach was made to avoid congestion of work-pieces in the flexible line.

4.2 - Control Application Architecture

The analysis of functionalities and features of the elements forming the flexible line and of services before mentioned on chapter 4.1, it was concluded that the control application architecture should have a good scalability (ease of adding components and/or switching the disposition of these components), and be composed by different layers that function with an abstraction level, offering services to other layers.

Thus, it was decided to use *object-oriented programming* paradigm, allowing to attain the distributed characteristics and the hierarchy described above. Therefore, the application to developing will be based on distinct classes, each one performing a certain task or operating a part of the flexible line through different functions. These classes are grouped in layers, organized hierarchically, supplying its functions to the exterior by means of services.

The class diagram is shown on figure 4.2, where each layer is designated with a color in the following way: orange for the upper layer, green for the intermediate layer and blue for the lower layer.

The lower layer (blue color), is composed of several components that constitute the modules of the flexible line such as linear conveyor, rotary conveyor, the both existing machines in the machining plates, and the pushers. As these are the "smaller" components forming the flexible line which are directly connected to the hardware (sensors and actuators) they are considered lower layer elements. Note that as in the existing machines, the pushers are fitted with a built-in conveyor, these are considered in the diagram as classes that inherit the class linear conveyor. The same happens in the multi-spindle drilling type machine given that it presents the same functionalities of the horizontal drilling machine and some more.

The intermediate layer is composed of the class 3AxilRobot, Floor and Warehouse. The class floor is composed of all the elements of the lower layer as shown in figure 4.2, so that this class will instantiate all these same components. The class warehouse is responsible for managing all the functions of warehouse and the class 3AxilRobot controls the actions of the robot in the assembly plate. Finally the ManufacturingLine is itself composed of classes Floor, warehouse and 3AxialRobot (this class will instantiate those objects). This class is responsible for the management of requests coming from the user as described on the previous subchapter 4.1.

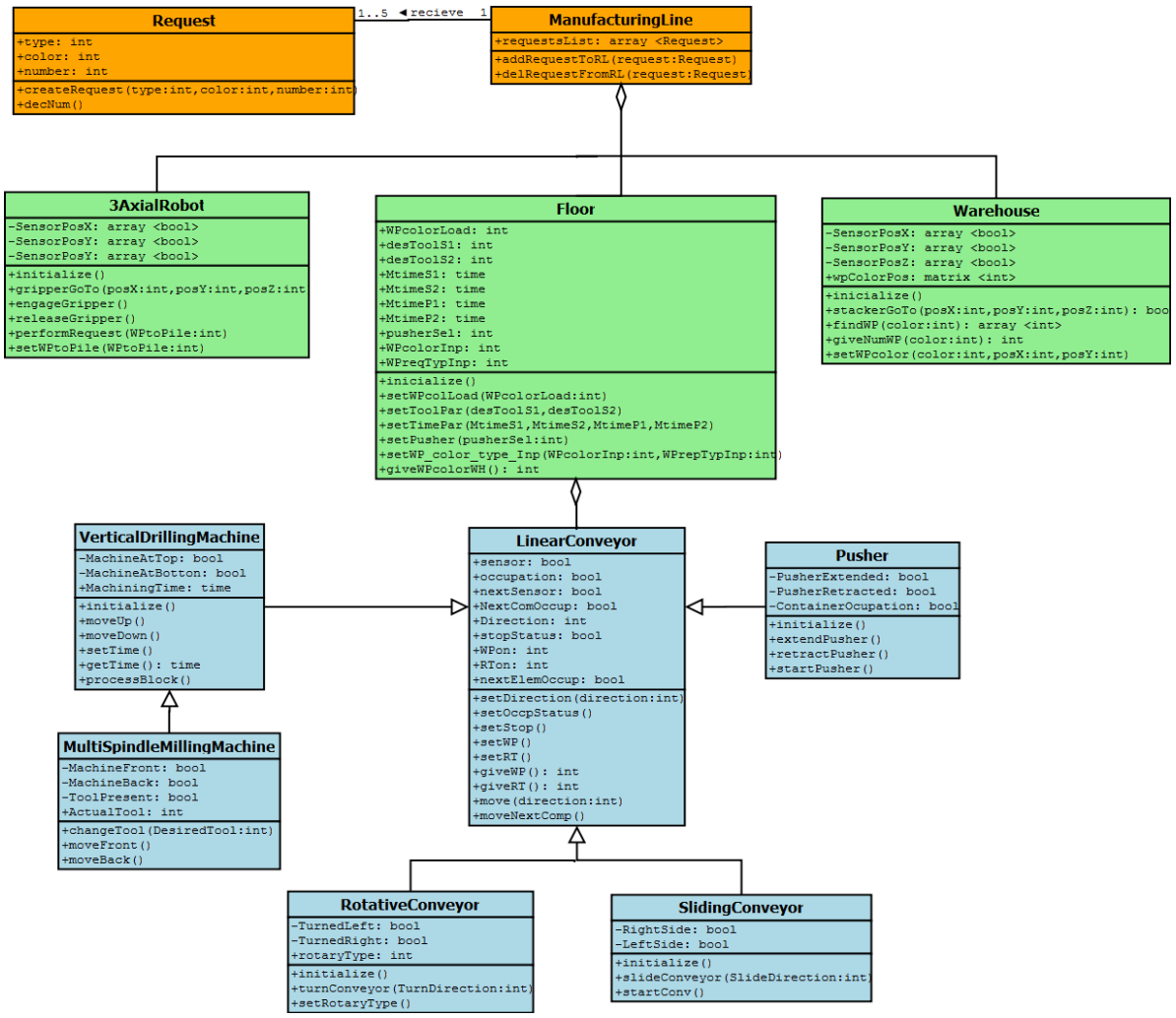

Figure 4.2 - Class diagram

The main services of the classes will be approached on the following subchapters organized according to the layer hierarchy already discussed. State and activity diagrams for the more important classes will also be presented in order that the reader can understand the main ideas of the control algorithm implemented.

4.2.1. Lower Layer

The LinearConveyor is perhaps one of the most important classes of the program, due to its number and its autonomy degree. I tried to implement a logic that would be as most independent as possible for this block. When objects of this class are instantiated, some parameters will have to be configured on the instantiation moment, and from there on these will only need to interact with the neighboring elements, such as other conveyors, machines, pushers, or even the stacker or the robot gripper. The implemented logic allows as well the possibility for objects of this class to behave as conveyors that only work for work-piece transfer, that is, receive the work-piece and then transfer it to a next conveyor or to conveyors that operate as input/output interface, similar to the ones present in the warehouse. When the conveyor is waiting one of two situations can occur as shown in the

state diagram of figure 4.3. One of them is the presence sensor to be active and an order is given for the conveyor to move (call). This is the case of the output conveyor that is attached to the warehouse where the command order to move it, is associated to the stacker of the warehouse that places the work-piece on the conveyor and then it gives the order for the conveyor to move. If the sensor is not activated, the conveyor transfers the work-piece until the conveyor own sensor is activated. Next it checks if the stopStatus attribute is active - the work-piece remains on the conveyor waiting for some other component to pick it up. If it's not active the work-piece will be transferred to the next conveyor. The component referred to can be the warehouse stacker that proceeds to the storage of the work-pieces on the input conveyor of the warehouse, or the robot gripper of the assembly plate. In this last case the work-piece is transferred until the input conveyor of the assembly plate and then the conveyor "calls" the robot to withdraw the work-piece from the conveyor and proceed to pilling work-pieces.

This way the linear conveyors can behave as input/output conveyors or act only for work-piece transferring. Another attribute important to describe is the "occupation" of the conveyor, this ensures that no other work-pieces will be transferred to the respective conveyor, in case it is busy transferring a work-piece or even when a work-piece is upon the conveyor. Only when the conveyor realizes that the work-piece was transferred to the next conveyor, it will be available to perform another request. This way we guarantee that one conveyor has only one work-piece upon it. Note that the "calls" for other components can include drilling machines, other types of conveyors (rotary or sliding), pushers, the warehouse stacker or the robot gripper. The diagram described on the Figure 4.3 also presents some simplification as well as the remaining diagrams that will be presented from now on. Remember that the main idea is to prove how the algorithm was implemented. Nevertheless an important aspect to describe and that is not related in the state diagram is the following: the type of request as well as the color of the work-piece is passed from one component to the other, and it is this way that it determines to which of the plates the work-pieces should proceed. It all starts on the output conveyor where the upper layers inform class Floor which kind of request for the work-piece as well as its color. From then on this information is passed from one component to the next component when the work-piece is transferred. When the work-pieces arrive at the warehouse, the stacker will recognize this way the color of the work-piece that is on the input conveyor.

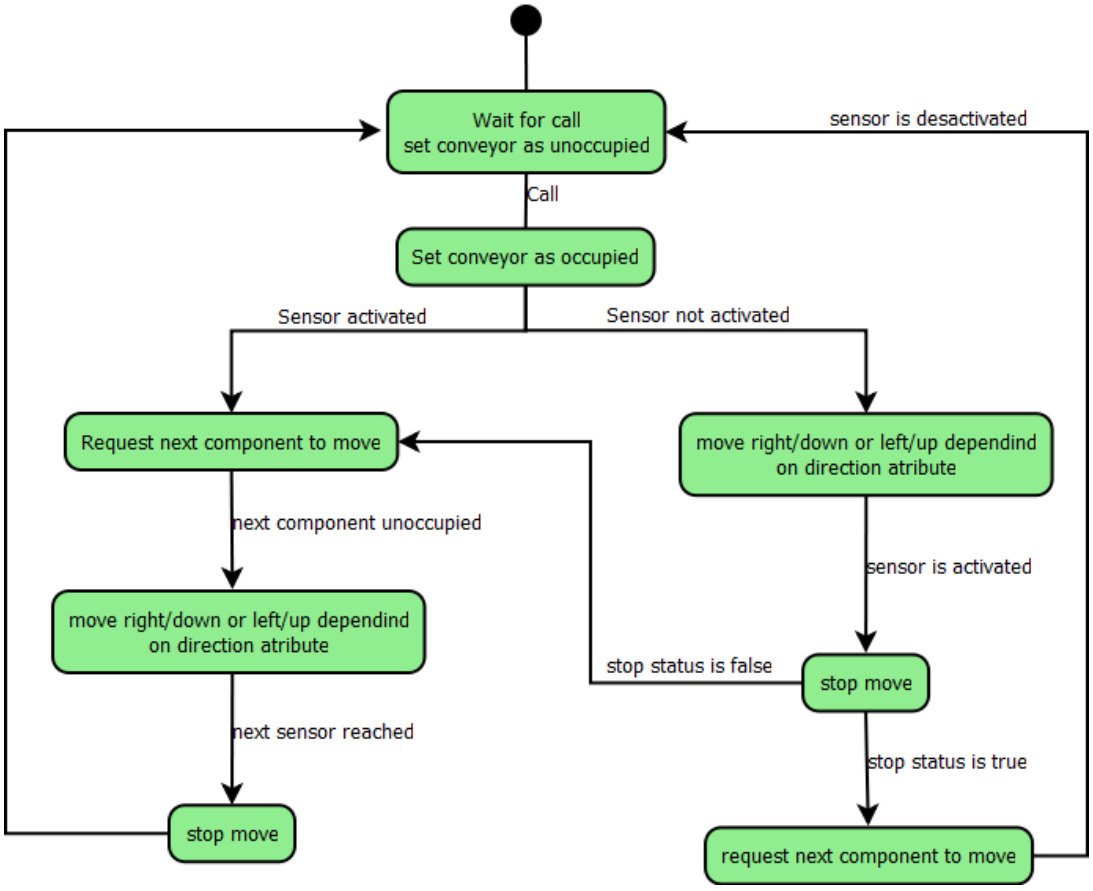

Figure 4.3 - Linear conveyor state diagram

The rotary conveyor is in all similar to the linear conveyor, except that it can spin around its own axel. When the approach analysis was described in subchapter 4.4.1, it was verified that two distinct cases for this kind of conveyor exist. The first case refers to rotary conveyors that are found in the upper sequence (see Figure 4.1). When the work-piece arrives at one of the four existing conveyors in this sequence, it will have to decide whether it will get in on a certain plate or proceed to the next plate. The second case refers to conveyors that are on the inferior sequence (see again Figure 4.1). Note that in these cases the conveyor can receive work-pieces from another conveyor positioned horizontally or from a conveyor positioned vertically that is on the interior of a specific plate. In the first case and for machining plates the decision is made based on the sliding conveyor of the parallel plate as described in subchapter 4.1. On the remaining plates the decision is made only concerning the request associated to the work-piece. For conveyors located on the inferior sequence the implemented logic was: if there is only one work-piece on the two possible next conveyors, the rotary conveyor performs the conveyor on which is the work-piece, otherwise if there are two waiting work-pieces to get in on the rotary conveyor it will give priority to the work-piece on the horizontal conveyor, rather than the vertical one. The entity that manages this operations is the class Floor of the intermediate layer that instantiates all the components as described in class diagram of figure 4.2. In conclusion when a "call" is made to the rotary conveyors, the first thing analyzed is which behavior that this conveyor should present based on what was previously referred to (turn or behave as a linear conveyor), and subsequently to proceed to the work-piece movement.

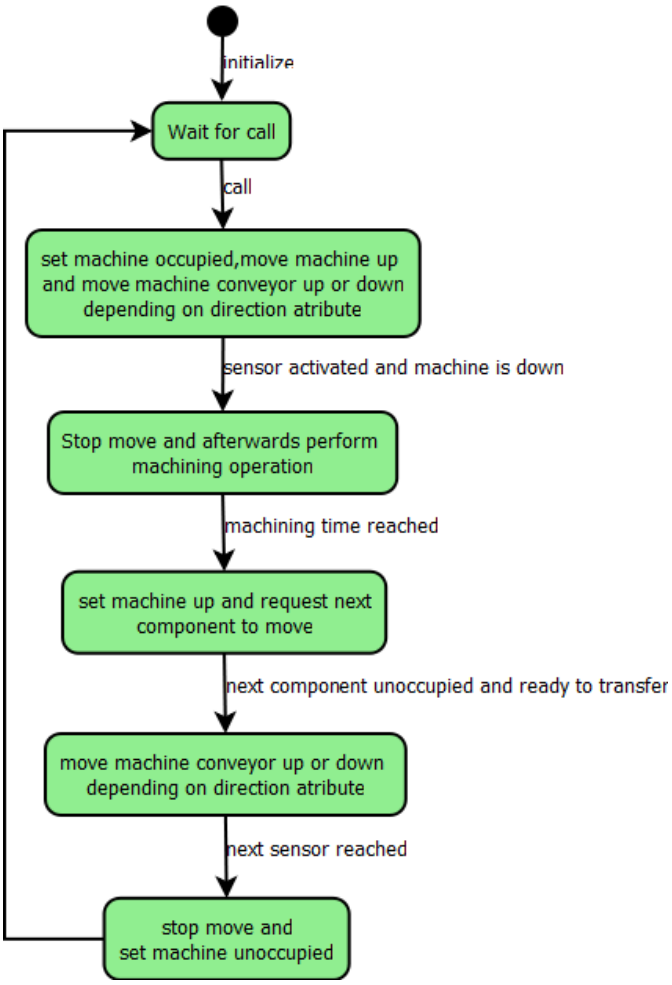

Figure 4.4 - Horizontal drilling machine state diagram

The behavior of the horizontal drilling machine is described in the state diagram in Figure 4.4. When the program starts running a verification is made if the machines are in the up position and in the case of the serial plate machines if they are pulled back and with one of the three available ready to machine. The behavior is equal to what happens on linear conveyors regarding transfer of which color of the work-piece, the type of request and the way how it manages the machine conveyor, not allowing the entrance of more than one piece at a time in the machine. Machining times can be changed at any moment and in the serial machines plate it is even possible to change among one of the three available tools as described in subchapter 4.1.

Pushers are also initialized when instantiated, verifying if it is extended - in this case it will be retracted. A detail of this component is that both existing pushers share the same conveyor which has two sensors. This way it was decided that an object of this class is formed by the set of the conveyor and both pushers. When a work-piece gets on the plate one of the pushers is selected to transfer the work-piece outside the flexible line, that can be selected any time by the user through the class Floor located on the intermediate layer.

4.2.2. Intermediate Layer

The class floor instantiates all the components of the lower layer floor. This is responsible for defining the surrounding components in each conveyor, machine or pusher. If the user intends to add, remove or alter any of the components of the lower layer it is here that changes must be made. Note the important and numerous modeling advantages and posterior programming of these problems using object-oriented model.

In figure 4.6 it is possible to visualize the correct coordination of the three components with the use of an activity diagram. The diagram refers to a linear conveyor, followed by a rotary conveyor, and a tool machine. It starts representing a sequence considering that one work-piece is already situated upon a linear conveyor, stopping when the work-piece is being processed in the machine, otherwise the diagram would be too extensive. Note how each state is associated to class attributes, normally related to physical sensors of the components. On the other hand, the operations are coupled to actuators of the components.

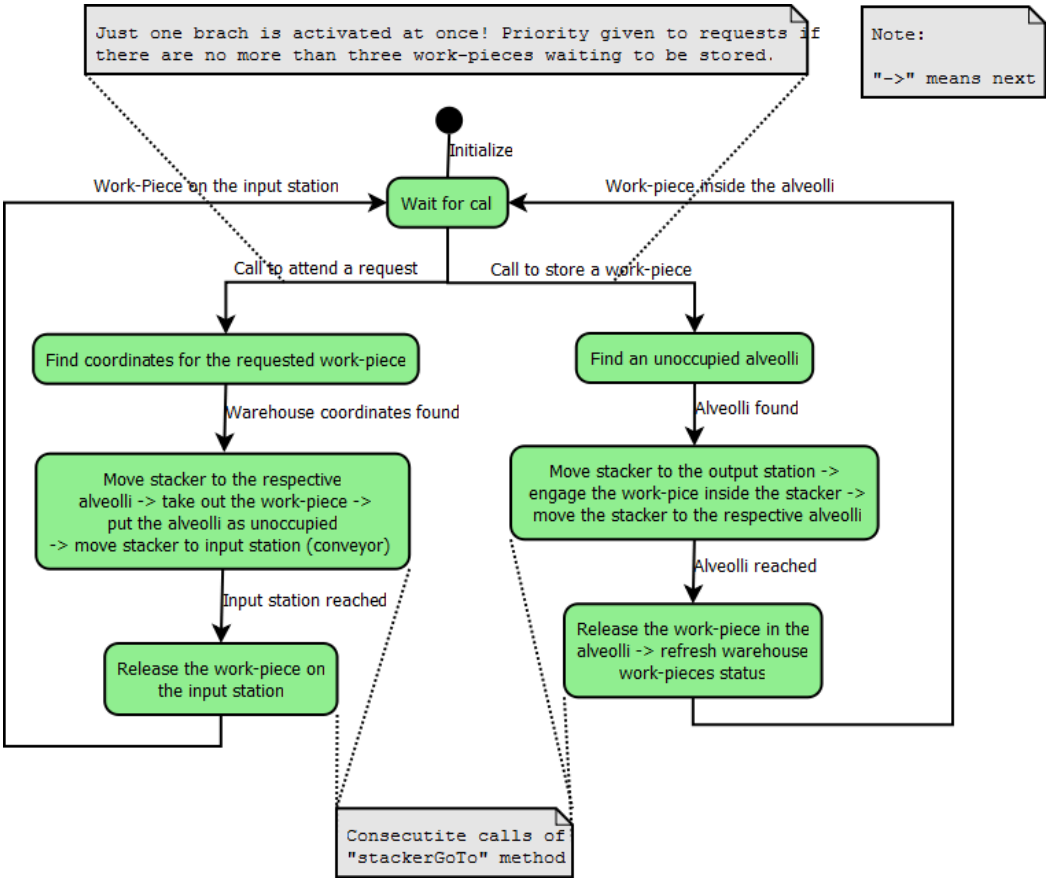

Figure 4.5 - Warehouse state diagram

The main behavior of the warehouse is described in the state diagram of Figure 4.5. During the initialization it is verified if the stacker is engaged in one alveoli. If this is the case the stacker moves to the outside and stand by for a "call". As described in the diagram, the stacker can be called to attend a request or to store a work-piece. The procedure to manage this fact is described in the diagram as well as in section 4.1. If the warehouse is "called" to

attend a request, the brief procedure is to find the coordinates of the warehouse where the work-piece is located (calls method findWP), remove that same work-piece, refresh the attribute wpColorPos (maps the type of work-piece that is inside each alveoli) through method setWPcolor and take the work-piece then to the output conveyor. In case the stacker attends a request to store a work-piece, the procedure is the following: locate a free alveoli, proceed to store of the work-piece in the given alveoli and update the attribute that maps the color of the work-piece that is present in each alveoli (attribute wpColorPos).

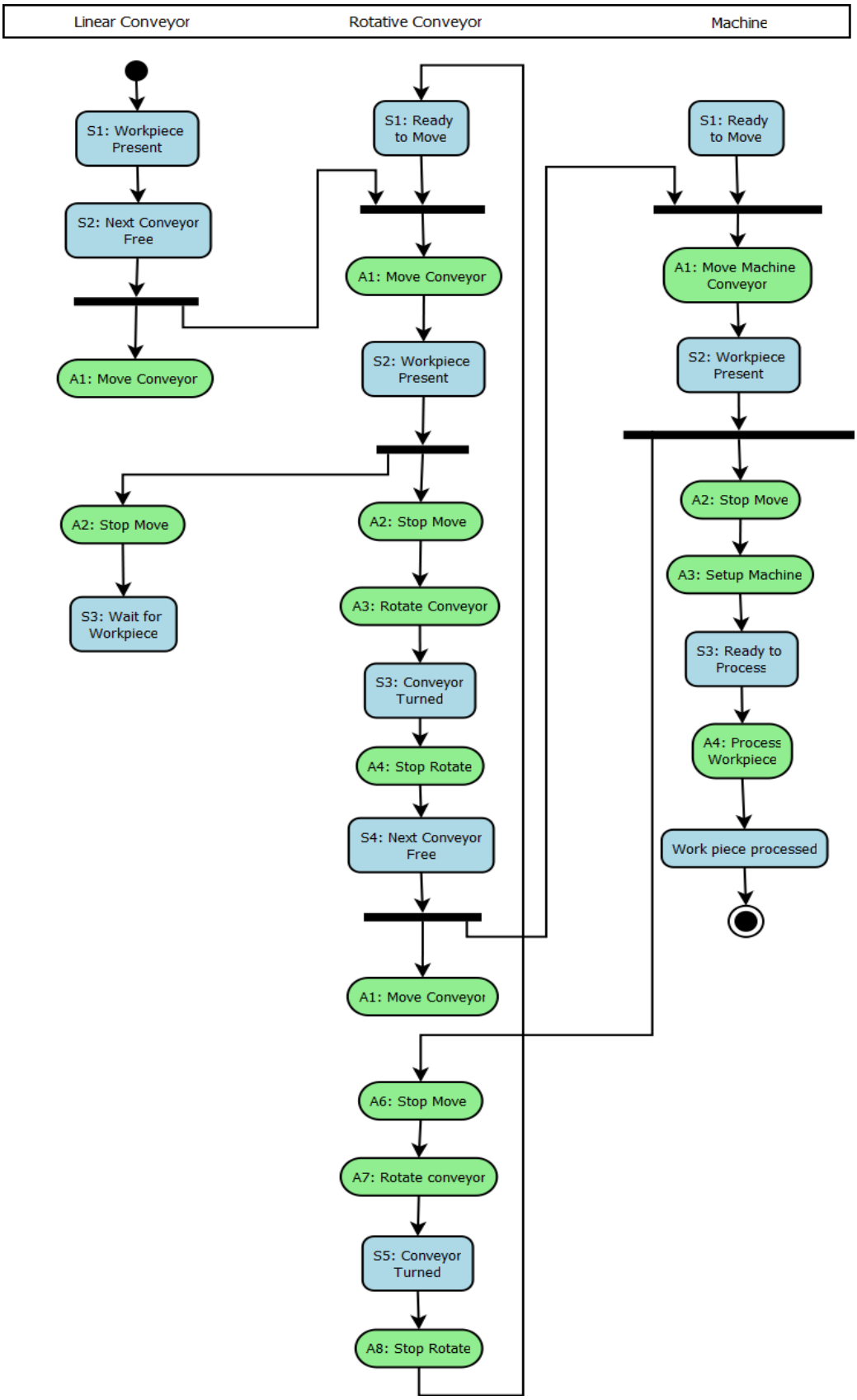

Figure 4.6 - Interlocking synchronization logic among three components using an activity diagram

Finally, the state diagram that represents the robot behavior is presented in figure 4.7. The "call" for the robot comes from the input conveyor where the robot will withdraw the work-pieces for posterior pilling on a work-table. When the robot performs the pilling operation it needs to know the number of pieces to be pilled through class Flexible line. After pilling all the number of work-pieces intended (two or three pieces) the robot uses the gripper to pick up the composed work-piece and move it to the output conveyor of the assembly plate.

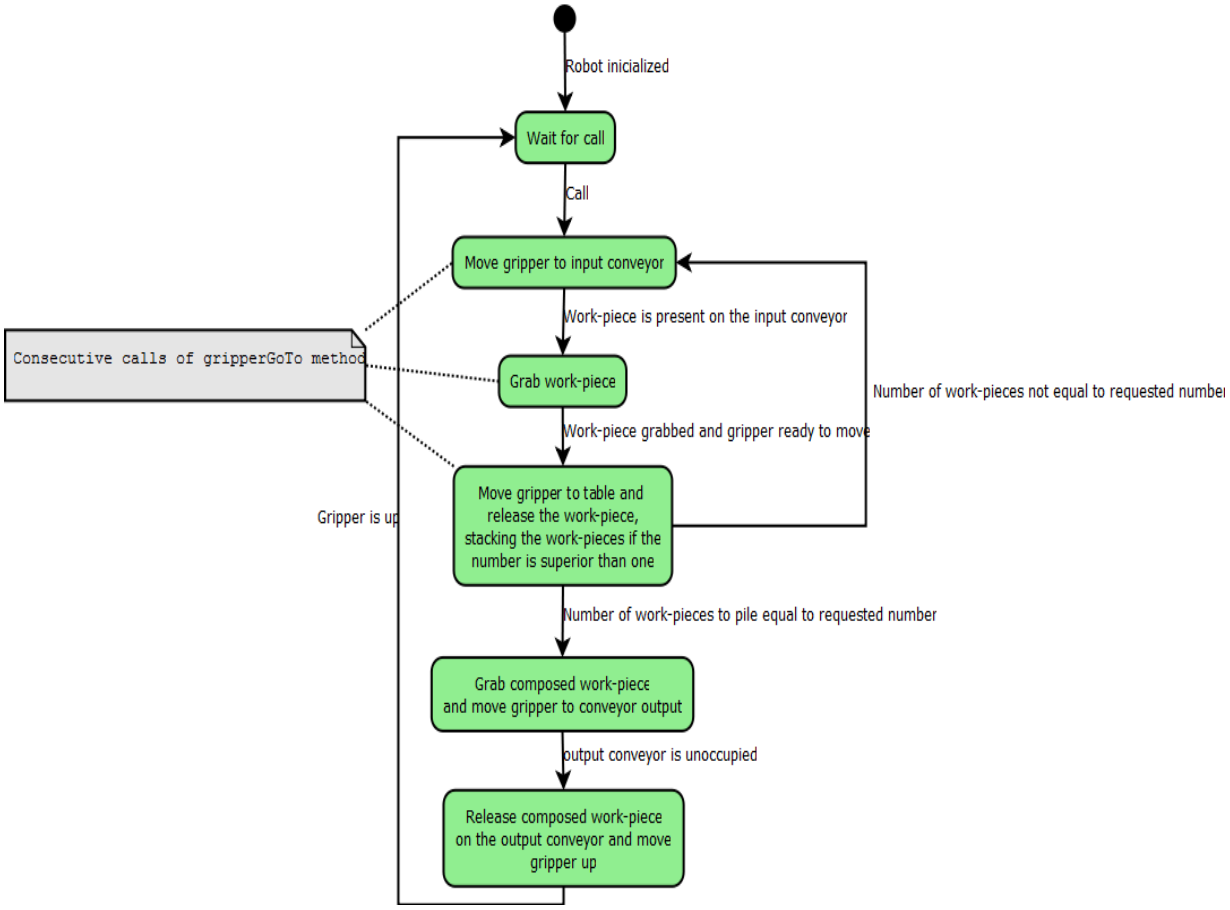

Figure 4.7 - 3AxialRobot state diagram

4.2.3. Upper Layer

Class factory is composed by classes of the intermediate layer and it is the class responsible for managing requests coming from the user. In case the user requests more work-pieces for a certain request then the ones existing in the warehouse, it is sent an appropriate message as referred in subchapter 4.1. So before processing a request (add a new request to the requests list), the fields that the user fills in are always checked. The behavior class is described figure 4.8.

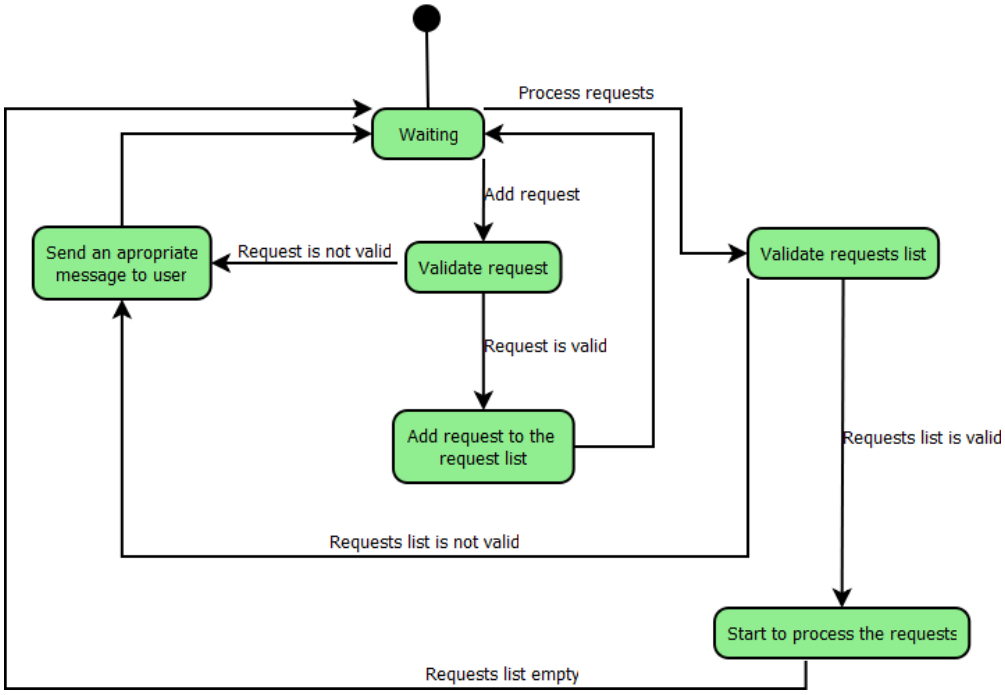

Figure 4.8 - ManufacturingLine state diagram

4.3 - IEC 61131-3 Implementation Details

In this subsection it is intended to show the POUs of standard IEC-61131-3 implemented in Beremiz, proving this way the strategy of the implemented control algorithm according to the architecture described in the previous subchapter.

The standard FBs are normally associated to classes in object orientated languages. This way all the classes described in the class diagram (see Figure 4.2) were separately implemented using different FBs.

When the implementation of the control algorithm is initiated, it is necessary to previously know in advance the classes that will be implemented depending on the approach of the problem, and it is also a good practice to divide the problem in abstraction layers in which, each of them supplies services to inferior/superior layers. However information flux between layers has to be well defined and limited.

The standard flexibility in terms of programming languages allows the FBs to be implemented using one of the five programming languages referred in subsection 3.2.3. The choice of the algorithm implementation was based on languages SFC, FBD and ST. In fact the SFC is the adequate language to model classes described in section 4.2 because the logic passage of the state diagrams to SFC is relatively easy (remember that SFC is based in state machines). Also for the reason that is a graphical language, it is possible to follow the evolution of the program in PLCOpen Editor of Beremiz in running time. FBD language was mainly used to connect building blocks among themselves, for instance, classes floor, robot and warehouse are declared in this language in class factory. In turn class factory was also declared in this language in a *Program Type* POU, which is instatieted in a task within the configuration of the SofPLC (note the encapsulation of the POUs objects). ST language was used to implement some FBs that are not the classes. These in turn are instantiated inside the FBs that implement the classes. This is the way how more than one method is

implemented in a specific class using the approach of IEC 61131-3 standard. For example, the FB rotary conveyor that is a class, uses a method referred to as "setRotaryType" which is another building block (but not a class) and it is "called" inside the rotary conveyor, being regarded as a method of this class.

The reason for the choice of these languages is explained by the fact that they are regarded as the three most powerful and flexible languages that the standard specifies, being that IL and LD are languages that exist in present time merely for historical reasons. Although the languages are not directly convertible to each other, some of them are more adequate than others for determined tasks, but still there are methods to convert a certain language into another. For example document [16] defines a conversion strategy of a program programmed in SFC to a program in LD.

Another aspect that is important to outline is: in a FB there are no constructors or destructors as in other object orientated languages such the case of C++ besides it can only have one default public function (the others methods are implemented using other building blocks as already mentioned).

Briefly a FB can be described as a block containing input, output variables or even variables that function as input and output simultaneously. These variables can be associated to other variables coming from other POUs. FB outputs develop depending on the logic of the respective FB code part and the inputs. This way the FBs inputs are associated to the sensors of the physical components such as the presence sensor of a conveyor and are seen in the class diagram as classes attributes. Whereas the FBs outputs are associated to actuators and this way connected to the methods of a class. For example for the case class conveyor the sensor and nextSensor attributes are linked with a conveyors physical sensors, and depending of the state in which the control algorithm is in, the necessary methods associated somehow to the methods of a class are performed, such as to run a conveyer to the left or to the right. This behavior can be verified in state diagrams described in chapter 4.2.

In the intermediate layer class Floor instantiates all the components (conveyors, pushers and machines) from the lower layer as mentioned in the previous subchapter. The connection of the components among themselves depends on the surrounding components that each comprises (see Figure 4.10). As an illustrative example the figure 4.9 indicates the connection between two linear conveyors using FBD language.

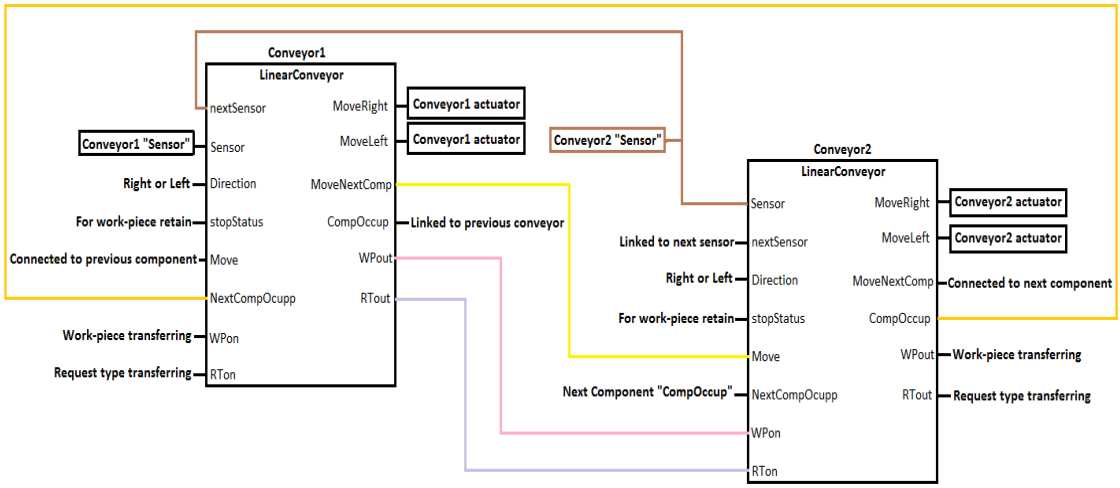

Figure 4.9 - Connection between two linear conveyors using FBD language

Figure 4.9 demonstrates the sensors connected as inputs - the sensor of a given conveyor and the sensor of the next conveyor, since the working state of a conveyor depends on the state of the next conveyor (see figure 4.3); apart from other necessary inputs for the evolution of the control algorithm, such as in which direction the conveyor should run, check if it has to retain the work-piece or transfer it to the next conveyor (stopStatus attribute), the busy state of the next conveyor (NextCompOccup attribute), that is, if the conveyor is busy with a work-piece or transferring a work-piece, and the variables related to the kind of request and the color of the work-piece that are passed among neighbor components. These variables are considered attributes on class diagram. Opposed to attributes the methods are combined with actuators, as the movement of a certain conveyor to the left or to the right, the moveNext that calls the next component, and the component occupation (CompOccup) that informs whether the conveyor is busy in a transfer or with a work-piece upon it, this in turn connects as an input in the previous conveyor (see figure 4.9).

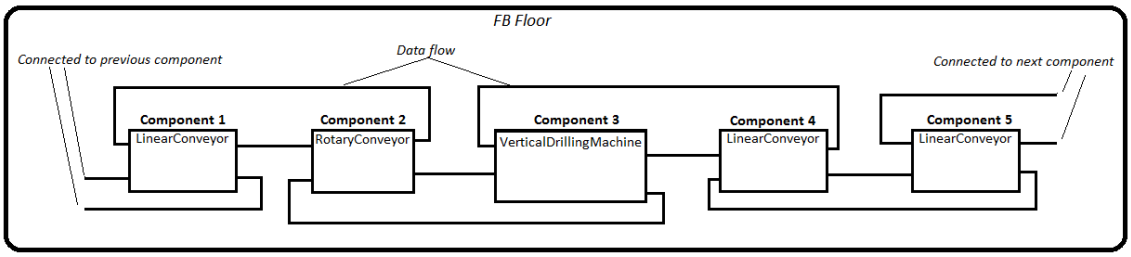

Figure 4.10 - Part of the FB Floor showing the connection between neighbor components

The class FB Floor besides instantiating all the components (conveyors, machines and pushers), also contains the sensors and actuators that form each one of those components. These are indeed seen as global variables and therefore declared in the configuration part of the SoftPLC (Beremiz). When these variables are called in class FB Floor will have to be declared as external variables (remember subchapter 3.2.3). Other functions were created in order to assist the FB floor. One of these functions is applied to an input variable of the rotary conveyors, indicating how these should behave when inputting a work-piece (remember section 4.1).

The robot and warehouse classes were also implemented with the use of FBs, being that these also invocate auxiliary POU types FB and Function. One of the FBs relates to gripperGoTo in the case of robot class and stackerGoTo in the case of warehouse class. Take note here on the flexibility that standard allows, existing the possibility of creating POUs in one of the five languages that the standard defines. The class FB warehouse also uses other functions, such is the case of member function giveNumWP described in class diagram (see Figure 4.2) which return the number of work-pieces of a certain color that exist in the warehouse.

Another relevant aspect that is worth to mention is the invocation among different types of POUs. This always follows what was mentioned in subchapter 3.2.2, where the program type POU can instantiate the other types of POUs (FBs and Functions), the FBs can instantiate others FBs or functions, and the functions can only call other functions. In the case of the Beremiz implemented algorithm, just one program was declared which instantiates the FB

factory, this is turn instantiates the FBs described in the intermediate layer of the class diagram, and class floor instantiates all the elements of the lower layer.

The implemented control algorithm is also composed of some user date types. In the warehouse for instance, it was created a multi-dimensional array (referred to as matrix in attributes of the class warehouse(see Figure 4.2)) was created to manage the color of the work-pieces that each alveoli contains. For class ManufacturingLine was created a structure named request (class request on class diagram of figure 4.2) that integrates the fields the user has to fill in when a request is made, and also a five element array that in turn contains elements of the request type (requestList on class diagram).

4.4 - Beremiz Evaluation

One of the goals of this project is to test and validate Beremiz tool which as was referred previously, tries to implement strictly the standard IEC-61131-3. Here it is intended to show the evolution of this tool along these months and also to show major problems that came up when the control algorithm for the flexible line (primary tool to validate this project) was in a developing state.

When this tool was used for the first time it was realized that this IDE had a relative lack of documentation when compared to its capabilities, leaving the user wondering where to start. The user's manual that comes along with the IDE only presents very briefly the way the IDE is organized, relevant aspects of the GUI and one example of the use of a Plugin CANOpen. There is no tutorial guiding the users to the aspects of the languages, or even the way one should proceed to force variables in a specific program (running time aspects). However it is clear in the manual that this follows the standard IEC 61131-3 and therefore the implementation aspects of the POUs use, data types and configuration have to follow what the standard determines. At the end of the manual it is stated that the associated IDE version is a preAlpha release, which means that we should not trust the software for any critical mission and consider that with the course of the time more functionalities will be available when other releases will be accessible.

It was decided to use the Linux version of this IDE the platform in which it was developed. The first tests were not pleasing because the platform presented some instability and short time was needed to find some bugs. In the first tests that were carried out in SFC language, the relevant elements *selection divergence* or *selection convergence* generate a bug when placed on the central panel of PLCOpen Editor, precluding this way the utilization of this powerful language. When new variables were added in a programmed POU in one of the five programming languages, it was also very frequent the program to simply shut down, presenting a message on the console where it was running with the message "segmentation fault" . This was probably the most frequent error, thus it was necessary to save the project very often to prevent loss of data. Another major problem was the instability of the IDE in running time when the variables to be monitored were added on debug panel, being that type time variables were not supported, generating a bug each time they were added to the referred panel.

In the end of March, this year, a new release of the IDE came up referred to as Spring 2011 1.01 bugfix release. This in contrast to the previous version was clearly more stable and the problem with SFC language mentioned above was already solved. The frequent mistake

"segmentation fault" was not so frequent but it still occurred with a certain frequency, mainly in softPLC running time. Variables TIME type were yet not supported in debug panel, but after a short time (about a week an a half) this bug was corrected.

SFC language started then to be used, to develop the main POUs of the program. It was realized that this language presents some small aspects then the standard dictates and that are not implemented, such as:

- Is not allowed to use hierarchical SFCs in SFCs actions
- Is not possible to access state variables (state.X and state.T)

There are also qualifiers related to SFCs actions that do not work properly, such is the case of reset and limited qualifiers. The standard predicts also that function blocks can be associated in a program in SFC, permitting to combine graphically the FBs to the transitions of a SFC POU . Although this functionally is implemented in PLCOpen Editor it presents bugs. The test produced was to use the standard FB TON, because it was necessary to use time type variables for the tool machines and the state.T state variable was not accessible. The alternative was to use the same standard FB (TON) inside an SFC action in ST. Other standard FBs or functions did not work properly, as the case of selection (SEL) and type conversion FBs. So it was decided to implement on my own the functions/FBs that were being necessary during the implementation.

When the implementation of the intermediate layer was started, it was figured out that the user data types were not working correctly. For example in class warehouse, it was necessary to use a bi-dimensional array to manage the colors of the work-pieces that were stored. When the user data types were created and then called in POUs (except for functions), it was not possible to attribute values from other variables, for example:

```
(*Considering i  an integer variable and V an array*)

i:=5;
v[1]:=i;
(*or even*)
v[i]:=3;
```

The same happened with the rest of the data types. In fact the program could have some more user data types, as an example, to define a color of the work-pieces it would be possible to use the enumeration data type, but these were not working properly and to move forward with the project it was decided to use variables of integer type. The IDE did not accept either that derivated data types were attributed as input/output variables in POUs. These last bugs about derivated date types were very recently solved and implemented in the last remaining weeks of the project development.

Other small details that are not in accordance with the standard that were found during algorithm implementation were the return user data types in functions and instantiates a FB directly in a task of a resource.

Perhaps the most frustrating aspect during all the implantation time was the way how the errors are presented to users. In this IDE, when the user makes some programming errors, this are shown in the console panel when the program is compiled. However many times just a

simple "internal error" message appeared, leaving very unclear where the error came from. It happens multiple times to be forced to erase all a POU and implement it step by step, that is, compiling as soon as it was being programmed to insure that the POU would work properly. Other errors are presented as hundreds of lines when the user just forgets to insert ":" in an attribution of value in a certain variable.

Apart from all the mentioned errors, a great effort has been done in the last months to fix the bugs that Beremiz presents. Almost all or even all the bugs before mentioned are already fixed in this moment. There is a proper space on the site named "mailling list" where users can discuss matters and problems related with this framework.

Besides all the problems and time spent in the implementation of control algorithm, it was possible to validate Beremiz, because the implemented algorithm was tested in the flexible line successfully. Some good aspects of this tool, would also be important to point out, mainly:

- Be a free, open source and multi-platform software
- Easy utilization by user interface (PLCOpen Editor)
- Modular implementation, allowing new user to aim new modules (plugins)
- MatIEC compiler generating C code that can run in any platform or microcontroller.
- Tries to follow strictly IEC-61131-3 standard

4.5 - Graphical User Interface

To provide the user a functional form to control and monitor the flexible line, it was also a goal to develop a graphical user interface (GUI). Unfortunately due the problems mentioned in the previous chapter it was not possible to its implementation. However the architecture was defined and in this subchapter it is intended to show alternatives and the important procedures to develop the graphical user interface for the developed control algorithm.

It was decided at the beginning of this project that the final application would later run in hardware ICNova AVR32 AP7000 (the same hardware that runs the interlock system (see Chapter 2)), operating as a softPLC after transferring the C code generated by MatIEC compiler of Beremiz. In addition, the procedures to apply a GUI in the control algorithm was decided based on the existing technologies already installed in the flexible line.

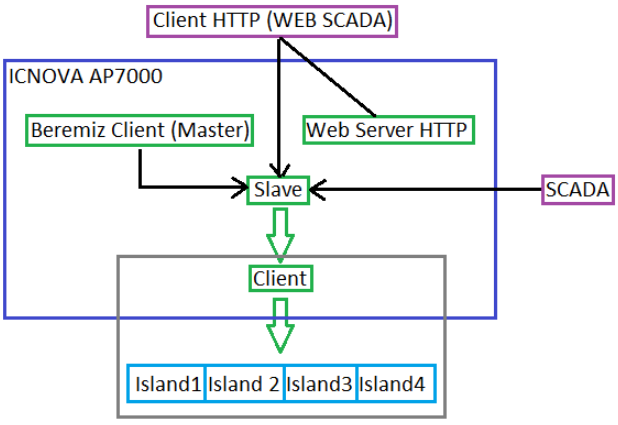

Figure 4.11 - Network architecture for the ICNova AP7000 installed on the flexible line

Figure 4.11 illustrates the network architecture of platform ICNova AP7000 (dark blue rectangle) that will run the generated control algorithm. Remember that in the generated control algorithm was used a plugin in Beremiz for the communication (modbus TCP) between the flexible line and the softPLC. This plugin for communication according to the referred protocol, was developed based in the dissertation of an ex FEUP student and just implements a simple Master modbus tcp [8].

The interlock logic implemented on this board uses the slave (modbus/TCP implementation) represented in the Figure 4.11 which maps all control variables (sensors actuators) of the flexible line linked to the four existing islands (light blue rectangles). Thus, the strategy would be to profit from the same slave to map necessary variables for the GUI, such as variables related to the user´s request. Note that in control algorithm those variables would also to be addressed (in Beremiz Master plugin), to insure the correct connection between Master-Slave of modbus protocol.

Two different architectures were idealized to develop the GUI. One of them would be elaborate a SCADA using an IDE such as LabView from national instruments [17] or Vijeo Designer from Schneider Electric [18]. As these suggested IDEs support modbus TCP protocol there wouldn't be any problems to connect between these devices. Another more versatile suggestion would to be to implement a web SCADA, due to the fact that the board ICNova has a HTTP web server integrated (see Figure 4.11). Here the idea to develop a logic in the web server in order to enable this to access the variables located in the slave, and on the client side to be able to use javaScript language that would run in the GUI side (client side).

In the SCADA to develop it would be possible to monitor the major part of the components of the flexible line, such as all the work-pieces transportation, being possible to visualize the status of each conveyor (conveyor movement or active status), verify the color of the work-piece on the conveyors and its respective kind of request, the storehouse status (inside organization and work-piece color) to name just a few. To proceed to the mentioned points it would be necessary to make use of the sensors and actuators of the flexible line as well as some variables created in the control algorithm. As an example to know the location of the work-pieces on the conveyors of the flexible line, the strategy would be as follows: If a presence sensor of a given conveyor is active, it means that a work-piece is upon it; if two connecting conveyors are in movement but no sensor is active (remember that conveyors only have a single sensor located in the center) it means that a work-piece is located between these two conveyors. When a work-piece is spotted it is also possible to know which color and the kind of request related to it, by the use of the work-piece color variables and the kind of requests passed between neighboring components in the control application (see Subchapter 4.2). For the warehouse study it was created a multi-dimensional array that maps the positions for each work-piece color inside it. With this variable it would be possible to have an accurate idea of the present status of the warehouse. The messages that warn the user that a certain request was not accepted (to request more work-pieces than the ones existing), are associated to boolean independent variables (one for each error linked to each request), and when they are changed from false to true (rising edge) it means that they were activated and the information about the error of the request should be screened on the GUI.

To summarize and conclude it will be necessary to map in the existing modbus slave to the variables created in the softPLC (Beremiz) that are intended to be used in the GUI, moreover the request´s variables are compulsory because without them the user won´t be

able to interact with the flexible line. Afterwards address this variables in the softPLC (Beremiz) and only then on the master responsible for running the control algorithm is able to read and write the variables associated to the SCADA located in the mentioned modbus slave. SCADA will later on behave as a client (Master) in the connection to the ICNova slave.

Chapter 5

Validation, Conclusions and Further Work

5.1 - Validation

In a first approach of the algorithm implementation, it was tested with the use of a shoop floor simulator [19], that acts as a Modbus server in a TCP/IP network. The simulator was made in java, uses the jamod library [20] to act as Modbus server and is similar in the functionalities available in flexible line used in this project, although with some simplifications. The implementation of this simulator enables a file (plant.proprieties) where aspects from configurations of the TCP connection to graphical elements can be defined.

For the tests carried out with the simulator, a configuration as similar as possible to the flexible line was used, defining this way the same five plates presented in the flexible line, such as warehouse, serial plate, parallel plate, assembly plate and load/unload plate. Figure 5.1 illustrates its graphical aspect. In the plugin of Beremiz, the IP was configured in loopback (127.0.0.1) and the port was defined with the number 5502. The components used in the simulator determine the memory zones that are available.

The warehouse in the simulator is simplified in relation to the one of the real flexible line, being that to proceed work-piece unloading the user simply needs to write in a register the color of the work-piece that is intended to be remove, and for storage perform in a boolean signal (actuator), which to pass from false to true (rising edge) indicates the work-piece present on the warehouse interface conveyor to store it. The conveyors (linear, rotary and sliding) are in all similar to the ones in the flexible line as well as the pushers (but this without container). The tool machines are equal in the two plates and are very similar to the machines existing in serial plate of the flexible line (multi-spindle drilling machine). On the

assembly plate there is no 3 axis robot, so in this plate only the disposition of the conveyers agrees with the one of the flexible line, work-pieces pilling is not possible though.

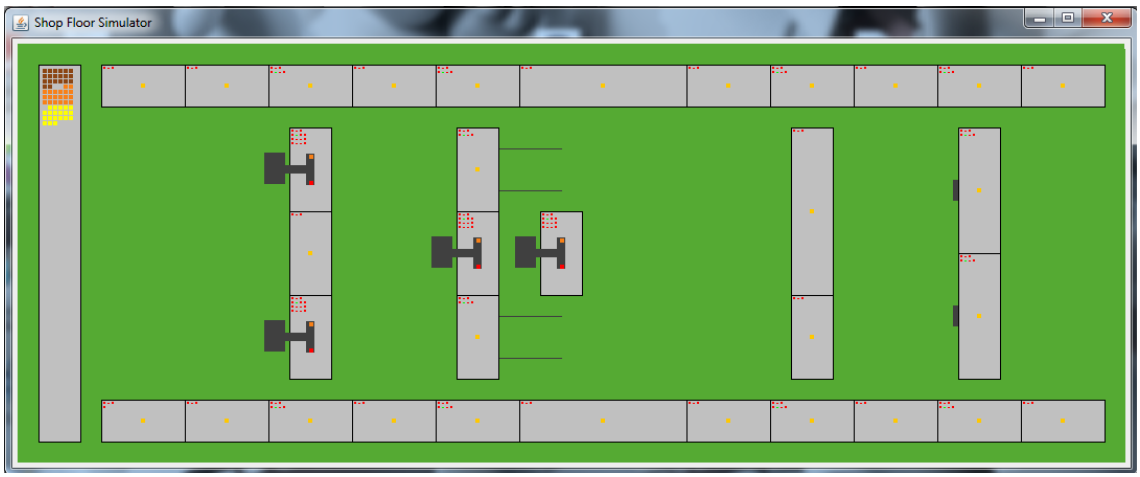

Figure 5.1 - Graphical aspect of the Shop Floor Simulator

The register for the work-piece color is mapped in an *Output Register* and all the rest of the simulator components are mapped as *Input Bits* and *Coils* (*Output Bits*). These items (inputs and outputs) are mapped in independent memory zones, each one of them addressed from zero (the same behavior for the real flexible line).

Although the existing simplifications to the java simulator, it was very useful to test on the first level the POUs of the lower level in the class diagram (see Figure 4.2) as well as all the transportation logics (class floor described earlier in Chapter 4). Classes warehouse and 3axialRobot could not be tested by using the simulator due to the fact that these did not contain all the necessary components, therefore the control algorithm started to be tested in the real flexible line. In a later stage it began the implementation of classes related to the assembly plate and to the warehouse, always testing each module separately in order to realize about the correct functioning. In the end it was implemented class ManufacturingLine and the connection of all FBs was performed. The executed tests were used to verify if the request list behave has described in section 4.1, filling the list with the three possible requests and with different sequences. The aspects that can be changed in run time as the machining time, or the pusher selection to unload pieces was also tested, proving all these tests the functioning described in section 4.1, can thus validate the implemented solution.

5.2 - Conclusions and Further Work

The objectives set for this project were to test and validate an open source framework, which implements the standard IEC 61131-3. Hence, the tool used to permit the validation of the control algorithm and consequently the project, was an assembly line existing at the DEEC. This line was already assembled and ready to be used; therefore the first approach was to study all the modules compounding this assembly line, in order to be able to establish afterwards which services would be available to the user.

For the algorithm implementation, it was decided to make use of the UMI abstraction layer in order to define the classes of the system and to expose the logic applied to the most

relevant classes through state and activity diagrams. Next, the abstraction of this architecture was checked on the programming model that the IEC 61131-3 standard establishes, showing differences comparatively to other object orientated languages, and also the way the programmer should proceed for the implementation of projects accordingly to IEC 61131-3 standard. It is thus expected that this document credits some interest as a support in modeling and conception of solutions in the area of industrial automation relying on standard IEC 61131-3.

It was furthermore object of study an IDE open source, the Beremiz. This framework for automation experienced a great evolution throughout these last months, however it is important to recall that during the implementation of the control algorithm there was a great deal of effort with the aim to finish the solution projected for the flexible line. There were serious problems as described previously in section 4.4 and if many of them hadn´t been solved during these last weeks, it wouldn't be possible to validate the proposed solution. This tool was thus exhaustively tested, the more diverse problems and aspects that don't agree with the standard, but still at the end of this project it was possible to validate the tool.

Initially, it was also planned the development and testing of a Graphical User Interface (GUI) so as to allow the final user to manage the flexible line more comfortably. Unfortunately, it was not possible to accomplish this, mostly because of the problems arising all through the implementation of the control algorithm. However, the approaches to the most important procedures to implement it, as well as a presentation of different implementation architectures have been described. It is expected that in future work there will be some availability to proceed the execution of a GUI, in order to make possible to perform demonstration sessions on the flexible line by any user.

Referências

[1] Srinivas Medida, "Pocket Guide on Industrial Automation."

[2] Edouard Tisserant, Laurent Bessard, and Mário de Sousa, "An Open Source IEC 61131-3 Integrated Development Environment."

[3] Modbus-ida. http://www.modbus.org/.

[4] Modbus-IDA. MODBUS Application Protocol Specification v1.1b, December 2006

[5] Modbus-IDA. MODBUS Messaging on TCP/IP Implementation Guide v1.0b, October 2006.

[6] Mário de Sousa and Adriano Carvalho. Programming with the IEC 61131-3, Languagues and the MatPLC.

[7] John, Karl-Heinz and Tiegelkamp, Michael, "IEC 61131-3: Programming Industrial Automation Systems," 2001.

[8] Vasco das Neves Fernandes, "Driver Modbus para Aplicação IEC 61131-3," June 2009.

[9] LOLITECH, Beremiz user manual, 2008. Avaliable in http://www.beremiz.org under GNU free Documentation License v1.2

[10] Edouard Tisserant, Laurent Bessard, and Mário de Sousa. "An Open Source IEC 61131-3 Integrated Development Environment." INDIN 2007, 2007

[11] Python bindings to the wxWidgets cross-platform toolkit. http://www.wxpython.org/.

[12] WxWidgets. A C++ cross-platform GUI library. http://www.wxwidgets.org/.

[13] PLCOpen TC6 XML Official Schema. http://www.plcopen.org/pages/tc6_xml/xml_intro/index.htm

[14] MatIEC official web site. http://mat.sourceforge.net/

[15] Mário de Sousa and Adriano Carvalho, "An IEC 61131-3 Compiler for the MatPLC."

[16] Eric Anderson, "Sequencial Function Chart to PLC Ladder Logic Translation," August 2009

[17] LabView. http://www.ni.com/labview/

[18] schneider-electric. http://www.schneider-electric.com

[19] André Restivo. Shoop floor simulator. Avaliable: http://github.com/arestivo/sfs/.

[20] Dieter Wimberger. Java Implementation of Modbus protocol. Avaliavle: http://jamod.sourceforge.net

[21] Manfredi Bruccoleri, "Reconfigurable control of robotized manufacturing cells," December 2005