



Universidade Federal  
do Rio de Janeiro  
Escola Politécnica

## IDENTIFICATION OF A DIDACTIC MANUFACTURE SYSTEM

Rafael Accácia Nogueira

Projeto de Graduação apresentado ao Curso de Engenharia de Controle e Automação da Escola Politécnica, Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Engenheiro de Controle e Automação.

Orientador: Marcos Vicente de Brito Moreira

Rio de Janeiro  
Abril de 2019

IDENTIFICATION OF A DIDACTIC MANUFACTURE SYSTEM

Rafael Accácio Nogueira

PROJETO DE GRADUAÇÃO SUBMETIDO AO CORPO DOCENTE DO CURSO  
DE ENGENHARIA DE CONTROLE E AUTOMAÇÃO DA ESCOLA POLITÉCNICA  
DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS  
REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE ENGENHEIRO  
DE CONTROLE E AUTOMAÇÃO.

Examinado por:

---

Prof. Marcos Vicente de Brito Moreira, D.Sc.

RIO DE JANEIRO, RJ – BRASIL  
ABRIL DE 2019

Nogueira, Rafael Accácio

Identificação de um Sistema de Manufatura Didático/Rafael Accácio Nogueira. – Rio de Janeiro: UFRJ/ Escola Politécnica, 2019.

XVII, 121 p.: il.; 29, 7cm.

Orientador: Marcos Vicente de Brito Moreira

Projeto de Graduação – UFRJ/ Escola Politécnica/  
Curso de Engenharia de Controle e Automação, 2019.

Bibliography: p. 104 – 106.

1. Discrete Event Systems. 2. Identification. 3.  
Failure Detection. I. Moreira, Marcos Vicente de Brito. II.  
Universidade Federal do Rio de Janeiro, Escola Politécnica,  
Curso de Engenharia de Controle e Automação. III.  
Identification of a Didactic Manufacture System.

*“It’s a dangerous business going  
out your door. You step onto the  
road, and if you don’t keep your  
feet, there’s no knowing where  
you might be swept off to.”*

*(J.R.R Tolkien)*

# Agradecimentos

Primeiramente a Deus, sem quem nada é possível e por **todas** as pessoas colocadas em meu caminho, que me fizeram crescer e ser o indivíduo que hoje sou.

Aos meus pais, Rosemeri e Rogério. Por todo amor, carinho, atenção e apoio dados, pela primeira educação, essencial para toda minha trajetória, educação não só acadêmica, mas também moral. Obrigado, por tudo ! Amo muito vocês.

A todas minhas professoras e professores por mostrarem o quão importante e bonita é a profissão e por terem sempre instigado a sede pelo aprendizado. Agradeço àqueles que contribuíram para minha base acadêmica e profissional.

As amizades que fiz, as que se foram de minha convivência e as que permaneceram, agradeço aqueles que conheci na UFRJ, mais especificamente a nossa turma T17, pois se chegamos até onde chegamos foi porque estivemos juntos, fortes, ombro no ombro, tentando não deixar o outro cair, mas quando alguém caía sempre uma mão amiga se estendia para ajudar a levantar e recomeçar.

Ao Paulo Yamasaki, pelo convívio no LABECA, e pelas trocas de ideias em assuntos gerais que por fim, intencionalmente ou não, se tornariam orientação em diversos projetos que fiz na faculdade, e até mesmo orientação acadêmica e profissional.

Aos melhores companheiros de grupo, Gabriel Pelielo e Rodrigo Moysés, um verdadeiro “Power Trio”. Também a Philipe Moura e Felipe Matheus, que me incentivaram a sair da minha zona de conforto e me fizeram compreender de fato o sentido do quão “perigoso” é sair pela porta de casa, pois quando saímos da nossa zona de conforto, coisas mágicas podem acontecer e pessoas mágicas podem aparecer em nossas vidas.

À Evelise, a pessoa mágica que apareceu em minha vida, que me ajudou fisicamente e psicologicamente nos momentos que mais precisei. Obrigado por escolher compartilhar parte de sua vida comigo e por toda a força dada para o término desse ciclo. Eu te amo!

Por fim às pessoas que me ajudaram mais diretamente neste projeto, Ryan Pitanga e ao meu orientador Marcos Moreira

Abstract of Undergraduate Project presented to POLI/UFRJ as a partial fulfillment of the requirements for the degree of Automation and Control Engineering.

## IDENTIFICATION OF A DIDACTIC MANUFACTURE SYSTEM

Rafael Accácia Nogueira

April/2019

Advisor: Marcos Vicente de Brito Moreira

Course: Automation and Control Engineering

This work has as primary objective to propose tools and a methodology for identification of discrete events systems using the **Deterministic Automaton with Outputs and Conditional Transitions (DAOCT)** model, which can be used to failure detection. In order to accomplish this, the control of a didactic manufacture system will be designed, using petri nets in a first phase and then converting it into Ladder logic. Once the control is implemented, the inputs and outputs of the plant will be logged and then fed to the **DAOCT** model identification algorithm. Each one of this steps will be depicted in this work and the identified model will be discussed.

1. Discrete Event Systems.
2. Identification.
3. Failure Detection.

Resumo do Projeto de Graduação apresentado à Escola Politécnica/ UFRJ como parte dos requisitos necessários para a obtenção do grau de Engenheiro de Controle e Automação.

## IDENTIFICAÇÃO DE UM SISTEMA DE MANUFATURA DIDÁTICO

Rafael Accácio Nogueira

Abril/2019

Orientador: Marcos Vicente de Brito Moreira

Curso: Engenharia de Controle e Automação

Este trabalho tem como objetivo propor ferramentas e uma metodologia para a identificação de sistemas a eventos discretos, utilizando o modelo **DAOCT**, que poderá ser usado para detecção de falhas. Para tanto, será realizado o projeto de controle de um sistema de manufatura didático, utilizando em uma primeira fase redes de petri, depois convertendo na linguagem Ladder. Uma vez implementado o controle os dados de entrada e saída da planta serão registrados e depois dados como entrada para o algoritmo de identificação do modelo **DAOCT**. Cada um desses passos serão descritos nesse trabalho e o modelo identificado será discutido.

1. Discrete Event Systems.
2. Identification.
3. Failure Detection.

# Contents

<b>List of Figures</b>	<b>x</b>
<b>List of Tables</b>	<b>xiii</b>
<b>List of Acronyms</b>	<b>xv</b>
<b>List of Symbols</b>	<b>xvii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Thesis Outline . . . . .	2
<b>2 Background</b>	<b>4</b>
2.1 Systems . . . . .	4
2.2 Discrete Event Systems . . . . .	5
2.3 Languages . . . . .	5
2.4 Representation of Languages . . . . .	6
2.4.1 Automata . . . . .	7
2.4.2 Petri Nets . . . . .	9
2.5 Control Interpreted Petri Nets . . . . .	14
2.6 Implementation of Control Interpreted Petri Nets . . . . .	18
2.6.1 Ladder Logic . . . . .	18
2.6.2 Conversion from Control Interpreted Petri Net (CIPN) to Ladder Diagram (LD) . . . . .	22
2.6.3 Petri Net divided in multiple PLCs . . . . .	24
2.7 Identification . . . . .	27
2.7.1 DAOCT . . . . .	29
<b>3 System</b>	<b>35</b>
3.1 Magazine Unit . . . . .	37

3.2	Conveyor Belt . . . . .	38
3.3	Sorting Unit . . . . .	38
3.4	Handling Unit . . . . .	40
3.5	Assembly Unit . . . . .	42
3.6	Storage Unit . . . . .	43
<b>4</b>	<b>Control</b>	<b>45</b>
4.1	Logic . . . . .	45
4.1.1	Initialization . . . . .	46
4.1.2	Metal Cube Half Sorting . . . . .	49
4.1.3	Plastic Cube Half Sorting . . . . .	52
4.1.4	Arm From Conveyor Belt to Assembly Unit . . . . .	55
4.1.5	Assembly Unit . . . . .	58
4.1.6	Arm From Assembly Unit To Storage Unit . . . . .	60
4.1.7	Storage Unit Positioning (y Axis) . . . . .	63
4.1.8	Storage Unit Positioning (x Axis) . . . . .	66
4.1.9	Cube Storage . . . . .	69
4.1.10	Arm Stop Logic . . . . .	72
4.2	Implementation of the Control . . . . .	75
<b>5</b>	<b>Identification</b>	<b>79</b>
5.1	Data Acquisition . . . . .	79
5.2	Model Identification . . . . .	90
<b>6</b>	<b>Results</b>	<b>93</b>
6.1	Identified Model . . . . .	93
6.2	About the choice of the first vector . . . . .	98
<b>7</b>	<b>Conclusion</b>	<b>101</b>
7.1	Concluding Remarks . . . . .	101
7.2	Issues and Workarounds . . . . .	102
	<b>Bibliography</b>	<b>104</b>
	<b>A Complete Petri Net</b>	<b>107</b>

<b>B Tools</b>	<b>116</b>
B.1 daoct . . . . .	116
B.2 dot2automata . . . . .	118
B.3 dot2petri . . . . .	119
B.4 Other Scripts . . . . .	120

# List of Figures

2.1	Input/Output Process model . . . . .	4
2.2	State Transition Diagram . . . . .	7
2.3	Diagram representing the automaton from example 2.1 . . . . .	8
2.4	Component nodes of a petri net. . . . .	9
2.5	Diagram representing the petri net structure from example 2.2 . . . . .	10
2.6	Diagram representing the petri net structure from example 2.2, but with $Post(p_1, t_1) = 2$ . . . . .	10
2.7	Same diagram as Figure 2.6 but with labeled arcs. . . . .	10
2.8	Example of unmarked and marked petri net graphs. . . . .	11
2.9	Example of petri net Dynamic. . . . .	12
2.10	Labeled Petri net. . . . .	13
2.11	Example of Petri net with inhibitor arc. . . . .	14
2.12	Representation of new labeling function . . . . .	15
2.13	Representation of a timed transition. . . . .	15
2.14	Representation of labeling of Actions. . . . .	16
2.15	Example of System to be controlled by the Petri Net . . . . .	17
2.16	Example of Control Interpreted Petri Net to control system in Figure 2.15	17
2.17	Types of Contacts. . . . .	19
2.18	Types of Coils. . . . .	19
2.19	And logic in a Ladder rung. . . . .	20
2.20	Not logic in a Ladder rung. . . . .	20
2.21	Or logic in a Ladder rung. . . . .	21
2.22	Examples of function blocks. . . . .	21
2.23	Example of Control Interpreted Petri Net converted to Ladder. . . . .	24
2.24	Example of Petri Net divided between 2 PLCs. . . . .	25
2.25	Example of Petri Net divided between 2 PLCs. . . . .	26
2.26	Observed Signals in a closed-Loop Discrete Event System (DES). . . . .	27

2.27	Venn diagram showing the relation between $L_{Orig}$ , $L_{OrigNI}$ , $L_{Obs}$ , $L_{Exc}$ and $L_{Iden}$	29
2.28	State transition diagram for identified model using $k = 1$ .	33
2.29	State transition diagram for identified model using $k = 2$ .	33
3.1	Cube halves.	35
3.2	Units of the Manufacture System.	36
3.3	Magazine Unit.	37
3.4	Conveyor Belt.	38
3.5	Sorting Unit - Identification.	39
3.6	Sorting Unit - Discharging.	39
3.7	Handling Unit.	41
3.8	Assembly Unit.	43
3.9	Storage Unit.	44
4.1	Petri net of Initialization module.	48
4.2	Petri net of metal cube half sorting module.	51
4.3	Petri net of plastic cube half sorting module.	54
4.4	Petri net of manipulator taking a cube half from conveyor belt to assembly unit module.	57
4.5	Petri net of assembly unit module.	59
4.6	Petri net of manipulator taking cube from assembly unit to storage module.	62
4.7	Petri net of storage unit positioning module (y axis).	65
4.8	Petri net of storage unit positioning module (x axis).	68
4.9	Petri net of cube storage module.	71
4.10	Arm Stop Logic Angles	72
4.11	Petri net of manipulator Stop Logic module.	74
4.12	Siemens Programmable Logic Controller (PLC) S7-1500	75
5.1	DataLogCreate block.	80
5.2	DataLogOpen block.	81
5.3	DataLogWrite block.	81
5.4	DataLogClose block.	81
5.5	DataLogDelete block.	82
5.6	Example of DataBlock used to log data.	82
5.7	LOGDATA block.	84
5.8	Example of Data struct.	85

5.9	UpdateValues block.	86
5.10	Code inside UpdateValues block.	86
5.11	CompareArrays block.	87
5.12	PutInDataStruct block.	87
5.13	Code inside PutInDataStruct block.	88
5.14	Inputs/Outputs from Handling-Assembly-Storage PLC.	89
5.15	Identified model from paths extracted from .csv file using $k = 1$ .	91
5.16	Identified model from paths extracted from .csv file using $k = 2$ .	92
6.1	Number of states of identified model for different values of $k$ .	94
6.2	Comparison between the cardinality of the exceeding language generated by the DAOCT (o) and NDAAO ( $\times$ ) models, for $k = 1$ and $k = 2$ .	95
6.3	Number of states of identified model for different values of $k$ .	96
6.4	Comparison between the cardinality of the exceeding language generated by the DAOCT (o) and NDAAO ( $\times$ ) models.	97
6.5	Scheme of the example 6.1.	98
6.6	Identified model using $[0 \ 0 \ 0]^T$ as initial state, $k = 1$ .	99
6.7	Identified model using $[1 \ 0 \ 0]^T$ as initial state, $k = 1$ .	99
6.8	Identified model using $[1 \ 0 \ 0]^T$ as initial state, $k = 2$ .	100
B.1	daoct help dialog.	117
B.2	daoct input csv file.	117
B.3	daoct graphviz output.	117
B.4	daoct $f$ output.	117
B.5	dot2automata Help.	118
B.6	dot2automata output.	118
B.7	dot2petri Help.	119
B.8	dot2petri output.	120

# List of Tables

2.1	Control Interpreted Petri Net Example Places. . . . .	17
2.2	Control Interpreted Petri Net Example Transitions. . . . .	17
4.1	Initialization Module Transitions. . . . .	46
4.2	Initialization Module Places. . . . .	47
4.3	Metal Half-cube Selection Module Transitions. . . . .	49
4.4	Metal Half-cube Selection Module Places. . . . .	50
4.5	Plastic Half-cube Selection Module Transitions. . . . .	52
4.6	Plastic Half-cube Selection Module Places. . . . .	53
4.7	Arm From Conveyor Belt to Press Module Transitions. . . . .	55
4.8	Arm From Conveyor Belt to Press Module Places. . . . .	56
4.9	Assembly Unit Module Transitions. . . . .	58
4.10	Assembly Unit Module Places. . . . .	58
4.11	Arm From Press To Storage Unit Module Transitions. . . . .	60
4.12	Arm From Press To Storage Unit Module Places. . . . .	61
4.13	Storage Unit (Y axis) Module Transitions. . . . .	63
4.14	Storage Unit (Y axis) Module Places. . . . .	64
4.15	Storage Unit (X axis) Module Transitions. . . . .	66
4.16	Storage Unit (X axis) Module Places. . . . .	67
4.17	Cube Storage Module Transitions. . . . .	69
4.18	Cube Storage Module Places. . . . .	70
4.19	Arm Stop Logic Module Transitions. . . . .	73
4.20	Arm Stop Logic Module Places. . . . .	73
4.21	Inputs Selection PLC . . . . .	75
4.22	Outputs Selection PLC . . . . .	76
4.23	Inputs Handling-Assembly-Storage PLC . . . . .	77
4.24	Outputs Handling-Assembly-Storage PLC . . . . .	77

A.1 Complete Places. . . . .	107
A.2 Complete Transitions. . . . .	110

# List of Acronyms

CCW

Counter Clockwise, p. 41

CIPN

Control Interpreted Petri Net, p. 18, 22–24, 45, 78, 101

CSV

Comma Separated Values .csv, p. 79

CW

Clockwise, p. 41

DAOCT

Deterministic Automaton with Outputs and Conditional Transitions, p. vi, vii, 2, 27, 28, 31, 33, 94–95

DES

Discrete Event System, p. 5–7, 27, 35, 101

DOF

Degrees of Freedom, p. 40

FBD

Function Block Diagram, p. 18

IL

Instruction List, p. 102

LCA

Control and Automation Laboratory, p. 35

LD

Ladder Diagram, p. 18, 22, 45, 78–80, 101, 102

NDAAO

Non-Deterministic Autonomous Automaton with Output, p. 94, 96

PLC

Programmable Logic Controller, p. 2–4, 18, 19, 24, 25, 27, 41, 45, 75, 78–80, 84, 88–90, 101

SCADA

Supervisory Control and Data Acquisition, p. 102

SFC

Sequential Function Chart, p. 18

ST

Structured Text, p. 18, 102

UFRJ

Federal University of Rio de Janeiro, p. 35

# List of Symbols

$R$

$R = \{1, 2, \dots, r\}$  Set of path indices, p. 32

$X$

Set of states, p. 32

$X_f$

$X_f \subseteq X$  Set of final states, p. 32

$\Omega$

$\Omega \subset \mathbb{N}_1^{m_i+m_0}$  Set of IO vectors, p. 32

$\Sigma$

Set of events, p. 32

$\lambda$

$\lambda : X \rightarrow \Omega$  State output function, p. 32

$\theta$

$\theta : X \times \Sigma \rightarrow 2^R$  Path estimation function, p. 32

$f$

$f : X \times \Sigma^* \rightarrow X$  Deterministic transition function, p. 32, 94, 96, 113, 114

$x_0$

Initial State, p. 32

# Chapter 1

## Introduction

In a world where the majority of the population lives in industrial societies, and machines take part on the bulk of the production of almost all goods, from food to cosmetics and drugs, from toothbrushes to automobiles, a well-paced throughput is crucial, and any non expected halt on the production or change can be disastrous, producing sometimes multimillionaire debts, provoking a snowball effect, affecting the economy and consequentially the welfare of the society.

A diverse number of causes of the halt or change of the throughput can be accounted for. Some are as simple as a power outage, or a component malfunction, but nowadays there are other players. As the industry walks, or even better runs, towards the so called Fourth Industrial Revolution, it urges the use of *connected sensors*, and since the Internet of Things is the fashion these days, the chances of a hacker attack increases. All these kinds of failures, intended or not can interfere somehow with the production throughput. There are two ways to attempt the reduction of the interference these agents can cause: prevention (before the interference) and remedy (After the interference).

The most common means of prevention in the industry are through preventive maintenance (for the physical components) and cyber security (for the software components).

Once the interference is caused, in order to remedy and reestablish the operation some steps are necessary. Detect the fault, determine the faulty part, and finally intervene. The crucial part of fixing something is to know how it should work<sup>1</sup>, because when we know how it should work, we can distinguish when it is and when it is not.

The focus of this bachelor thesis is system identification aimed for fault detection and diagnosis.

---

<sup>1</sup>“To determine why something has stopped working, it’s useful to know how it was supposed to work in the first place” **DAVIS and HAMSCHER (1988)**

As great part of the manufacture facilities uses discrete sensors and actuators, as conveyor belts, pneumatic cylinders, limit switches and proximity sensors, it is very common to see **PLCs** controlling those plants. And when a system is ruled by discrete events and also its states are discrete it can be modeled by Discrete Event Systems.

On the literature, we can find an expressive number of articles using Discrete Event Systems for identification, fault detection and fault diagnosis. **CABRAL** and **MOREIRA (2017)**; **KLEIN et al. (2005)**; **KUMAR** and **TAKAI (2014)**; **VERAS et al. (2018)** can be used as examples.

This work is based on one of this articles, **MOREIRA** and **LESAGE (2018)**, that develops an algorithm to identify a model of the system using just its inputs and outputs, using a black box approach, also seen in other works as **KLEIN et al. (2005)** and **ROTH et al. (2009)**. This identified model can be later used to detect faults on the system.

The objective is to apply the identification algorithm shown in **MOREIRA** and **LESAGE (2018)** in a Didactic Manufacture System with a strong parallel behavior and a moderate number of inputs and outputs ( over 40) and show that this algorithm can be used on the parallel subsystems, so we can achieve scalability.

During this work all steps from the conception of the control of the system to its identification will be described. So, in order to ease the path throughout this work we have in the next section its outline.

## 1.1 Thesis Outline

### Background

In chapter **2** the minimal background material needed to understand this work is presented: the basic principles of Discrete Event Systems, two ways of modeling them (Automata and Petri Nets), how to use Petri Nets to design a controlled system and how to translate this control to Ladder Logic (as presented in **MOREIRA** and **BASILIO (2013)**), in order to be programmed on a PLC, then finally we show the identification algorithm and the identified model **DAOCT** (as presented in **MOREIRA** and **LESAGE (2018)**).

### System

In chapter **3** we present the system that is going to be identified, the devices that compose it, its sensors and actuators and some reference points to simplify the understanding of the described movements on the following sections and chapters.

## **Control**

In chapter 4 we describe the design process of the control and its implementation on the PLCs.

## **Identification**

In chapter 5 we describe the process of identification of the system, logging of its input and output data and using the algorithm shown in chapter 2 to identify the model.

## **Results**

In chapter 6, the data acquired on the steps described on chapter 5 is discussed and fed to the algorithm, then the output models are discussed and the behavior of the system is addressed.

## **Conclusion**

In chapter 7, this thesis is concluded, with some final remarks about the work done. The drawbacks presented during the implementation are collected, and other approaches on some specific parts of this work are proposed as a sequel of this work.

# Chapter 2

## Background

This chapter will discuss the main topics needed to understand this work, from discrete event systems to discrete control implementation on [PLCs](#), a more detailed explanation of each topic can be found on the respective cited work.

### 2.1 Systems

A System as defined by the Cambridge's dictionary is “a set of connected things or devices that operate together”. As seen two basic properties of systems are :

- they are formed by grouping smaller parts
- the smaller parts when grouped work together to carry out a specific function

As its definition is so abstract almost anything can be defined as a system, physical or not, beings can be defined as systems and even economic mechanisms can also be considered as systems.

Usually systems are modelled by a Input/Output process. The system is fed with a set of inputs, it process the inputs resulting on the output set, as we can see in [Figure 2.1](#).

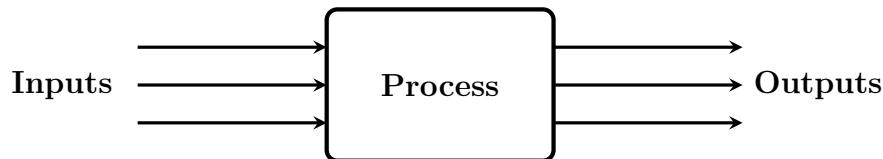


Figure 2.1: Input/Output Process model

In some systems, its inputs and outputs can't represent it's behaviour, so the concept of state is created, and it represents the behaviour of the system in a given instant  $t$ .

The states can be continuous or discrete, and the systems which these states represent can be considered as Continuous Systems, Discrete Systems or even Hybrid Systems, which combine both kind of states.

The systems modelled in this work are Discrete Systems, more details about other kinds of systems as well as examples and their analysis can be found on [OPPENHEIM et al. \(1996\)](#) and [KALOUPTSIDIS \(1997\)](#).

## 2.2 Discrete Event Systems

Discrete Systems can be driven by time and by events. It means, the states can be changed continuously by the time or instantaneously by some ensemble of events.

In this thesis we are interested in the event-driven type. Some basic mathematical formalisms, nomenclature and representations can be developed to facilitate the understanding. Some of those will be presented in the following sections based on [CAS-SANDRAS and LAFORTUNE \(2009\)](#); [DAVID and ALLA \(1989, 2005\)](#).

## 2.3 Languages

A language can be defined by the Merriam-Webster's dictionary as "a systematic means of communicating ideas or feelings by the use of conventionalized signs, sounds, gestures, or marks having understood meanings" And as it is defined by this dictionary entry we pursue to communicate the complete behaviour of the DES. Firstly we need to define a group, or set of marks to characterise the singular behaviour of the system. So, we define a set  $\Sigma$ . This set contains all elements which combined can create a language. Again in analogy with linguistics, each one of these marks, the events can be compared to letters , provided that  $\Sigma$  can be called an "alphabet", and the combination of its events "words". Words are also called "strings " or even "traces". Considering the use of the word "string" as the variable type used on several programming languages used in this work, we prefer the use of the vocables "word" and "trace". We can also define a mark to represent an empty word,  $\epsilon$ , that is, a word that is not formed by any event.

The combination process to form words is called concatenation. For instance, given two events  $a$  and  $b$ , the words  $ab$  and  $ba$  can be created concatenating these two events and there is no particular reason to suppose that  $ab$  is equal to  $ba$ , the same way the words "ten" and "net" have different meanings in English.

We can also concatenate two words, to create a different one, we can take the words

*ab* and *ba* and create words like *abba* and *baab*.

As we extended the definition of concatenation to words, we define  $\epsilon$ , the empty word, as the identity element of concatenation:  $w\epsilon = \epsilon w = w$  for any word  $w$ .

Likewise, we can define the length of a word as the number of events contained by this word, we denote the length with two vertical bars, given a word  $w$  its length is equal to  $|w|$  and by definition  $|\epsilon| = 0$ .

As we know, there is a great number of human western languages, as portuguese, english, french, spanish etc, that roughly are formed by the same alphabet, but overall they are formed by different combination of words. Similar things can happen with languages that define the DESs, so we can define as in CASSANDRAS and LAFORTUNE (2009).

### Definition 2.1 (Language)

*A Language defined over an alphabet  $\Sigma$  is formed from finite-length words generated from the concatenation of the events in  $\Sigma$  and  $\epsilon$ .*

Take for example an alphabet  $\Sigma = \{a, b, g\}$ , we can define different languages

$$L_1 = \{\epsilon, a, abb\}$$

$$L_2 = \{\text{all possible words of length 3 starting with } g\}$$

$$L_3 = \{\text{all possible words starting with } g\}$$

The cardinality of this sets are  $|L_1| = 3$ ,  $|L_2| = 9$ ,  $|L_3| = \infty$ . As we can see from the same alphabet very different languages can be created, thus we can define a way to encapsulate all possible languages generated from the same alphabet  $\Sigma$ . Let us denote by  $\Sigma^*$  the set containing all finite words composed with the elements of  $\Sigma$  and  $\epsilon$ . The  $*$  operation is called the *Kleene-closure*. Similarly to  $L_3$  it is countably infinite since it contains arbitrarily long words. For instance the *Kleene-closure* of the alphabet  $\Sigma = \{a, b, c\}$  is:

$$\Sigma^* = \{\epsilon, a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc, aaa, \dots\}$$

There are a few operations with languages and alphabets that can be defined, but they are outside the scope of this work, they can be found on CASSANDRAS and LAFORTUNE (2009).

## 2.4 Representation of Languages

Although languages can describe the behaviour of DESs, there are cases, as the one shown by the language  $L_3$  in the last section, in which the language is enormous, in that case countably infinite, what makes them not so simple to communicate the behaviour of the system. For this purpose, there are some other formalisms that aid the comprehension, since they can be a more compact way of expressing the system's behaviour or accompanied by diagrams.

In the following subsections two of the most known representations will be presented: Automata and Petri Nets.

### 2.4.1 Automata

One of the most known representation of languages are automata. The notion of automaton is basically the definition of DESs, as we saw in the section 2.2: a set of events can change the state of the system. If we know all the events composing the language of the system and its states, we can have its alphabet  $\Sigma$  and we can create a set  $X$  composed by all states. From  $\Sigma$  and  $X$  we can derive a function that represents the transition from a state to other, this function is called *transition function* of the automaton denoted as  $f : X \times \Sigma \rightarrow X$ . For example if a system have an alphabet  $\Sigma = \{a, b\}$  and 2 states, we can name the states  $x$  and  $y$ , and then create the set  $X = \{x, y\}$ . Knowing that the system begins at state  $x$  and that when event  $a$  happens it changes to state  $y$  we can create a function  $f(x, a)$  and define it as  $y$ . Likewise if we know that when the system is at state  $y$  and event  $b$  happens, a function  $f(y, b)$  can be defined as  $a$ .

As a visual aid, a representation of these functions can be made through a diagram, called *state transition diagram*. In this kind of diagram the states are represented by circles labeled with their names, and the functions as arcs labeled with the corresponding event, connecting two states, with arrows in one of their extremities indicating the transition from a state to other. The initial state of the automaton has an arc pointing towards it coming from no other state. Figure 2.2 can represent the functions  $f(x, a)$  and  $f(y, b)$  described in the last paragraph.

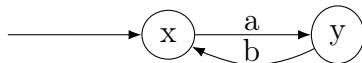


Figure 2.2: State Transition Diagram

Now, for a more complex example, from CASSANDRAS and LAFORTUNE (2009):

### Example 2.1 (Simple Automaton)

Given  $\Sigma = \{a, b, g\}$ ,  $X = \{x, y, z\}$  and the following transition functions:

$$\begin{array}{ll} f(x, a) = x & f(x, g) = z \\ f(y, a) = x & f(y, b) = y \\ f(z, b) = z & f(z, a) = f(z, g) = y \end{array}$$

We can represent this automaton with the diagram on Figure 2.3

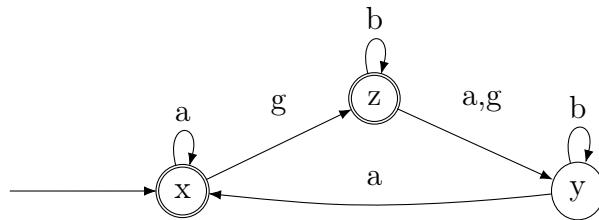


Figure 2.3: Diagram representing the automaton from example 2.1

We can also mark states that have some special meaning, a final state for instance. In this work, as in CASSANDRAS and LAFORTUNE (2009) they are going to be identified by double circles.

Now a deterministic Automaton can be defined:

### Definition 2.2 (Deterministic Automaton)

A Deterministic Automaton, denoted by  $G$ , is a five-tuple

$$G = (X, \Sigma, f, x_0, X_m)$$

where:

$X$  is the set of **states**

$\Sigma$  is the finite set of **events** associated with  $G$

$f : X \times \Sigma \rightarrow X$  is the **transition function**

$x_0$  is the **initial state**

$X_m \subseteq X$  is the set of **marked states**

Other kinds of automata and operations between automata exist but are not going to be used in this work, again CASSANDRAS and LAFORTUNE (2009) present them.

## 2.4.2 Petri Nets

Another kind of representation of languages are Petri Nets, whose concept was created by C.A.Petri in the early 1960's. Differently from the automata representation that are basically formed from states, Petri nets are bipartite graphs, formed by nodes called *places* and *transitions*. Transitions represent the events that drive the system, and places represent the conditions for these events to happen. The mechanism to represent the fulfilment of the conditions is named marking. A petri net is built over three basic concepts, the petri net graph/structure, its marking and firing transitions. The next subsections will be based on [DAVID and ALLA \(2005\)](#) and [CASSANDRAS and LAFORTUNE \(2009\)](#).

### Petri Net Graph

Similarly, arcs are used to connect the nodes and have arrowheads to identify the direction, but differently, all arcs must have exclusively one node at each end, that means no arc is used to identify the initial state of a petri net. As said, a petri net is bipartite graph, that means places can only connect to transitions and vice versa. In this work as in [DAVID and ALLA \(2005\)](#) places will be represented by circles and transitions by bars.



Figure 2.4: Component nodes of a petri net.

The same way a function was created to define the transitions of states in an automaton, two functions will be created to define the connections between places and transitions. First we need to define the sets of places and transitions.  $P$  is the set of places and  $T$  the set of transitions. With this two sets we can then define those functions. The first one represents the arcs from places to transitions, and is denoted as  $Pre : P \times T \rightarrow \{0, 1\}$ , the second one the arcs that connects transitions to places, denoted as  $Post : P \times T \rightarrow \{0, 1\}$ . The value 1 is attributed to arcs that exist and 0 to the nonexistent ones.

### Example 2.2 (Simple Petri Net structure)

Given  $P = \{p_0, p_1\}$ ,  $T = \{t_0, t_1, t_2\}$  and the following transition functions:

$$\begin{array}{llll} Pre(p_0, t_0) = 0 & Post(p_0, t_1) = 0 & Pre(p_1, t_0) = 0 & Post(p_1, t_1) = 1 \\ Post(p_0, t_0) = 0 & Pre(p_0, t_2) = 0 & Post(p_1, t_0) = 0 & Pre(p_1, t_2) = 0 \\ Pre(p_0, t_1) = 0 & Post(p_0, t_2) = 0 & Pre(p_1, t_1) = 0 & Post(p_1, t_2) = 0 \end{array}$$

We can represent this petri net structure with the diagram on Figure 2.5

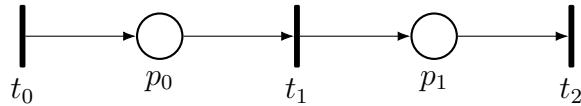


Figure 2.5: Diagram representing the petri net structure from example 2.2

A drawback from the definition of this functions, is that is not possible to have more than an arc linking two nodes, so we can generalize them to any natural number:

$$Pre : P \times T \rightarrow \mathbb{N}_0$$

$$Post : P \times T \rightarrow \mathbb{N}_0$$

with this new definition we can change the definition of  $Post(p_1, t_1)$  from 0 to 2 resulting on the following petri net structure.

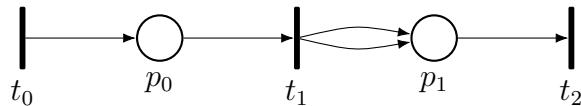


Figure 2.6: Diagram representing the petri net structure from example 2.2, but with  $Post(p_1, t_1) = 2$

In order to reduce the number of arcs in a diagram, usually only one arc is drawn and a label is added with the value of its respective function, if it is greater than 1, Figure 2.7 illustrates it:

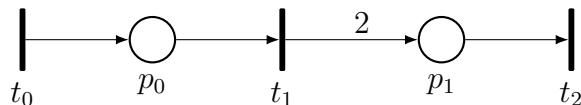


Figure 2.7: Same diagram as Figure 2.6 but with labeled arcs.

## Marking

As said in the beginning of this subsection marking is used as the mechanism to represent if the condition of occurrence of a determined event is met or not. But also it can be used to represent the state of the system. The mechanism works as follows. Tokens can be designated to places and the way the tokens are distributed among places is called the marking of a petri net graph. We can define a marking function  $x : P \rightarrow \mathbb{N}$  that denotes the number of tokens in a determined place. In this work, as in the majority of articles and books, the tokens will be represented as black dots inside the places.

The Figures 2.8a and 2.8b show an unmarked and a marked petri net graph.

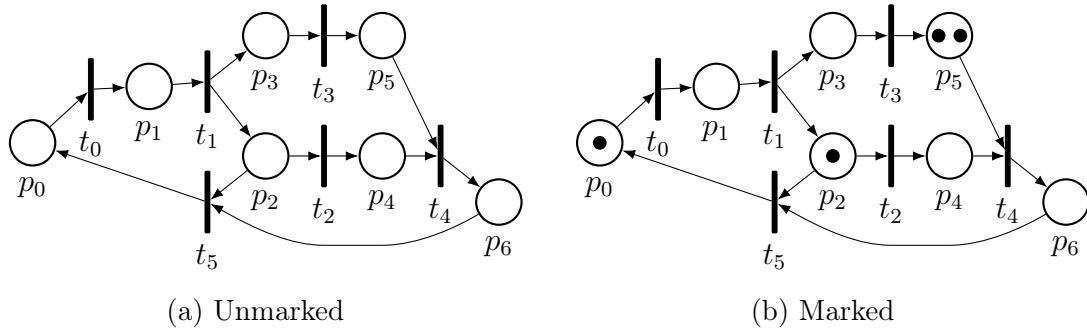


Figure 2.8: Example of unmarked and marked petri net graphs.

The marking of a petri net, can be represented as a vector of the function  $x$  applied on all places, for example the marking of the Figure 2.8b is the following vector  $\mathbf{x}$

$$\mathbf{x} = \begin{bmatrix} x(p_0) \\ x(p_1) \\ x(p_2) \\ x(p_3) \\ x(p_4) \\ x(p_5) \\ x(p_6) \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 2 \\ 0 \end{bmatrix}$$

This vector  $\mathbf{x}$ , the marking of the petri net, can be identified as the state of the petri net. So, different configurations of tokens mean different states of the system, now we only need a way to change from a state to other, in other words, move the tokens.

## Firing Transitions

The mean of move the tokens is firing transitions. When an event happens and the corresponding transition is enabled, this transition is fired and tokens are moved between places. We can define the functions  $I : T \rightarrow 2^P$  and  $O : T \rightarrow 2^P$  that describe the set of places considered as inputs and outputs of a transition:

$$I(t_j) = \{p \in P : Pre(p, t_j) > 0\}$$

$$O(t_j) = \{p \in P : Post(p, t_j) > 0\}$$

### Definition 2.3 (Enabled transition)

A transition is enabled if

$$x(p_i) \geq Pre(p_i, t_j) \text{ for all } p_i \in I(t_j)$$

if  $I(t_j) = \emptyset$ ,  $t_j$  is always enabled.

And we can define the dynamic of the petri net, how the tokens move:

### Definition 2.4 (Petri net dynamics)

It is possible to define a state transition function,  $f : \mathbb{N}^n \times T \rightarrow \mathbb{N}^n$ , where  $n$  is the length of the state vector  $\mathbf{x}$ . This function  $f$  is defined for a transition  $t_j \in T$  if and only if this transition is enabled. If  $f(\mathbf{x}, t_j)$  is defined, then we create a new state vector  $\mathbf{x}'$ :

$$x'(p_i) = x(p_i) - Pre(p_i, t_j) + Post(p_i, t_j), i = 1, \dots, n$$

As an example we can take Figure 2.9:

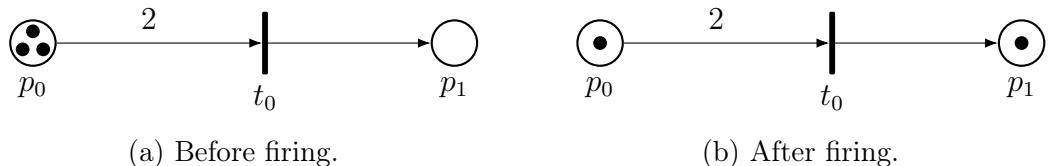


Figure 2.9: Example of petri net Dynamic.

The state before firing transition  $t_0$  is  $\mathbf{x} = [3 \ 0]^T$  and as we see  $Pre(p_0, t_0) = 2$  and  $Post(p_1, t_0) = 1$  so applying the petri net dynamic we can find the next state  $\mathbf{x}' = [1 \ 1]^T$

Once all these fundamentals are presented we can finally define a Petri Net

### Definition 2.5 (Petri net)

A Petri net is defined as a five-tuple

$$PN = (P, T, Pre, Post, \mathbf{x}_0)$$

where:

$P$  is the set of **places**

$T$  is the set of **transitions**

$Pre$  is the **input incidence** function

$Post$  is the **output incidence** function

$\mathbf{x}_0$  is the initial marking of the net

And its dynamic is ruled by the state transition function  $f$  defined at [2.9](#).

To make the connection between the Petri net and the events of the system, called  $\Sigma$ , the alphabet, we can define a labeling function,  $l : T^* \rightarrow \Sigma^*$  that makes the link between a sequence of firing transitions and a sequence of events. But each transition can only have one respective event.

### Definition 2.6 (Labeled Petri net)

A Labeled Petri net is defined as a seven-tuple

$$PN = (P, T, Pre, Post, \mathbf{x}_0, \Sigma, l)$$

where:

$(P, T, Pre, Post, \mathbf{x}_0)$  is a Petri Net

$\Sigma$  is the set of **events**

$l$  is the **labeling** function

Usually the events are represented in the petri net graph over its respective transition as shown in the [Figure 2.10](#). This system has an alphabet  $\Sigma = \{a, b\}$  and labeling functions  $l(t_0) = a$  and  $l(t_1) = b$ .

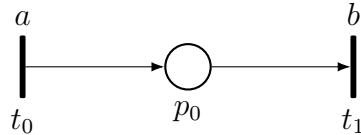


Figure 2.10: Labeled Petri net.

## 2.5 Control Interpreted Petri Nets

One of the greatest uses of Petri Nets, besides modeling a system, is its ability to model the control of a system. For this intent we use Control Interpreted Petri nets. It is an extension from the labeled Petri net, we add actions for places, so it is possible to change the outputs of the system, conditions to the transitions, so it is possible to change the state of the control based on the inputs of the system, and even the ability to delay the firing transitions based on time. Another artifice used is an inhibitor arc, that prevents the firing of a transition based on the marking of the corresponding place.

### Definition 2.7 (Control Interpreted Petri net)

A Labeled Petri net is defined as a seven-tuple

$$PN = (P, T, Pre, Post, \mathbf{x}_0, In, \Sigma, C, l_C, D, L_D, A, I_A)$$

where:

$(P, T, Pre, Post, \mathbf{x}_0)$  is a Petri Net

$In$  is the **inhibitor arc** function that prevents the enablement of transitions

$\Sigma$  is the set of **events** associated to transitions

$C$  is the set of **conditions** associated to transitions

$l_C$  is the **labeling** function that associates a transition with events and conditions from  $\Sigma$  and  $C$

$D$  is the set of **delays** associated to transitions

$l_D$  is the **labeling** function that associates a transition with a delay from  $D$

$A$  is the set of **actions** associated to places

$l_A$  is the **labeling** function that assigns actions from  $A$  to a place

The definition of  $In : (P \times T) \rightarrow \mathbb{N}$  is that a transition  $t_j$  is inhibited if  $x(p_i) \geq In(p_i, t_j)$ . Inhibitor arcs are not used in this work but usually they are represented with an arc with a circle in one of its ends, as shown in Figure 2.11.

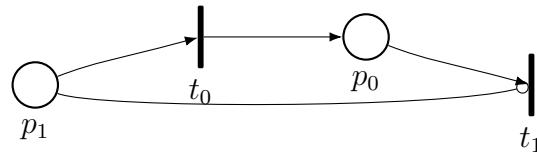
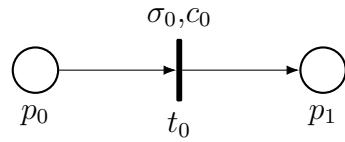


Figure 2.11: Example of Petri net with inhibitor arc.

As we can see from the definition there are two labeling functions to connect transitions,  $l_C$  and  $l_D$ . The  $l_c$  is defined for transitions with no firing delay and  $l_D$  for transitions with firing delay.

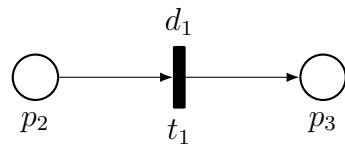
The labeling function  $l_C : T^0 \rightarrow (\Sigma \times C)$  defines a pair of event and boolean condition from  $\Sigma$  and  $C$  respectively. A transition  $t_i$  belonging to  $T^0$  (a subset of  $T$  that represents the transitions with no time delay) has a corresponding event, condition tuple  $(\sigma_i, c_i)$ . For example, take a transition  $t_0$ ,  $\Sigma = \{\sigma_0\}$  and  $C = \{c_0\}$ . If a function  $l_c(t_0) = (\sigma_0, c_0)$  is defined, this transition will be fired when the condition  $c_0$  is true and the event  $\sigma_0$  happens, but obviously, if and only if this transition is enabled and not inhibited. The transition  $t_0$  is represented graphically as shown in [Figure 2.12](#)



[Figure 2.12: Representation of new labeling function](#)

If the event is missing from the representation of the transition, it is equal to the  $\lambda$ , the always occurring event. And if the condition is missing, that means it is equal to 1, it is always *true*. If both are missing, that means the transition will be automatically executed if it is enabled.

By the other hand, the labeling function  $l_D : T^D \rightarrow D$ , defines a delay for the transition to be fired. A timed transition  $t_i$ , a transition in  $T^D$  (a subset of  $T$  that represents the transitions with a time delay), has a corresponding delay  $d_i$ . As an example, take a timed transition  $t_1$  and  $D = \{d_1\}$ , after the enablement of the transition, it takes  $d_1$  time units in order to be fired. In this work timed transitions are represented as bars slightly larger than normal conditions. An example of this representation we can see [Figure 2.13](#)



[Figure 2.13: Representation of a timed transition.](#)

Another labeling function that was created is  $l_A : P \rightarrow 2^A$ , assigning a set of actions belonging to  $A$  to a place. Actions can be impulse actions or continuous. A continuous

action happens always that the marking of a place is greater than 0,  $x(p_i) > 0$ , an impulse action, on the other hand, happens only when the marking of the place changes from 0 to a value greater than 0. Actions are represented graphically as labels in places. Impulse actions are differed by a star (\*) at its end. So an action  $F$  is continuous and  $B^*$  is an impulse action. Figure 2.14 show a representation of a Place with both kinds of actions.

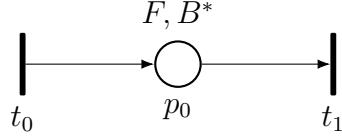


Figure 2.14: Representation of labeling of Actions.

Although these representations exist, in this work events, conditions and actions labels are suppressed from the diagrams and tables accompany the drawings showing the meaning of the transitions (firing events and conditions) and places (Actions). This choice was made because when the controlled interpreted Petri nets are very large as the ones shown in the next chapters, if the events, conditions and actions labels are long they can increase the size of the diagram.

To illustrate this better we give an example based on one example from DAVID and ALLA (1989).

### Example 2.3 (Loading of a wagon)

We consider the system represented by the scheme in Figure 2.15. A wagon can be moved between the points  $a$  and  $b$ , using the inputs  $L$  and  $R$  (moving it to the left or right, respectively). At point  $a$  there is a button  $m$  that can be pressed by an operator and a limit switch called  $a$  that is activated when the wagon is on the left. At point  $b$ , an homonym limit switch is placed and activated when the wagon is on the right. There is a hopper that can be opened when the input  $Open$  is turned on and closed when not. If it is opened its content is poured. There is also a button  $p$  that is activated when the weight applied over the plate is equal or greater to the weight of a full wagon.

The objective of the control is, when the wagon is in its leftmost position and the button  $m$  is pressed, it moves to the right, stops at  $b$ , the hopper is opened and the wagon is loaded, when it is completely full it moves to the left and it stops at  $a$  waiting to be unloaded and for a next press of  $m$  to recommence the loop.

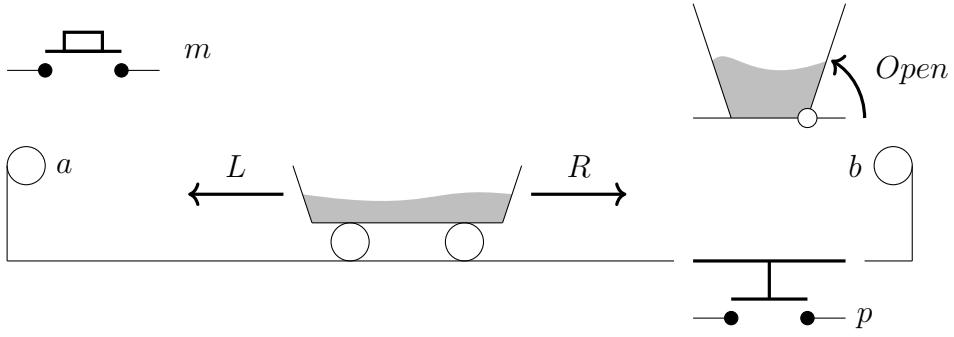


Figure 2.15: Example of System to be controlled by the Petri Net

From the description of the control it is possible to create a Control Interpret Petri Net to describe it, as the one in [Figure 2.16](#)

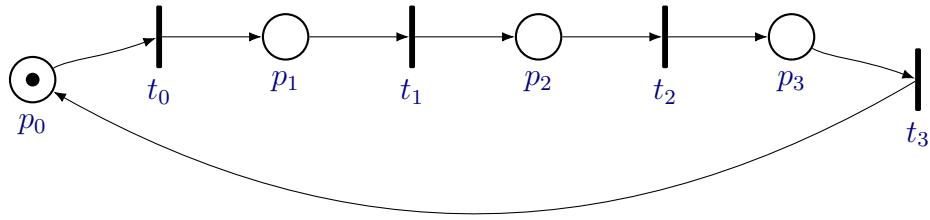


Figure 2.16: Example of Control Interpreted Petri Net to control system in [Figure 2.15](#)

The meaning/description of each place and transition is given by the following tables:

Table 2.1: Control Interpreted Petri Net Example Places.

Places	Meaning
$p_0$	System Stopped
$p_1$	R (Car Moving to the Right)
$p_2$	Open (Container Opened)
$p_3$	L (Car Moving to the Left)

Table 2.2: Control Interpreted Petri Net Example Transitions.

Transitions	Meaning
$t_0$	$\uparrow m$ (filling request)
$t_1$	$\uparrow b$ (Right Limit Switch)
$t_2$	$\uparrow p$ (Car is Full)
$t_3$	$\uparrow a$ (Left Limit Switch)

In this work as in the usual boolean notation, when just the name of a variable is given in a table it means the variable is equal to true, and when there is a bar in its top it is equal to false, so they determine conditions. E.g.:  $b$  and  $\bar{b}$ . And when a variable is preceded by  $\uparrow$  and  $\downarrow$ , they determine events corresponding to its raising and falling edge.

## 2.6 Implementation of Control Interpreted Petri Nets

Once the control of a system is modeled by a [CIPN](#), it is needed to implement the control in a real controller. The most used controllers in the industry are [PLCs](#). The international standard IEC 61131, defines all the standards for [PLCs](#), and its third part (IEC 61131-3) defines five languages to program [PLCs](#): [LD](#), [Function Block Diagram \(FBD\)](#), [Structured Text \(ST\)](#), [IL](#) and [Sequential Function Chart \(SFC\)](#). One of the most used in the industry is [LD](#), because of its resemblance with electric connections. So we are going to use [LD](#) to implement the control designed with the [CIPN](#).

### 2.6.1 Ladder Logic

The ladder logic is based on two components, contacts and coils. Their terminals are interconnected to transmit boolean signals. This connection is similar to the ladder physical implementation, from where came its inspiration, the components were connect to boards and they formed electric circuits, turning on and off motors and other actuators, based on the combination of its inputs. The name Ladder comes from the resemblance between its structure (circuits formed in parallel one above the other) and a ladder, so each circuit is called a Rung by analogy. The logic values in a [LD](#) rung are transmitted from the left to the right of the diagram. The components let the logic “current” flow from its left terminal to the right terminal depending on some conditions, and these conditions vary from component to component. The rungs are executed one by one and once the very last rung is executed, the first rung is re-executed, thus creating an infinite loop. The graphical representation of the most used types of contacts and coils can be seen in Figures [2.17](#) and [2.18](#)

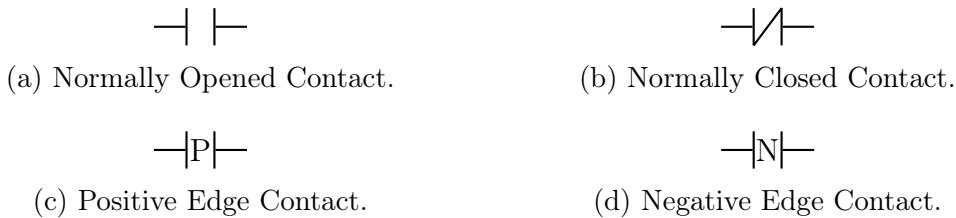


Figure 2.17: Types of Contacts.

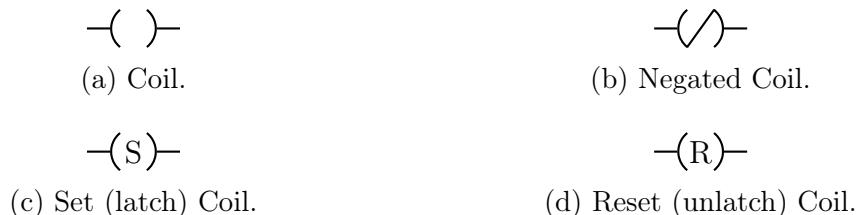


Figure 2.18: Types of Coils.

## Contacts

Contacts represent the conditions of the ladder logic depending on inputs. These inputs can be any variable in a **PLC**, an external input ( sensors of the system to be controlled), a variable stored in memory or the current value sent to an output from the **PLC**. A normally opened contact activates its right terminal (set it to *true*) if the logic value in its left terminal is *true* and its corresponding input is equal to *true*. A normally closed contact activates its right terminal if the logic value in its left terminal is *true* and its corresponding input is equal to *false*. The Positive Edge contact activates its right terminal only in the instant that its input change from logic value *false* to *true*, if the logic value in its left terminal is *true*. And the Negative Edge contact activates its right terminal only in the instant that its input change from logic value *true* to *false*.

As we can see, positive and negative contacts can be used to represent raising ( $\uparrow$ ) and falling edge ( $\downarrow$ ) events and normally opened and closed contacts to represent conditions (and their negation).

## Coils

Coils, by the other side represent the actuation in outputs. These outputs can be a variable stored in memory or the outputs of the controller (actuators of the system to be controlled, for instance). A coil sets its output variable to *true* if the logic value of its left terminal is *true*, and sets the output to *false* otherwise. A negated coil does

the exact opposite, sets the output value to true if the logic value of its left terminal is *false* and sets it to *true* if the logic is *true*.

A set coil (or latch) sets its output variable to *true* if the logic value of its terminal is *true* and it remains *true* until the variable is reset. And a reset coil (or unlatch) sets its output variable to *false* if the logic value of its terminal is *true* and it remains *false* until the variable is set.

## Combinational Logic

In boolean logic, in order to show functional completeness, it is need to show a complete set of connectives ( a set that can create all other logic connectives as a combination of its elements ). A well-know complete set is  $S = \{AND, NOT\}$ , binary conjunction and negation. To show that the ladder logic is functional complete we need only to present how to construct this two connectors in it. The conjunction of two inputs, can be made using two contacts in series, as shown in [Figure 2.19](#)

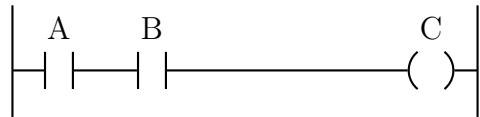


Figure 2.19: And logic in a Ladder rung.

In this case C will only be activated if A and B are equal to *true*. ( $C = AB$ )

The negation of a variable can be achieved by the use of a normally closed contact ([Figure 2.20](#)).

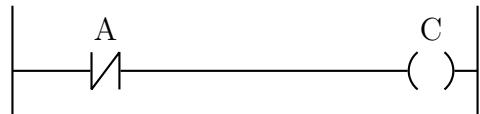


Figure 2.20: Not logic in a Ladder rung.

C will only be activated if A is *false*. ( $C = \bar{A}$ )

Although all logic connectives can be constructed with this two connectors, the OR connector can be achieved by the use contacts in parallel ([Figure 2.21](#)).

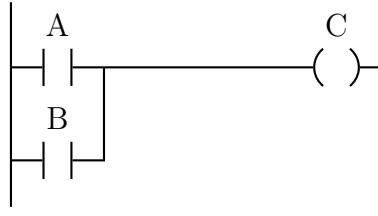


Figure 2.21: Or logic in a Ladder rung.

### Function Blocks and extensions

In order to increase functionality some function blocks and extensions to contacts were created. We can see examples of these blocks and contacts in the next figure:

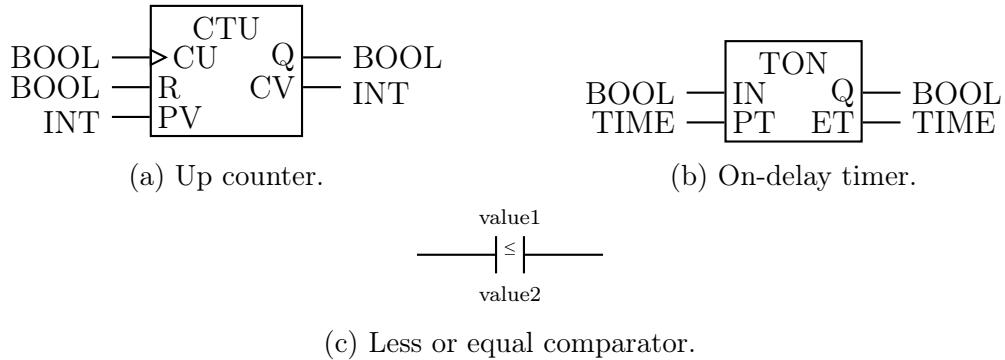


Figure 2.22: Examples of function blocks.

Up counters (Figure 2.22a) save the value of a counter in a *CV* variable. Every raising edge on input *CU* it increments *CV* value. If *CV* = *PV*, the logic value of output *Q* is set to 0. When the input *R* is true *CV* value is set to 0 and the output *Q* set to false.

On-delay timers (Figure 2.22a) set a timer when input *IN* is *true* and save it to *ET*. If *ET* = *PT*, the logic value of output *Q* is set to *true*. But if, meanwhile the counting, the value of *IN* returns to *false*, *ET* is reset to 0.

Comparator contacts as the less or equal comparator in Figure 2.22c, work similarly to contacts, but instead of an input as a condition, there are two inputs (*value1* and *value2*) and the condition is a comparison between both of them. In this case, the contact is activated once its left terminals' logic value is *true* and *value1*  $\leq$  *value2*.

Other blocks and functions can be found in the IEC 61131-3, as adders, subtractors, communication blocks etc.

## 2.6.2 Conversion from CIPN to LD

A simple method of conversion from [CIPN](#) to [LD](#) is presented in [MOREIRA and BASILIO \(2013\)](#).

It consists in dividing the [CIPN](#) in 4 modules:

1. A module of external events

To create conditions to the firing of transitions based on external events (inputs)

2. A module of firing conditions

To indicate what condition will be fired using the *Pre*, and *In* functions, the conditions found on the last module and time delays (if it is a timed transition )

3. A module of Petri Net dynamics

Uses the *Pre* and *Post* functions to determine the tokens “motion”

4. A module of actions

Determines the places where each action is performed.

In this work, the external events and firing conditions was combined in order to reduce the size of the program. But every module will be described as in [MOREIRA and BASILIO \(2013\)](#).

### External events

As external events are associated with positive and negative edge of the inputs of the system, in this module, positive and negative edge contacts are used to detect the rising and falling edge events, and they are stored in variables using coils, a variable is created for every event. For visibility’s and organisation’s sake a rung is used for each event, resulting  $|\Sigma|$  rungs.

### Firing Conditions

As said in [2.5](#), for a transition  $t_j$  to be fired, first it needs to be enabled ( $x(p_i) \geq Pre(p_i, t_j)$  for all  $p_i \in I(t_j)$ ), not inhibited ( $x(p_i) < In(p_i, t_j)$  for all  $p_i \in I(t_j)$ ) and the conditions and events  $\sigma_j c_j$  are met or the delay  $d_j$  is elapsed, depending on the kind of transition. As places can have multiple tokens, we can use *int* variables to store the number of tokens, and comparator contacts to determine if the transitions are enabled and not inhibited. When a place can only bear at most one single token for all markings of the Petri net, a *bool* variable can be used to store the number of tokens, in this case a

single normally open contact can be used to determine if there is a token in that place. The time delays are implemented using on-delay timers. The state of fulfilment of the conditions is stored in variables, one for each transition. Similarly, for organisation's sake a rung is used for each transition, resulting  $|T|$  rungs.

## Petri Net Dynamics

In this module the dynamic of the tokens is implemented. If the condition for the firing of a transition is fulfilled (represented by normally open contacts), adders and subtractors can be used to represent the movement of tokens, increasing and decreasing the values from the *int* variables that represent the marking of each place, if their capacity is greater than one, if not they are represented by boolean variables, and instead Set and Reset coils can be used to represent a token entering a place and a token exiting another, respectively. Again, for organisation's sake a rung is associated for each transition, resulting  $|T|$  rungs.

## Actions

In the Action module, we use coils to act on the outputs. Depending on the type of action and the logic of the control, set/reset coils or normal coils can be used. The condition to activate/deactivate the output is the presence of a token in the places where the action is performed, this can be achieved by comparing the numbers of tokens in a place. If the tokens of a place is represented by an *int* variable, we use greater or equal comparators, but if it is represented by a *bool* variable, a normally opened contact is enough. If the action is an impulse action we can put a positive edge in series with the places contact or the comparator.

Differently from the original article, we will use one rung per each action in A, resulting in  $|A|$  rungs. This is made because in [RENAULT \(2017\)](#) it is recommended to perform an action in only one rung, and group the conditions using OR connectors. In the industry this is important in order to reduce errors and to ease the debugging process.

## Example

An example of this conversion can be given using the same [CIPN](#) from example 2.3. As said, the external events and firing condition modules are grouped in same module in this work. The converted Ladder Logic can be seen in [Figure 2.23](#).

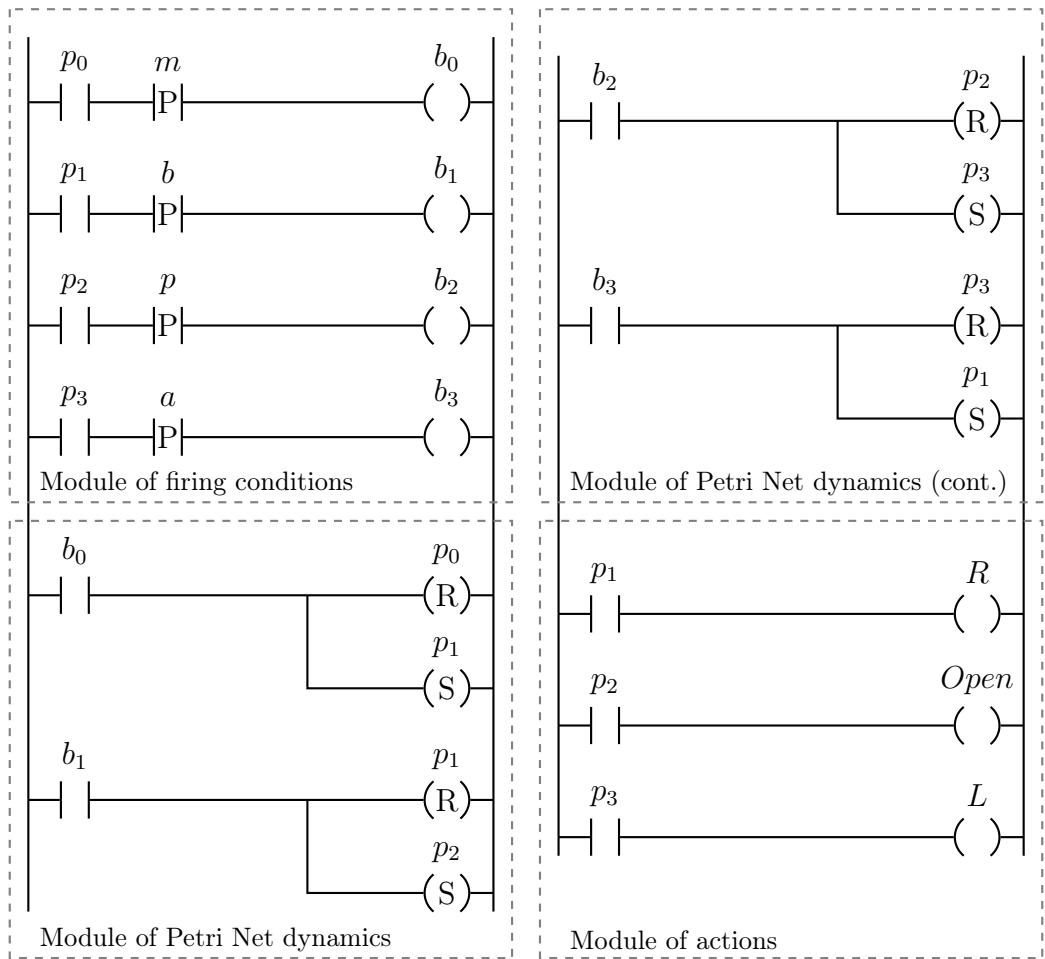


Figure 2.23: Example of Control Interpreted Petri Net converted to Ladder.

### 2.6.3 Petri Net divided in multiple PLCs

A problem that can occur, is the case where a **CIPN** must be divided in multiple **PLCs**, because of how the assembling of the plant was (some inputs/outputs are only connected to a single **PLC**, and other inputs/outputs to another **PLC**), this division can be arbitrarily decided by the person who will implement the control.

In Figure 2.24, it is shown an example of a **CIPN** divided between 2 **PLCs**. As we can see, in Figures 2.24a and 2.24b there are dotted transitions and places. In this work, we will represent as dotted, transitions and places that are part of another section of the Petri Net. They are not represented in the same figure, but the arcs show the connection between the sections of the net, showing that when connected they form a complete Petri net. In the digital form<sup>1</sup> of this thesis it is possible to travel between

<sup>1</sup> Available at: <https://github.com/Accacio/docsTCC/raw/master/monografia.pdf>

figures that are not in the same page just by clicking in the name of the corresponding dotted place/transition.

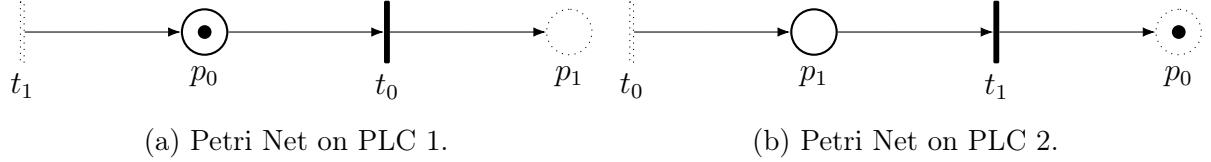


Figure 2.24: Example of Petri Net divided between 2 PLCs.

In order to solve the problem of communication caused by the division, there are different ways, one of them is the method shown by [ANTUNES FLORIANO \(2019\)](#), where different sections are synchronised in a distributed manner using common places. In this work, a master/slave approach was used. CLP1 was considered as the master and CLP2 as a slave. For the master, it is created another module called “Data Sending/Receiving”, divided in 2 parts, the first that happens before all other modules, with the objective of getting all needed variables from other [PLCs](#), and another part, that happens after all other modules, with the objective of sending variables to all other [PLCs](#). By the other side, the slaves have 2 modules, one at the beginning, called “Prepare Received Data” and another at the very end called “Prepare Data to Send”, in which the data received from the master is prepared to be used and the data is prepared to be sent to master.

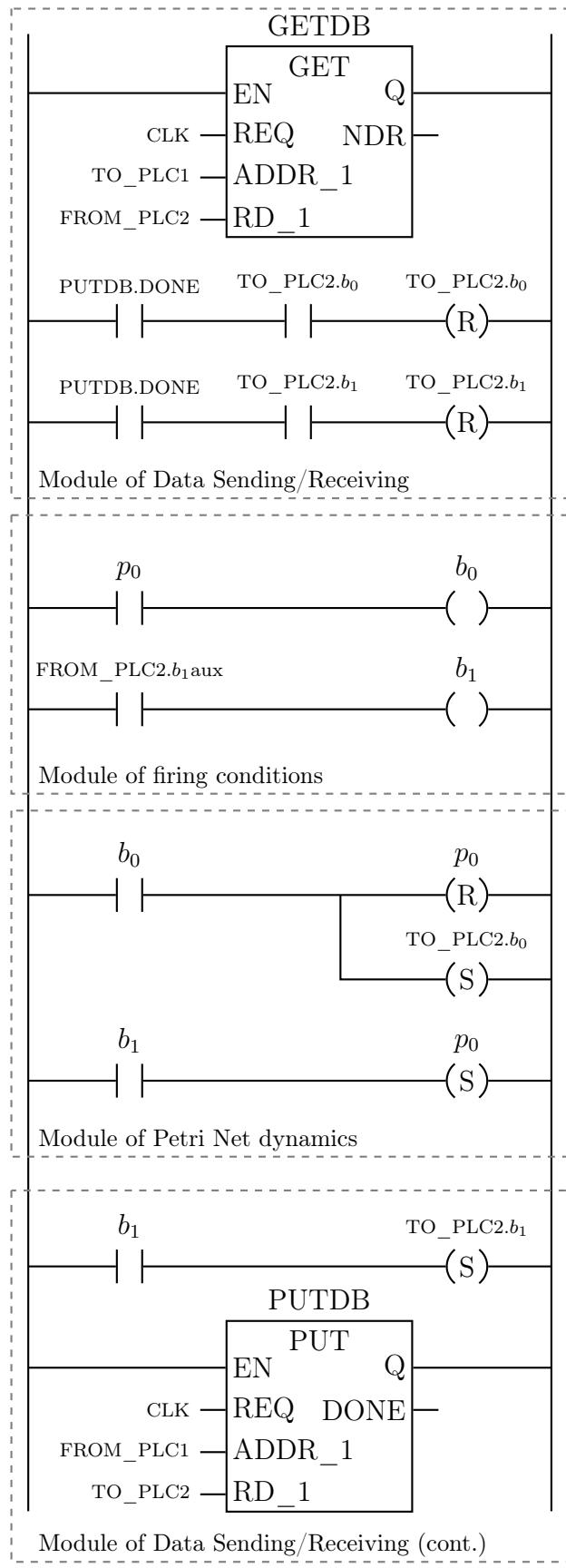
The communication between [PLCs](#) can be made using Profinet protocol, and if we take for instance Siemens [PLCs](#), they have two function blocks called “Get” and “Put”, that are used to establish data transfer between two [PLCs](#) using the Profinet protocol. Tutorials on how to configure this blocks can be found on ([ANTUNES FLORIANO, 2019; OLIVEIRA, 2016; ROCHA PEREIRA, 2019](#)).

The basic idea is to transfer the state of the variables that determine the conditions of firing transitions between both [PLCs](#). The problem is that as the two [PLCs](#) are not in sync, the condition of a firing transition could change in the middle of the logic. So, in order to circumvent it, we can duplicate those variables in the slave PLC, and in the “Prepare Received data” module update the values of these variables (using positive edges contacts), forcing them to be updated only once in each loop and when their value are changed from 0 to 1.

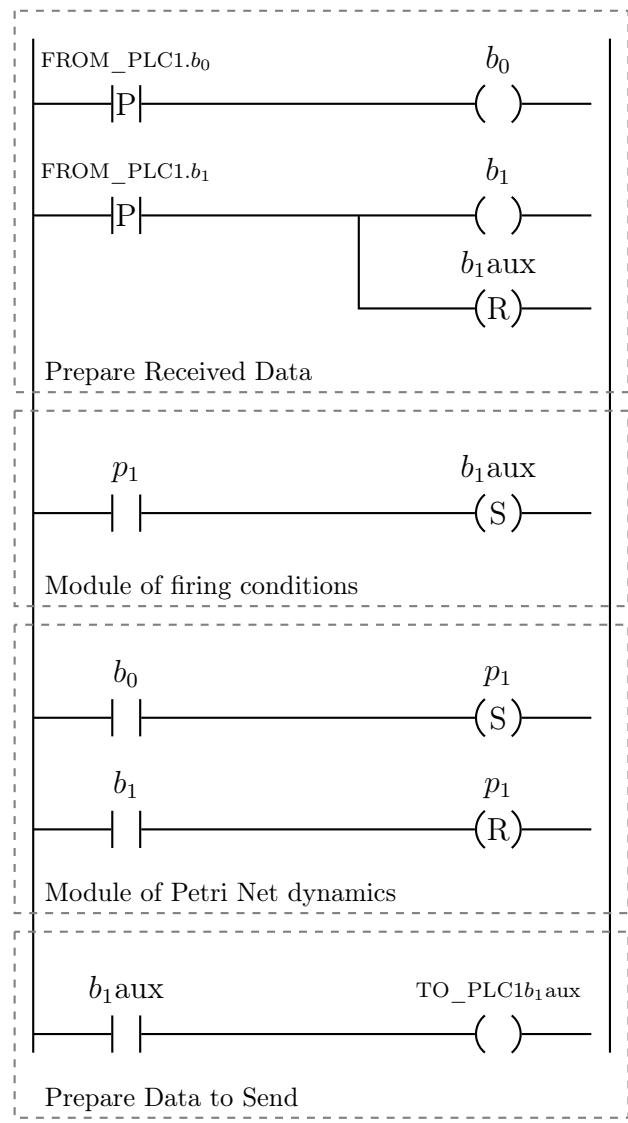
When a transition change the value of tokens in more than one [PLCs](#), as  $t_1$ , an auxiliary variable is created in order to create a kind of acknowledgement/permission from the master, as an “ok, you can proceed” to the slaves to perform the dynamic of

that transition.

The example of those three new modules are shown in Figure 2.25



(a) Ladder Logic on PLC 1.



(b) Ladder Logic on PLC 2.

Figure 2.25: Example of Petri Net divided between 2 PLCs.

As we can expect, this kind of logic of master/slave works well when there are 2 PLCs, but when there are more PLCs, this centralised approach creates a single point of failure and as the master PLC works as a hub, it increases the communication delay. When there are more than two PLCs it is preferable to use a distributed approach as the one shown in [ANTUNES FLORIANO \(2019\)](#).

## 2.7 Identification

Once the control is implemented and the system is working as it should, we arrive at the objective of this work, the identification of the system. As said in the introduction, this work is based on [MOREIRA and LESAGE \(2018\)](#). In this article, a new model for DES identification is proposed, called Deterministic Automaton with Outputs and Conditional Transitions (DAOCT). This model was created with aim of fault detection based on the observation of the fault free behaviour of the system, as in the models proposed by [ROTH et al. \(2009\)](#) and [KLEIN et al. \(2005\)](#), but its use of paths increases the efficiency for fault detection when compared with the latter articles.

This fault free observation is made by the acquisition of the observable signals of the system (controller inputs and outputs) for a sufficiently long period of time while the system works normally. These signals can be seen on [Figure 2.26](#).

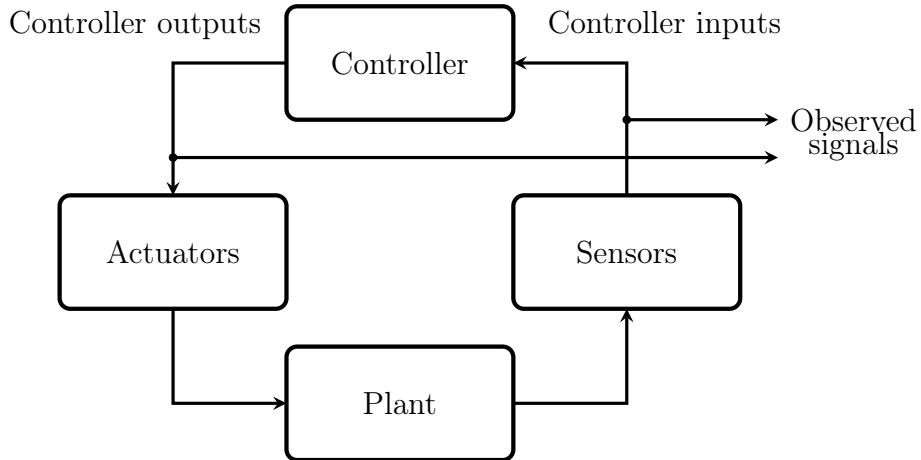


Figure 2.26: Observed Signals in a closed-Loop DES.

First, we assume the controller have  $m_i$  binary inputs,  $i_h$ , for  $h = 1, \dots, m_i$  and  $m_o$  binary outputs,  $o_h$ , for  $h = 1, \dots, m_o$ , and we create a Input/Output vector called  $\mathbf{u}$ , that varies with time:

$$\mathbf{u}(t_1) = \begin{bmatrix} i_1(t_1) & \dots & i_{m_i}(t_1) & o_1(t_1) & \dots & o_{m_o}(t_1) \end{bmatrix}^T$$

This vector  $\mathbf{u}(t)$  represents the status of the system in a instant  $t$ . In the DAOCT model, only untimed system models are considered, thus the status of the system can only be modified via system events,  $\sigma$ . To reduce the size of the notation  $\mathbf{u}(t)$  is going to be represented as  $\mathbf{u}_t$ . The transition between status in  $t_i$  and  $t_j$  is represented as  $(\mathbf{u}_i, \sigma, \mathbf{u}_j)$ . If we have a sequence of  $l$  input/output vectors, we have an observed path of the system,  $p = (\mathbf{u}_1, \sigma_1, \mathbf{u}_2, \sigma_2, \dots, \sigma_{l-1}, \mathbf{u}_l)$ . So, if we observe multiple paths meanwhile our observation process, we can create multiple paths  $p_i = (\mathbf{u}_{i,1}, \sigma_{i,1}, \mathbf{u}_{i,2}, \sigma_{i,2}, \dots, \sigma_{i,l_i-1}, \mathbf{u}_{i,l_i})$ , for  $i = 1, \dots, r$ , where  $r$  is the number of observed paths, and  $l_i$  is the number of vertices in each path  $p_i$ .

Supposing that all paths begin with the same I/O vector, that means, all observations begin from the same status, it is possible to associate to each path  $p_i$  a sequence of events and a sequence of I/O vectors, called,  $s_i$  and  $\omega_i$ , and they are defined as:

$$s_i = \sigma_{i,1}\sigma_{i,2} \dots \sigma_{i,l_i-1}$$

$$\omega_i = \mathbf{u}_{i,1}\mathbf{u}_{i,2} \dots \mathbf{u}_{i,l_i}$$

Using these observed sequences of events  $s_i$ , we can define a language , called observed language,  $L_{Obs}$ :

$$L_{Obs} := \bigcup_{i=1}^r \overline{\{s_i\}} \quad (2.2)$$

**N.B. 2.1** *If any sequence  $s_i$  is the prefix of another sequence  $s_j$ , the path  $p_i$  should be discarded, along with  $s_i$  and  $\omega_i$ , since it does not present any new information for the identification process.*

The objective of identification is to find a model with a language that can simulate this observed language. This language of the identified model is called  $L_{Iden}$ . We can describe this relation between these languages as  $L_{Obs} \subseteq L_{Iden}$ . As said in MOREIRA and LESAGE (2018), in finite time only part of the sequences of events the system can generate are observed. So we can define a language  $L_{Orig}$ , the original language generated by the system, that we is never known.

From  $L_{Orig}$  and  $L_{Iden}$ , we can define another language, an exceeding language  $L_{Exc}$ , that is, part of the identified language that is not present in the original language,

$L_{Exc} = L_{Iden} \setminus L_{Orig}$ . The relation between  $L_{Orig}$  and  $L_{Obs}$  is  $L_{Obs} \subset L_{Orig}$ . And from these two languages we can define another language  $L_{OrigNI}$ , part of the original language that was not identified,  $L_{OrigNI} = L_{Orig} \setminus L_{Iden}$ . The relation between all these languages is shown in Figure 2.27.

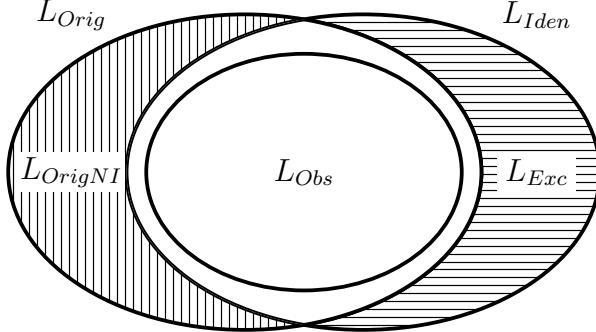


Figure 2.27: Venn diagram showing the relation between  $L_{Orig}$ ,  $L_{OrigNI}$ ,  $L_{Obs}$ ,  $L_{Exc}$  and  $L_{Iden}$

As  $L_{Exc}$  represents a part of the identified language that is not part of the original language, some faulty sequences will not be detected, as they are part of the identified language. So in order to reduce the number of these non detected faults,  $L_{Exc}$  should have a cardinality as close to 0 as it is possible, this is possible by creating identification models that generate smaller  $L_{Exc}$ .

And as  $L_{OrigNI}$  represents the part of the original system that was not identified, that means some sequences present on the fault-free behaviour of the system will be detected as faults, generating false alarms. This language should also be reduced, so the false alarms generated are reduced. As the original language of the system is never known, it is very difficult to know for sure if  $L_{OrigNI}$  is small or not. KLEIN *et al.* (2005) show that if a system is observed for a sufficiently long time, there exists a number  $n_0 \in \mathbb{N}$  such that  $L_{Orig}^{\leq n_0} \setminus L_{Obs}^{\leq n_0} \approx \emptyset$ , where  $L_{Orig}^{\leq n_0}$  and  $L_{Obs}^{\leq n_0}$  denote the languages formed by all sequences of events of length smaller than or equal to  $n_0$  of  $L_{Orig}$  and  $L_{Orig}$ . Since  $L_{Obs} \subseteq L_{Iden}$ ,  $L_{OrigNI}^{\leq n_0}$  is also approximately the empty set.

In this work we assume that all sequences of events that have length  $n_0 + 1$  were observed, thus  $L_{OrigNI}^{\leq n_0} = \emptyset$ . So it leaves to the identification model the sole problem of reducing the language  $L_{Exc}^{\leq n_0}$  and  $L_{Exc}$  consequentially.

### 2.7.1 DAOCT

In this subsection the modified automaton model proposed by MOREIRA and LESAGE (2018) will be explained and the algorithm to construct it by the observed paths  $p_i$  will be described. The definition of the automaton is the following:

#### Definition 2.8 (DAOCT)

*A Deterministic Automaton, denoted by DAOCT, is a nine-tuple*

$$\text{DAOCT} = (X, \Sigma, f, \lambda, R, \theta, x_0, X_f)$$

where:

*X is the set of states*

*$\Sigma$  is the finite set of events*

*$\Omega \subset \mathbb{N}_1^{m_i+m_o}$  is the set of I/O vectors*

*$f : X \times \Sigma^* \rightarrow X$  is the deterministic transition function*

*$\lambda : X \rightarrow \Omega$  is the state output function*

*$R = 1, 2, \dots, r$  is the set of path indices*

*$\theta : X \times \Sigma \rightarrow 2^R$  is the path estimation function*

*$x_0$  is the initial state*

*$X_f \subseteq X$  is the set of final states*

The sets of events and I/O vectors of each path  $p_i$  are denoted  $\Sigma_i$  and  $\Omega_i$ . Thus  $\Sigma$  and  $\Omega$  can be calculated by the union of these sets:  $\Sigma = \bigcup_{i=1}^r \Sigma_i$  and  $\Omega = \bigcup_{i=1}^r \Omega_i$ . From the vertices of the paths  $p_i$ , the I/O vectors, the states of the model are extracted. Each vertex is chosen as a new state. But depending on the system, the vertices can be repeated in different paths and even in the same path, so a kind of memory of the past I/O vectors can be interesting to differentiate them, that means, taking them as different status of the system. This kind of memory is presented on the form of a sequence of I/O vectors that stores a certain number of the past ones. To determine the number of vectors stored in this sequence an arbitrary variable  $k$ , a free parameter, is created. The modified paths created by substituting the vertices for these sequences of I/O vectors are denoted  $p_i^k$  and are defined as follows:

$$p_i^k = (y_{i,1}, \sigma_{i,1}, y_{i,2}, \sigma_{i,2}, \dots, \sigma_{i,l_1-1}, y_{i,l_i}) \quad (2.3)$$

where

$$y_{i,j} = \begin{cases} (\mathbf{u}_{i,j-k+1}, \dots, \mathbf{u}_{i,j}), & \text{if } k \leq j \leq l_i \\ (\mathbf{u}_{i,1}, \dots, \mathbf{u}_{i,j}), & \text{if } j < k \end{cases} \quad (2.4)$$

**N.B. 2.2** Depending on the choice of the value of  $k$ , some characteristics can be observed on  $p_i^k$ . For instance, if  $k = 1$ ,  $p_i^k = p_i$ . And if  $k$  is equal to the larger  $l_i$ , then all  $y_{i,l_i}$  are composed with all vertices of its corresponding path  $p_i$ .

As an example of the computation of the paths  $p_i^k$ , we can take the observation of the following paths, from MOREIRA and LESAGE (2018):

$$\begin{aligned} p_1 &= \left( \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, a, \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}, b, \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}, c, \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, d, \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, e, \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \right) \\ p_2 &= \left( \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, g, \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, h, \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}, b, \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}, c, \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, i, \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, j, \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}, l, \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \right) \\ p_3 &= \left( \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, g, \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, h, \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}, b, \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, i, \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}, m, \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, d, \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, n, \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \right) \end{aligned}$$

The events of each path is associated with the rising or the falling edges of the controller signals. The event  $a$  represents the rising edge of the second controller signal, that is  $a = \uparrow 2$  and the event  $l$  the rising edge of the first controller signal and the falling edge of the second and third controller signals,  $l = \uparrow 1 \downarrow 2 \downarrow 3$ .

As said, for  $k = 1$ ,  $p_i^k = p_i$ , so in order to better illustrate the construction of  $p_i^k$ , we choose  $k = 2$ . Using Equations 2.3 and 2.4 we can obtain the following modified paths:

$$\begin{aligned} p_1^2 &= \left( \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, a, \begin{bmatrix} 1 & 1 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}, b, \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix}, c, \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \end{bmatrix}, d, \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix}, e, \begin{bmatrix} 0 & 1 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \right) \\ p_2^2 &= \left( \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, g, \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}, h, \begin{bmatrix} 0 & 1 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}, b, \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix}, c, \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \end{bmatrix}, i, \begin{bmatrix} 0 & 1 \\ 0 & 0 \\ 1 & 0 \end{bmatrix}, j, \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \end{bmatrix}, l, \begin{bmatrix} 0 & 1 \\ 1 & 0 \\ 1 & 0 \end{bmatrix} \right) \\ p_3^2 &= \left( \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, g, \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}, h, \begin{bmatrix} 0 & 1 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}, b, \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{bmatrix}, i, \begin{bmatrix} 0 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix}, m, \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 1 & 0 \end{bmatrix}, d, \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix}, n, \begin{bmatrix} 0 & 1 \\ 0 & 1 \\ 1 & 0 \end{bmatrix} \right) \end{aligned}$$

As we can see, comparing the vertices from  $p_i$  with the vertices from  $p_i^k$ , the number of unique vertices from the latter is greater than the number from the former.

As presented, the states of the system are extracted from these vertices, in order to do so a labelling function is defined, denoted  $\tilde{\lambda}$ , which definition is  $\tilde{\lambda} : X \rightarrow \Omega^k$ . Where  $\Omega^k$  is formed by all sequences of  $\Omega$  of length smaller than or equal to  $k$ . This function  $\tilde{\lambda}$  associates a sequence of I/O vectors  $\omega^k \in \Omega^k$  to each state  $x \in X$ . And we can denote  $\tilde{\lambda}_l(x)$  as the last vector of  $\tilde{\lambda}(x)$ .

These functions are used in the identification algorithm of the DAOCT model. This identification algorithm, adapted from MOREIRA and LESAGE (2018), is presented in algorithm 1.

---

**Algorithm 1:** Identification Algorithm

---

**Input:** Modified observed paths  $p_i^k$ , for  $i = 1, \dots, r$   
**Output:** DAOCT =  $(X, \Sigma, \Omega, f, \lambda, R, \theta, x_0, X_f)$

```

1 Create an initial state  $x_0$ , and define  $\lambda(x_0) = \tilde{\lambda}(x_0) = y_{1,1}$ 
2  $X = \{x_0\}, X_f = \emptyset, R = \emptyset$ 
3 for  $i = 1$  to  $r$  do
4    $R = R \cup \{i\}$ 
5   for  $j = 1$  to  $l_i - 1$  do
6     Find the State  $x \in X$  such that  $\tilde{\lambda}(x) = y_{i,j+1}$ 
7     if  $\tilde{\lambda}(s) \neq y_{i,j+1}$  for all  $s \in X$  then
8       Create state  $x'$  and define  $\tilde{\lambda}(x') = y_{i,j+1}$ 
9        $X = X \cup \{x'\}$ 
10       $\lambda(x') = \tilde{\lambda}_l(x')$ 
11    else
12      Find  $x' \in X$  such that  $\tilde{\lambda}(x') = y_{i,j+1}$ 
13    end
14     $f(x, \sigma_{i,j}) = x'$ 
15    Add  $i$  to  $\theta(x, \sigma_{i,j})$ 
16    if  $j = l_i - 1$  then
17       $X_f = X_f \cup \{x'\}$ 
18    end
19  end
20 end

```

---

Feeding the paths modified paths  $p_1^1, p_2^1, p_3^1$ , that are equal to the normal paths

$p_1, p_2, p_3$ , and the paths  $p_1^2, p_2^2, p_3^2$  to algorithm 1, two models are identified, one for  $k = 1$  and another for  $k = 2$ , their state transition diagrams are represented in Figures 2.28 and 2.29 respectively.

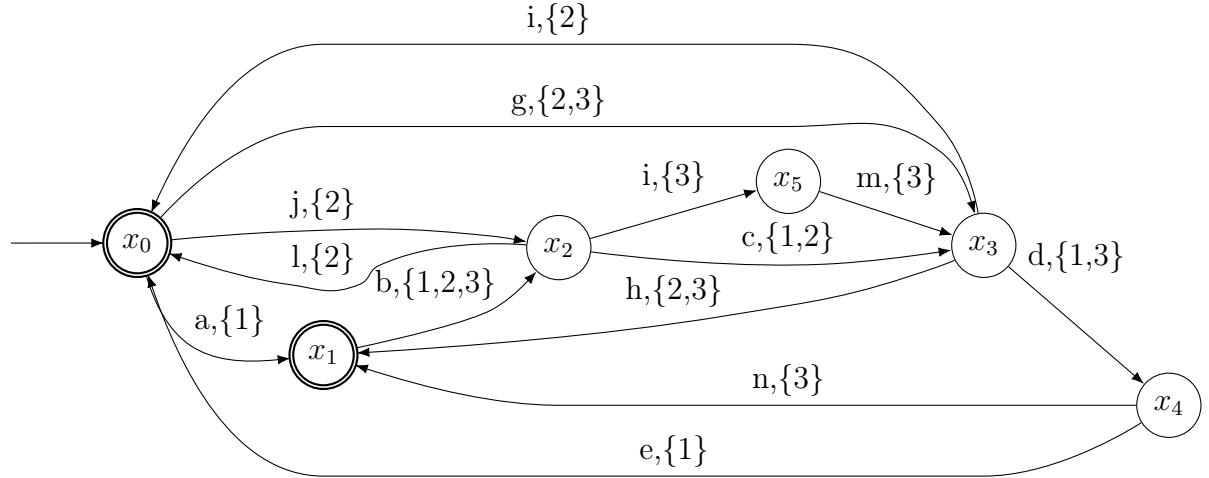


Figure 2.28: State transition diagram for identified model using  $k = 1$ .

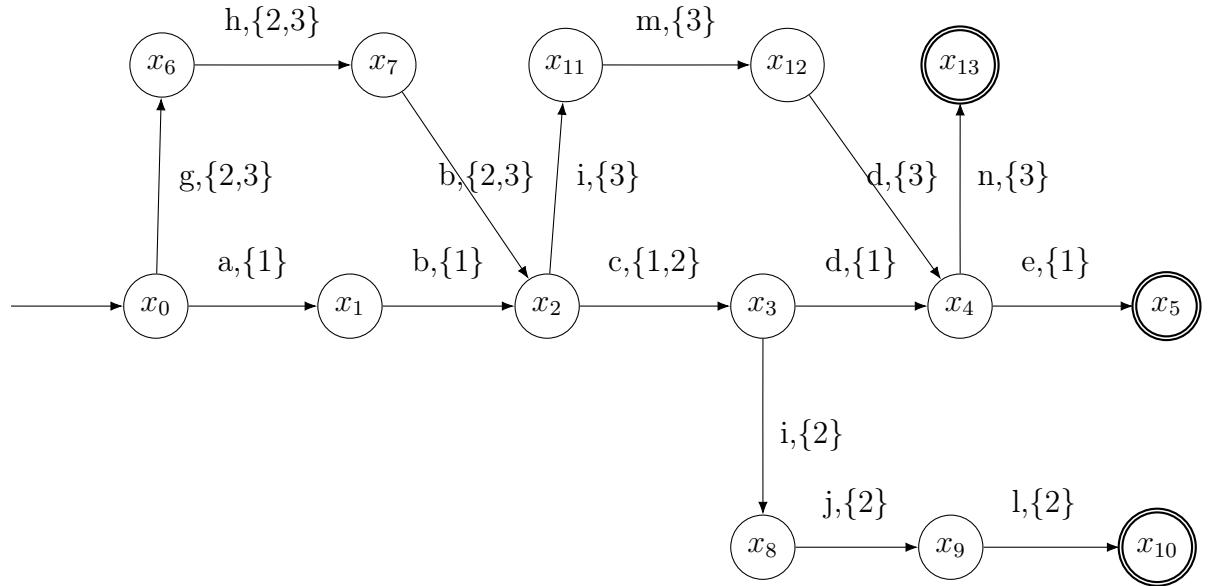


Figure 2.29: State transition diagram for identified model using  $k = 2$ .

As expected, with a greater value of  $k$ , more states are identified, since the number of unique vertices increases with  $k$ .

One thing that can be noted in those state transition diagrams is that the resulting set of the path estimation function  $\theta$  of each transition is represented besides the

corresponding event of that transition. That is made with the purpose of showing the “conditional transitions” part of the **DAOCT** model. A transition is enabled and can occur if and only if all previous transitions are associated to the same path. To summarise this in mathematical language it is needed to expand the domain of function  $\theta$  to consider sequence of events, instead of only an event. This new estimation function will be denoted as  $\theta_s : X \times \Sigma^* \rightarrow 2^R$ . And it is defined recursively as:

$$\begin{aligned} \theta_s(x, \epsilon) &= R, \\ \theta_s(x, s\sigma) &= \begin{cases} \theta_s(x, s) \cap \theta(x', \sigma), & \text{where } x' = f(x, s), \text{ if } f(x, s\sigma)! \\ \text{undefined, otherwise.} & \end{cases} \end{aligned} \quad (2.5)$$

Where ! denotes *is defined*.

And once this function is defined we can describe the language generated by this identified model:

$$L(DAOCT) := \{s \in \Sigma^* : f(x_0, s)! \wedge \theta_s(x_0, s) \neq \emptyset\} \quad (2.6)$$

From a similar way it is possible to define the language formed by all subsequences of events of length  $n$  generated by the identified model:

$$L_S^n(DAOCT) := \{s \in \Sigma^* : (|s| = n) [\exists x_i \in X, f(x_i, s)! \wedge \theta_s(x_i, s) \neq \emptyset]\} \quad (2.7)$$

In order to calculate the exceeding language  $L_{Exc}^{\leq n}$ , another language is presented,  $L^{\leq n}(DAOCT)$ , since the definition of  $L_{Exc}^{\leq n}$  is  $L_{Exc}^{\leq n} = L^{\leq n}(DAOCT) \setminus L_{Orig}^{\leq n}$ .

$$L^{\leq n}(DAOCT) := \left( \bigcup_{i=0}^n L_S^i(DAOCT) \right) \cap L(DAOCT) \quad (2.8)$$

As we assumed that all sequences of events that have length  $n_0 + 1$  were observed, then  $L_{Orig}^{\leq n_0} \approx L_{Orig}^{< n_0}$ . If  $n \leq n_0$ , then  $L_{Orig}^{\leq n} \approx L_{Orig}^{< n}$ , and  $L_{Exc}^{\leq n} = L^{\leq n}(DAOCT) \setminus L_{Obs}^{\leq n}$ , and this formula will be used to calculate  $L_{Exc}^{\leq n}$  throughout this work.

# Chapter 3

## System

In this chapter, the system to be control and identified is presented. The system is a didactic manufacture system assembled from submodules fabricated by Christiani<sup>1</sup>, a German constructor specialised in Mechatronic Systems and Industry Models to didactic ends. This manufacture system is located in the [Control and Automation Laboratory \(LCA\)](#), situated in the [Federal University of Rio de Janeiro \(UFRJ\)](#). This system is normally used for the under-graduated studies about Industrial Automation and control of [DESSs](#), and also for data acquisition in some bachelor/master/doctorate thesis, as this one.

This manufacture system is a cube assembly system, where the different cube halves shown in [Figure 3.1](#) are put together to form cubes.



Figure 3.1: Cube halves.

The pieces can be of two materials, metal or plastic, and the plastic ones can be white or black. The permutation of cube halves needed to form a cube is selected via a type of sorting, selecting the type of piece by material and colour. The assembled cubes are then stored. In order to perform these tasks (sorting, handling, assembling

---

<sup>1</sup>All images from the Christiani modules are present on its sales catalog, available at [www.christiani.de](http://www.christiani.de). All rights are reserved to Christiani.

and stocking), 6 Units from Christiani manufacturer are used. These units can be seen in Figure 3.2.

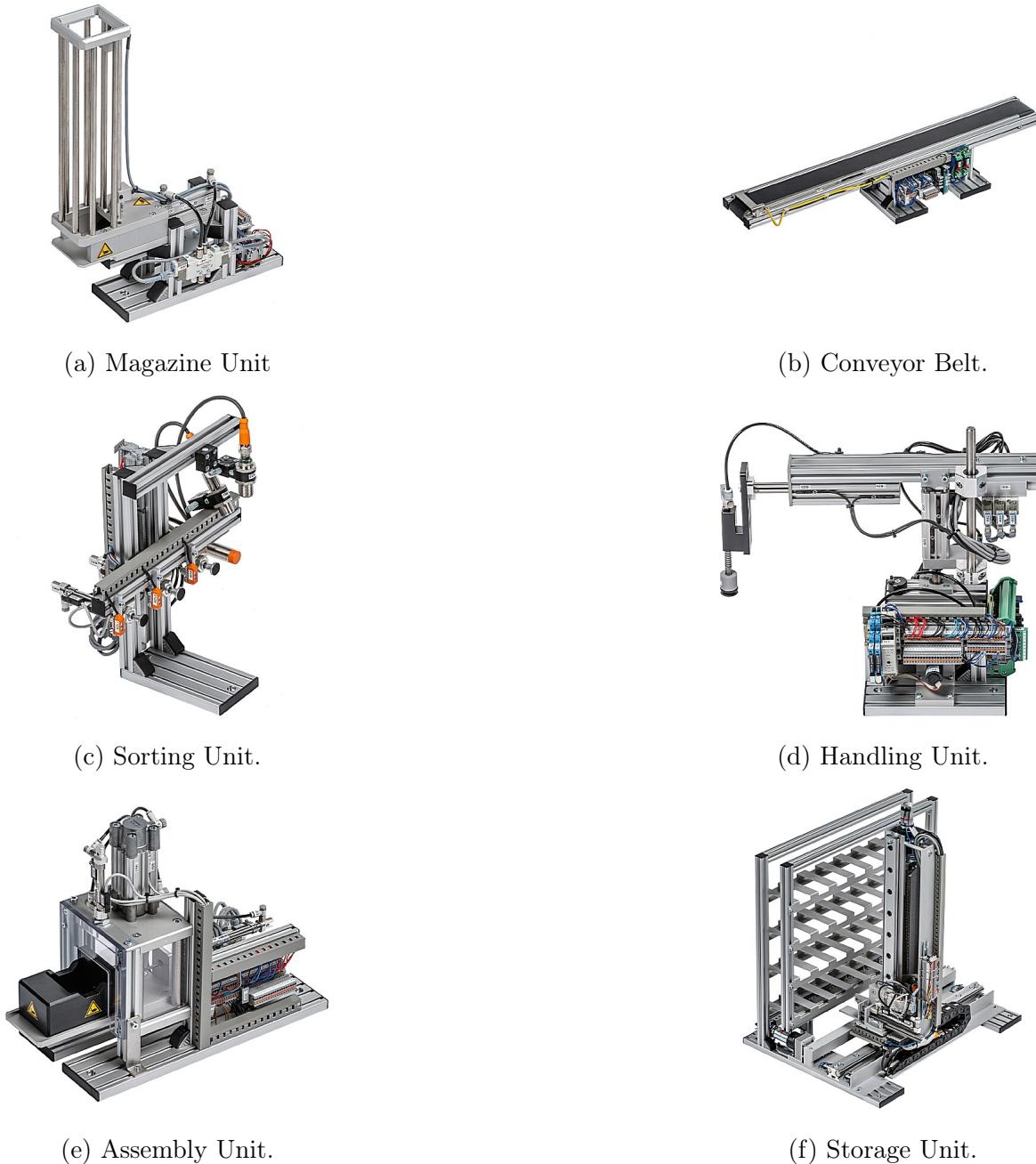


Figure 3.2: Units of the Manufacture System.

In the next sections each unit and their Inputs/Outputs will be detailed.

**N.B. 3.1** *What is described in the next sections as an input of a certain module, it is considered as an output for the controller and vice versa.*

### 3.1 Magazine Unit

The magazine is a unit with the objective to stock the cube halves to be used. There are 2 types of magazines, one to stock pieces without connection pins (the pins shown in Figure 3.1) and another to stock pieces with those pins inserted, they can stack 10 and 8 pieces respectively. They will be denominated MAG 1 and MAG 2. Each magazine has a pneumatic cylinder and a presence button. The cylinder serves to extract a piece from the bottom of the stack, and the button to know if the stack is empty or not. Each one of these cylinders have 2 inputs that are used to extend and retract the cylinders (if they are set to *true*), this kind of cylinder is called *double acting pneumatic cylinder*. There are also 2 outputs used to know if the cylinders are extended or retracted, the output is equal to *true* if the respective condition is fulfilled. The inputs to control the cylinders are called in this work **Extend MAG 1/2 Cylinder** and **Retract MAG 1/2 Cylinder**, and the outputs are called **MAG 1/2 Cylinder Extended** and **MAG 1/2 Cylinder Retracted**. The presence button of each magazine outputs a *true* value if the stack is empty and *false*, otherwise. Thus this presence buttons are called in this work **MAG 1/2 Empty**, their localisation on the magazine can be seen in Figure 3.3.

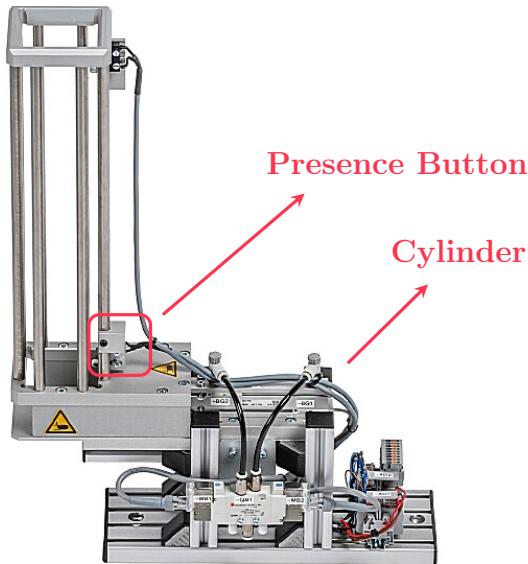


Figure 3.3: Magazine Unit.

## 3.2 Conveyor Belt

The conveyor belt is a unit with the objective of transporting the pieces from a unit to another. It has 2 inputs and 1 output. The inputs are used to turn the belt on, but each input makes it turn in a direction or the other. The output is generated by a presence sensor located in one extremity of the belt (see [Figure 3.4](#)), it is equal to *true* if there is a piece in front of it and *false* otherwise. The directions of the movement of the pieces are denominated Forward if it is going towards the presence sensor and reverse if not. Thus the names given to the inputs that generate these movements are **Conveyor Belt Forward** and **Conveyor Belt Reverse**. And the output is called **Proximity Sensor End of Conveyor Belt**.

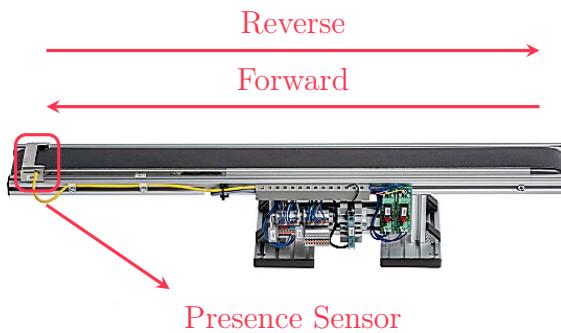


Figure 3.4: Conveyor Belt.

## 3.3 Sorting Unit

As the name says, the sorting unit serves to sort the pieces. To understand its inputs and outputs, we will divide them in 2 parts, identification and discharging.

The identification part uses 3 sensors to identify the type of half cube: a distance sensor to identify the orientation of the concavity of the piece, an optic sensor to identify the color of the plastic piece, and an inductive sensor to identify the material of the piece. The output of the inductive sensor is *true* if the piece is made of metal, and *false* if it is not, thus the given name to this output is **Metallic Sensor**. The output of the optic sensor is equal to *true* if the piece is reflexive (white) and *false*, otherwise (black), thus it is named **White Color Sensor**. The distance sensor outputs an integer corresponding to the distance between the piece and the sensor, it is named **Distance Sensor**, the logic used to find the orientation of the piece is discussed in [chapter 4](#). The placement of these 3 sensors can be seen in [Figure 3.5](#).

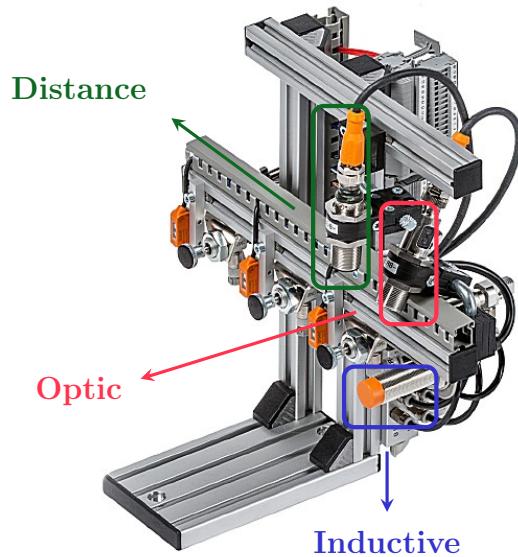


Figure 3.5: Sorting Unit - Identification.

The discharging part is formed by 3 groups of inputs and outputs, denoted Left, Center and Right, as shown in [Figure 3.6](#). Each group has a pneumatic cylinder and a presence sensor, and uses them to discharge pieces depending on the logic of sorting and the identified piece by the identification part of the sorting unit.

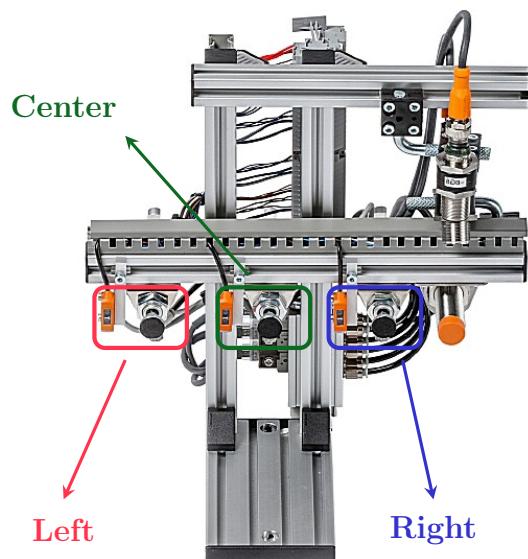


Figure 3.6: Sorting Unit - Discharging.

Differently from MAG 1/2, each cylinder can be extended and retracted using a single

input, the name of this kind of cylinder is *single acting pneumatic cylinder*. When the corresponding input is equal to *true* it extends, but when it is *false* it is automatically retracted. A name for each input is given depending on the group name, for instance, to extend the left cylinder we use **Extend Left Discharge Cylinder** as input. Each one of these cylinders has 2 outputs to determine if the cylinder is extended or retracted, similarly the names depend on the group, e.g.: **Right Discharge Cylinder Extended** and **Right Discharge Cylinder Retracted**. The presence sensor of each group detects if there is a piece in front of the cylinder, and its name also depends on the group, e.g.: **Proximity Sensor Center Discharge Cylinder**.

## 3.4 Handling Unit

The handling Unit is a robotic manipulator that serves to transfer the pieces and eventually assembled pieces, from a unit to other. By the definitions of robotic manipulators shown in **KHALIL and DOMBRE (2004)**, this manipulator has 3 **Degrees of Freedom (DOF)**, and it is from the type called *RPP*, as it is formed by a Revolute joint and two Prismatic joints, the latter joints being orthogonal regarding each other.

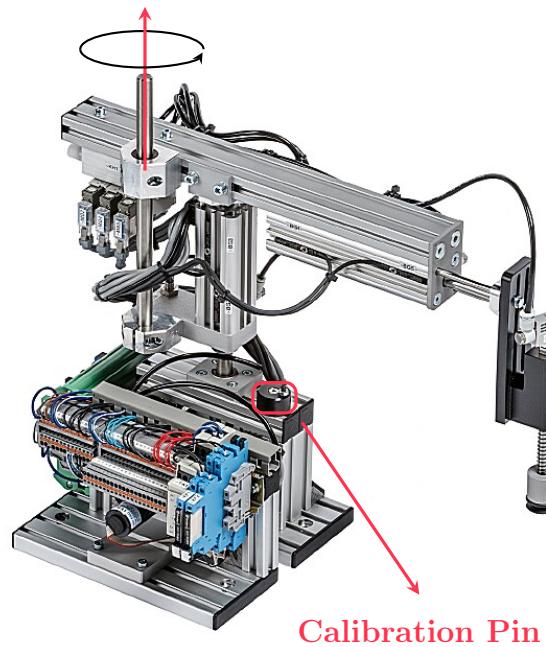
Since the position of its *end-effector* (the end of the robotic arm) can be described using a cylindrical coordinate system, this kind of manipulator is also called *cylindrical shoulder*. With the end of easing the understanding of the verbs used in this work to describe the movements of the manipulator, in this section we will use beside these verbs the cylindrical coordinate system  $(\rho, \phi, z)$ , where  $\rho$  is the axial distance,  $\phi$  is the azimuth and  $z$  is the height. The *end-effector* of this manipulator is equipped with a vacuum suction device capable of holding the pieces, which is controlled by an input called **Turn Vacuum Gripper On** that evidently turns the vacuum on when activated.

In order to control the position of the *end-effector* of the manipulator, which for brevity's sake will be called "arm" throughout this work, there is a couple of pneumatic cylinders, whose behaviour is similar to the ones in the sorting unit (*single acting pneumatic cylinder*), placed in each prismatic joint, thus raising (increasing the height  $z$ ) and extending (increasing the axial distance  $\rho$ ) the arm when they are activated respectively. The respective inputs of these cylinders are called **Raise Arm** and **Extend Arm**. Each cylinder also has 2 outputs to identify if they are retracted or extended, the names are given in the most mnemonic way possible, they are called **Arm Lowered**, **Arm Raised**, **Arm Retracted** and **Arm Extended**.

In order to rotate the revolute joint, a motor is placed under the arm. This motor

have two inputs that when activated makes the arm rotate in one direction or the other, which will be called **Clockwise (CW)** and **Counter Clockwise (CCW)**, and the inputs that generate these kind of movements denoted **Turn Arm CW** and **Turn Arm CCW**. But to identify what is considered **CW** and **CCW** it is needed to impose one of those rotation direction. In this work we imposed the **CCW** direction as shown by the black arrow superposed to the arm in [Figure 3.7](#). This same direction is considered as the positive direction where the azimuth  $\phi$  increases. The *zero position* of the arm azimuth ( $\phi = 0$ ), is considered when the *end-effector* is diametrically opposed to the calibration pin shown in [Figure 3.7](#). This pin as the name suggests is used for the calibration of the arm. The arm has an inductive sensor in the opposite to the *end-effector* that is activated when it is aligned to this pin. The azimuth of the arm in [Figure 3.7](#) is  $\phi = 180^\circ$ .

An encoder is also present in this arm, in order to estimate the azimuth angle, but as the angular velocity of the arm is relatively big, once the resolution of the arm is very precise, the output of this encoder is connected to a High Speed Counter in a **PLC**, in order to correctly estimate the angle. The configuration of the High Speed Counter can be seen in [ANTUNES FLORIANO \(2019\)](#); [ROCHA PEREIRA \(2019\)](#).



[Figure 3.7: Handling Unit.](#)

## 3.5 Assembly Unit

The assembly unit serves to mount two pieces, resulting in a fully assembled cube. As a safety measure, the assembly unit has a compartment made of acrylic, in which a pneumatic cylinder is vertically arranged pointing downwards to work as a press. When the cylinder is extended, this press is lowered exerting a considerable pressure on the pieces binding them together.

The inputs to lower and raise the press are called **Lower Press** and **Raise Press**.

In order to open and close the compartment, there is an acrylic door combined with a smaller pneumatic cylinder. When this cylinder is extended, the door closes and when it is retracted the door opens. The inputs to open and close the door are called **Open Safety Door** and **Close Safety Door**. There are a couple of sensors to verify if the door is opened or closed based on the extension and retraction of the cylinder, their respective outputs are called **Safety Door Opened** and **Safety Door Closed**.

Since there is a safety compartment, where the cubes are assembled, There must be a device with the purpose of transposing the pieces from outside to inside and vice-versa. This device is called in this work **Assembly Unit Holder**, and as the name says it holds the pieces. This device is coupled with another pneumatic cylinder that when it is retracted it transports the **Assembly Unit Holder** to the inner part of the compartment, and outside the compartment if it is extended. The inputs that move the **Assembly Unit Holder** are called **Extend Assembly Unit Holder** and **Retract Assembly Unit Holder**. The outputs to tell the extension and retraction are called **Assembly Unit Holder Extended** and **Assembly Unit Holder Retracted**. The position of the cylinders can be seen in [Figure 3.8](#).

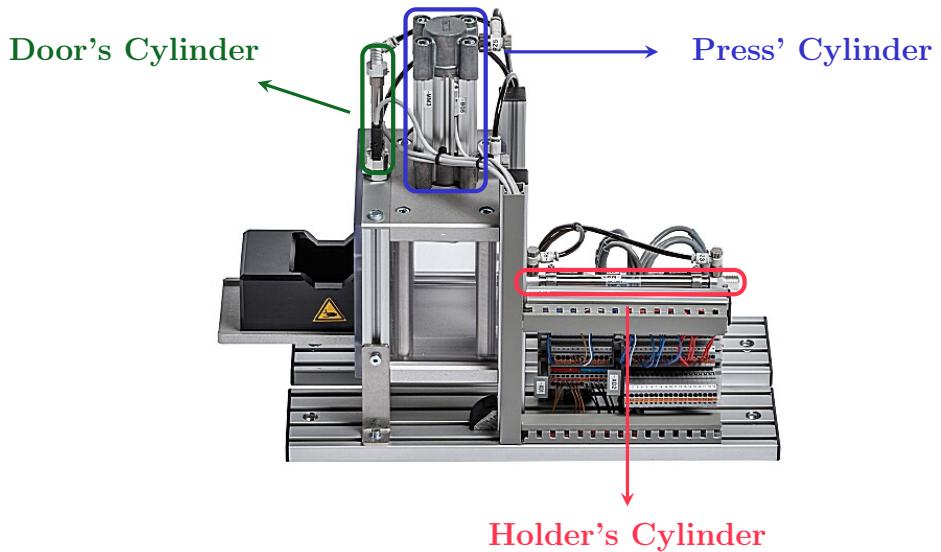


Figure 3.8: Assembly Unit.

### 3.6 Storage Unit

The storage unit in reality is a storage and retrieval system, but in this work will be called only storage unit, since it will be its sole use. The storage unit is a rack composed by 4 shelves, each one of them with enough room to store 7 pieces, resulting a total of 28 storage spaces. In order to elevate the pieces, a motor with a spiral shaft is used to raise and lower the device where the piece is placed to be stored. This piece holder is also called as **Storage Device** and sometimes as **Storage Unit**, so when a movement is given to the **Storage Unit**, that means this holder is moved and not the rack itself. There is also another motor with spiral shaft that moves the **Storage Unit** horizontally from Right to Left and vice versa. As a reference for the direction of the movements of the **Storage Unit** used in this work the [Figure 3.9](#) shows what is considered the Right, Left, Top and Bottom of this unit.

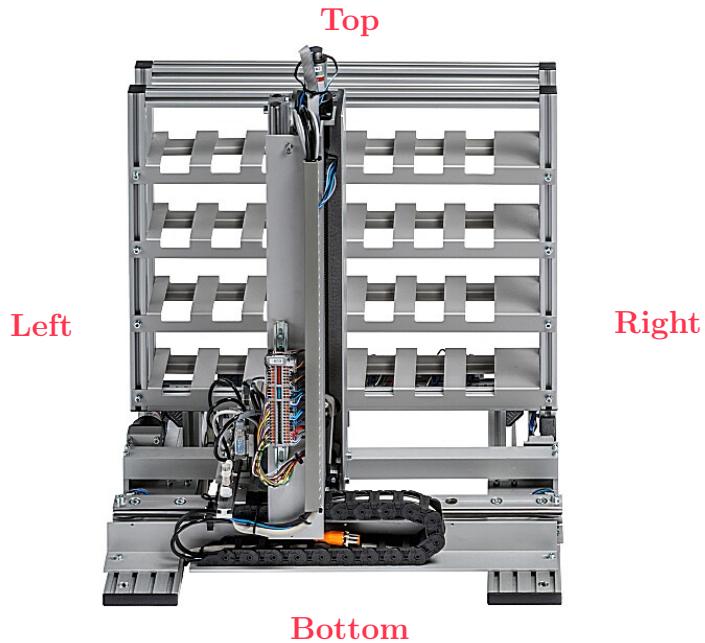


Figure 3.9: Storage Unit.

To effectively store the piece in the rack it is needed to move the **Storage Device** towards the rack, so a pneumatic cylinder is coupled with this device, when this cylinder is extended the device approaches the rack and it leaves the rack when the cylinder is retracted. This cylinder is a *single acting pneumatic cylinder* and its input is called **Extend Storage Unit**. There are also 2 outputs to tell if the cylinder is extended or retracted, called **Storage Unit Extended** and **Storage Unit Retracted**.

The movement of the **Storage Unit** is controlled by 4 inputs called **Move Storage Unit Upwards**, **Move Storage Unit Downwards**, **Move Storage Unit to the Right** and **Move Storage Unit to the Left**.

In order to estimate the position of the **Storage Device** there are two encoders, called **Storage Unit Vertical Encoder** and **Storage Unit Horizontal Encoder**, that in conjunction with holes specifically placed, aligned with the store spaces can identify if the device is aligned with the store spaces. There are also 4 limit switchers whose outputs are called **Storage Unit Inferior Limit Switch**, **Storage Unit Superior Limit Switch**, **Storage Unit Right Limit Switch** and **Storage Unit Left Limit Switch** that have the purpose to tell if the **Storage Device** is in one of the limits of the rack.

# Chapter 4

## Control

As said in the [Background](#) chapter, a [CIPN](#) can be used to represent the control of a system and then this Petri net can be converted into a [LD](#), with the aim of being implemented in a [PLC](#). So, this section will be divided in two parts: the first part to describe the logic of the control and its design using [CIPN](#) and the second part for the implementation in the [PLC](#).

### 4.1 Logic

The logic of the control is to use the 6 units presented in [chapter 3](#), to assemble cubes made of a metallic half cube with a white plastic one on top, and once the cube is assembled it is stored in one of the store spaces of the [Storage Unit](#). The logic is divided in 10 modules: Initialization, Metal Cube half sorting, Plastic Cube half sorting, Arm From Conveyor Belt to Assembly Unit, Assembly Unit, Arm From Assembly Unit to Storage Unit, Storage Unit positioning (y axis), Storage Unit positioning (x axis), Cube Storage and Arm Stop Logic.

Each module will be briefly described in the next subsections, and their Petri Nets will be presented along with tables that describe the meaning of each place and transition.

**N.B. 4.1** *Each Petri net shown in this chapter is a section of a complete Petri net. The complete one is presented in [Appendix A](#). As said in [chapter 2](#) the dotted places/transitions represent places/transitions that are part of other sections of the Petri net. In the digital form of this work, it is possible to travel between figures by clicking the name of the dotted places/transitions. By clicking in normal places/transitions it is possible to travel between the figure and its corresponding entry in the table that describes its meaning.*

### 4.1.1 Initialization

This module has as objective to make sure that all units are in order to begin the assembling process, that means, all variables used are reset, the arm is calibrated, the conveyor belt is free of pieces, all cylinders are retracted, the assembly unit is ready to receive a piece and the storage unit is in its rightmost and lower position. The Petri net used for this module can be seen in [Figure 4.1](#) and the corresponding meaning of its transitions and places can be seen in [Tables 4.1](#) and [4.2](#)

Table 4.1: Initialization Module Transitions.

Transitions	Meaning
$t_0$	Initialization Button
$t_1$	MAG1's Cylinder Retracted
$t_2$	MAG2's Cylinder Retracted
$t_3$	Right Discharge Cylinder Retracted
$t_4$	Center Discharge Cylinder Retracted
$t_5$	Left Discharge Cylinder Retracted
$t_6$	
$t_7$	T=12s
$t_8$	T=2.5s
$t_9$	Safety Door Opened
$t_{10}$	Assembly Unit Holder Extended
$t_{11}$	Storage Unit Retracted and Arm Lowered and Retracted
$t_{12}$	Storage Unit Right Limit Switch
$t_{13}$	Storage Unit Inferior Limit Switch
$t_{14}$	T=2s
$t_{15}$	Inductive Sensor Arm
$t_{16}$	T=1s
$t_{17}$	ARMCOUNTER <= BELT_ANGLE_CW
$t_{18}$	
$t_{19}$	Start Button

Table 4.2: Initialization Module Places.

Places	Meaning
$p_0$	System Stopped
$p_1$	Retract MAG1's Cylinder *
$p_2$	MAG1's Cylinder Retracted
$p_3$	Retract MAG2's Cylinder *
$p_4$	MAG2's Cylinder Retracted
$p_5$	Retract Right Discharge Cylinder *
$p_6$	Right Discharge Cylinder Retracted
$p_7$	Retract Center Discharge Cylinder
$p_8$	Center Discharge Cylinder Retracted
$p_9$	Retract Left Discharge Cylinder *
$p_{10}$	Left Discharge Cylinder Retracted
$p_{11}$	Turn Conveyor Belt On (Reverse)
$p_{12}$	No Pieces On Conveyor Belt
$p_{13}$	Reset Variables
$p_{14}$	Raise Press
$p_{15}$	Open Safety Door
$p_{16}$	Extend Assembly Unit Holder
$p_{17}$	Assembly Unit Ready
$p_{18}$	Arm Lowered and Retracted, and Storage Unit Retracted
$p_{19}$	Move Storage Unit to the Right
$p_{20}$	Storage Unit ready ( horizontal )
$p_{21}$	Move Storage Device Downwards
$p_{22}$	Storage Unit ready ( vertical )
$p_{23}$	Rotate Arm CCW
$p_{24}$	Turn HSC Off ( Arm Stopped )
$p_{25}$	Rotate Arm CW
$p_{26}$	Arm Stopped facing conveyor belt
$p_{27}$	System Ready

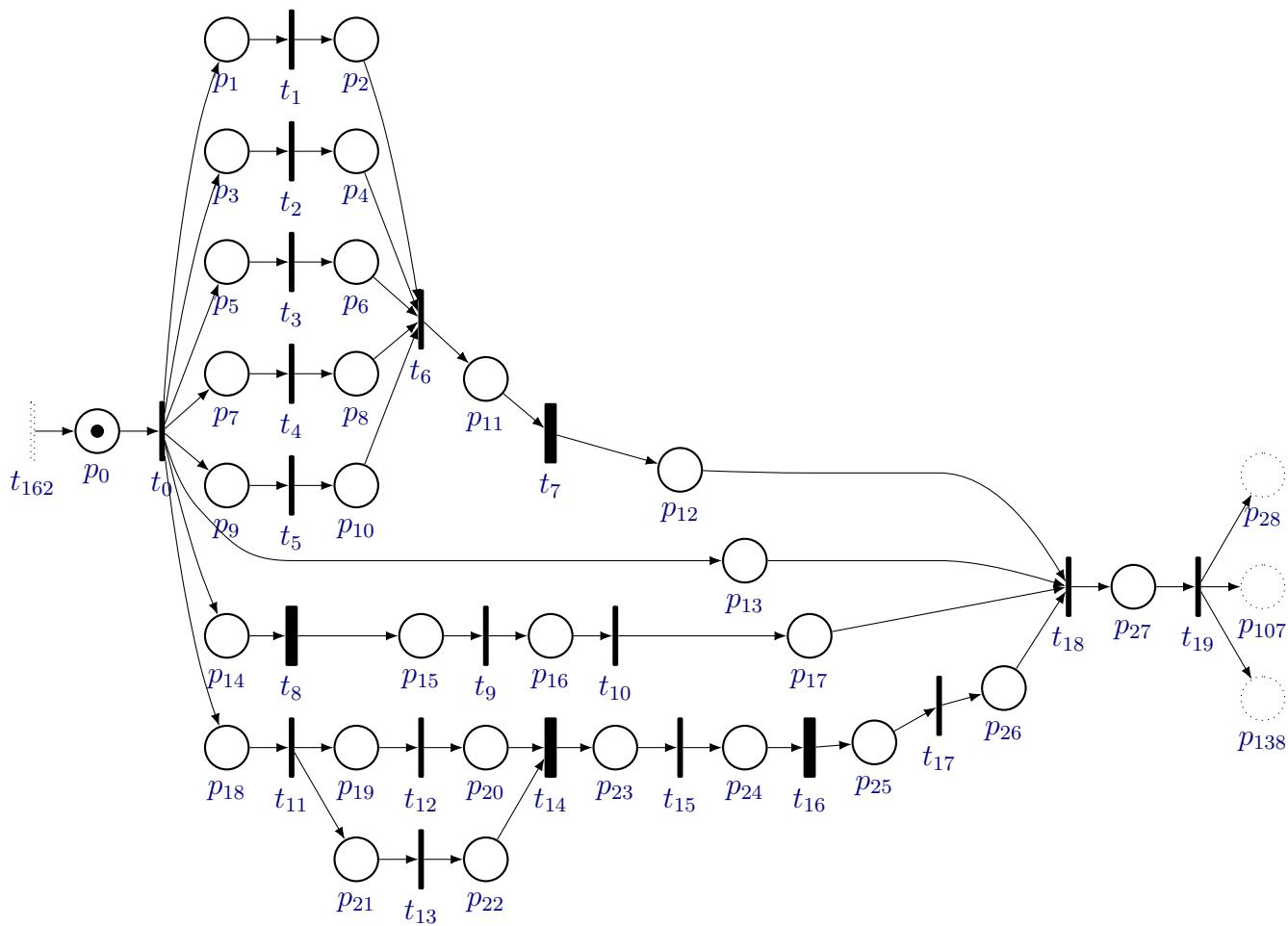


Figure 4.1: Petri net of Initialization module.

### 4.1.2 Metal Cube Half Sorting

This module serves to sort the cube halves stacked in MAG 1. The piece is extracted from the bottom of the stack to the conveyor belt, and the piece is transported by the belt to the identification part of the sorting unit, if it is a metal piece with a upwards concavity the piece continues in the belt until it reaches the end of it, waiting to be picked by the arm, otherwise this piece is discharged using the sorting unit and the cycle recommences and stops only when a metallic piece is at the end of the belt. In order to recognize the orientation of the pieces (upwards or downwards), the distance sensor is combined with comparison blocks to create two variables ConcUP and ConcDWN. The corresponding Petri net and tables can be seen in Figure 4.2 and Tables 4.3 and 4.4.

Table 4.3: Metal Half-cube Selection Module Transitions.

Transitions	Meaning
$t_{20}$	MAG1 Empty
$t_{21}$	
$t_{22}$	↑ MAG1's Cylinder Extended
$t_{23}$	↑ MAG1's Cylinder Retracted
$t_{24}$	T=0.5s
$t_{25}$	↑ Presence T=0.5s
$t_{26}$	<u>Metallic Sensor</u>
$t_{27}$	<u>White Color Sensor</u>
$t_{28}$	↑ Proximity Sensor Left Discharge Cylinder
$t_{29}$	Right Discharge Cylinder Extended
$t_{30}$	Right Discharge Cylinder Retracted
$t_{31}$	White Color Sensor
$t_{32}$	↑ Proximity Sensor Center Discharge Cylinder
$t_{33}$	Center Discharge Cylinder Extended
$t_{34}$	Center Discharge Cylinder Retracted
$t_{35}$	Metallic Sensor
$t_{36}$	Concavity Downwards
$t_{37}$	↑ Proximity Sensor Left Discharge Cylinder
$t_{38}$	Left Discharge Cylinder Extended
$t_{39}$	Left Discharge Cylinder Retracted
$t_{40}$	

Continued on next page

Continued from previous page

Transitions	Meaning
$t_{41}$	Concavity Upwards
$t_{42}$	↑ Proximity Sensor End Of Conveyor Belt
$t_{43}$	T=0.5s
$t_{44}$	↓ Proximity Sensor End Of Conveyor Belt
$t_{45}$	

Table 4.4: Metal Half-cube Selection Module Places.

Places	Meaning
$p_{28}$	MAG1 Empty
$p_{29}$	MAG1 Not Empty
$p_{30}$	Extend MAG1's Cylinder *
$p_{31}$	Retract MAG1's Cylinder *
$p_{32}$	MAG1's Cylinder Retracted
$p_{33}$	Turn Conveyor Belt On
$p_{34}$	
$p_{35}$	Plastic Half-cube
$p_{36}$	Turn Conveyor Belt On
$p_{37}$	Extend Right Discharge Cylinder *
$p_{38}$	Retract Right Discharge Cylinder *
$p_{39}$	Turn Conveyor Belt On
$p_{40}$	Extend Center Discharge Cylinder *
$p_{41}$	Retract Center Discharge Cylinder *
$p_{42}$	
$p_{43}$	Metal Half-cube
$p_{44}$	Turn Conveyor Belt On
$p_{45}$	Extend Left Discharge Cylinder *
$p_{46}$	Retract Left Discharge Cylinder *
$p_{47}$	Turn Conveyor Belt On
$p_{48}$	Turn Conveyor Belt On
$p_{49}$	Metal Half-cube Ready
$p_{50}$	Conveyor Belt Stopped

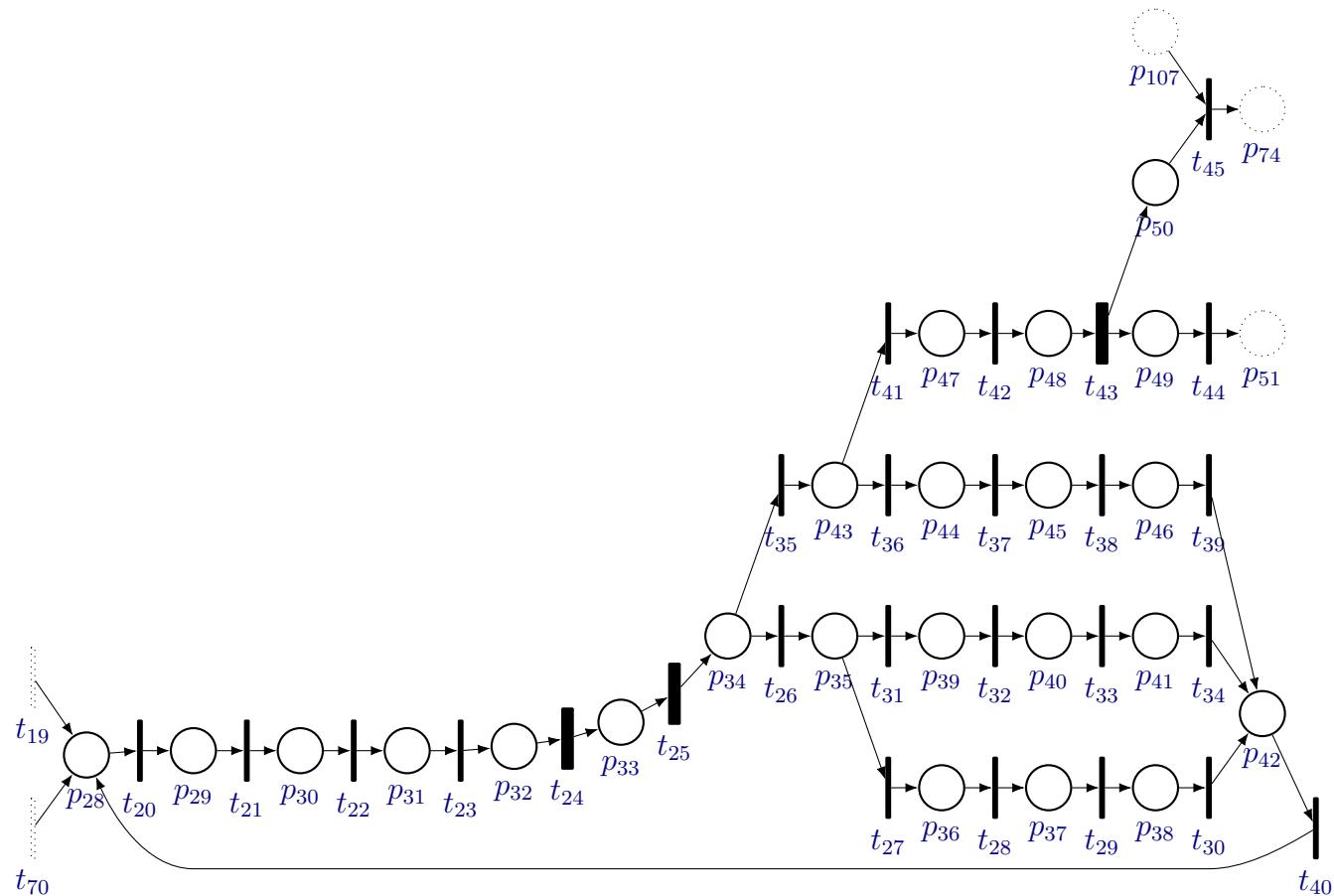


Figure 4.2: Petri net of metal cube half sorting module.

### 4.1.3 Plastic Cube Half Sorting

This module is similar to its metallic counterpart. This module sorts the cube halves stacked in Mag 2 and instead of metal pieces with upwards concavity, this module accepts white plastic pieces with downwards concavity. The corresponding Petri net and tables can be seen in [Figure 4.3](#) and Tables [4.5](#) and [4.6](#).

Table 4.5: Plastic Half-cube Selection Module Transitions.

Transitions	Meaning
$t_{46}$	MAG2 Empty
$t_{47}$	
$t_{48}$	↑ MAG2's Cylinder Extended
$t_{49}$	↑ MAG2's Cylinder Retracted
$t_{50}$	T=0.5s
$t_{51}$	↑ Presence T=0.5s
$t_{52}$	Metallic Sensor
$t_{53}$	↑ Proximity Sensor Left Discharge Cylinder
$t_{54}$	Left Discharge Cylinder Extended
$t_{55}$	Left Discharge Cylinder Retracted
$t_{56}$	Metallic Sensor
$t_{57}$	White Color Sensor
$t_{58}$	↑ Proximity Sensor Right Discharge Cylinder
$t_{59}$	Right Discharge Cylinder Extended
$t_{60}$	Right Discharge Cylinder Retracted
$t_{61}$	White Color Sensor
$t_{62}$	Concavity Upwards
$t_{63}$	↑ Proximity Sensor Center Discharge Cylinder
$t_{64}$	Center Discharge Cylinder Extended
$t_{65}$	Center Discharge Cylinder Retracted
$t_{66}$	
$t_{67}$	Concavity Downwards
$t_{68}$	↑ Proximity Sensor End Of Conveyor Belt
$t_{69}$	T=0.5s
$t_{70}$	↓ Proximity Sensor End Of Conveyor Belt
$t_{71}$	

Table 4.6: Plastic Half-cube Selection Module Places.

Places	Meaning
$p_{51}$	MAG2 Empty
$p_{52}$	MAG2 Not Empty
$p_{53}$	Extend MAG2's Cylinder *
$p_{54}$	Retract MAG2's Cylinder *
$p_{55}$	MAG2's Cylinder Retracted
$p_{56}$	Turn Conveyor Belt On
$p_{57}$	
$p_{58}$	Turn Conveyor Belt On
$p_{59}$	Extend Left Discharge Cylinder *
$p_{60}$	Retract Left Discharge Cylinder *
$p_{61}$	Metal Half-cube
$p_{62}$	Turn Conveyor Belt On
$p_{63}$	Extend Right Discharge Cylinder *
$p_{64}$	Retract Right Discharge Cylinder *
$p_{65}$	White Half-Cube
$p_{66}$	Turn Conveyor Belt On
$p_{67}$	Extend Center Discharge Cylinder *
$p_{68}$	Retract Center Discharge Cylinder *
$p_{69}$	
$p_{70}$	Turn Conveyor Belt On
$p_{71}$	Turn Conveyor Belt On
$p_{72}$	Plastic Half-cube Ready
$p_{73}$	Conveyor Belt Stopped

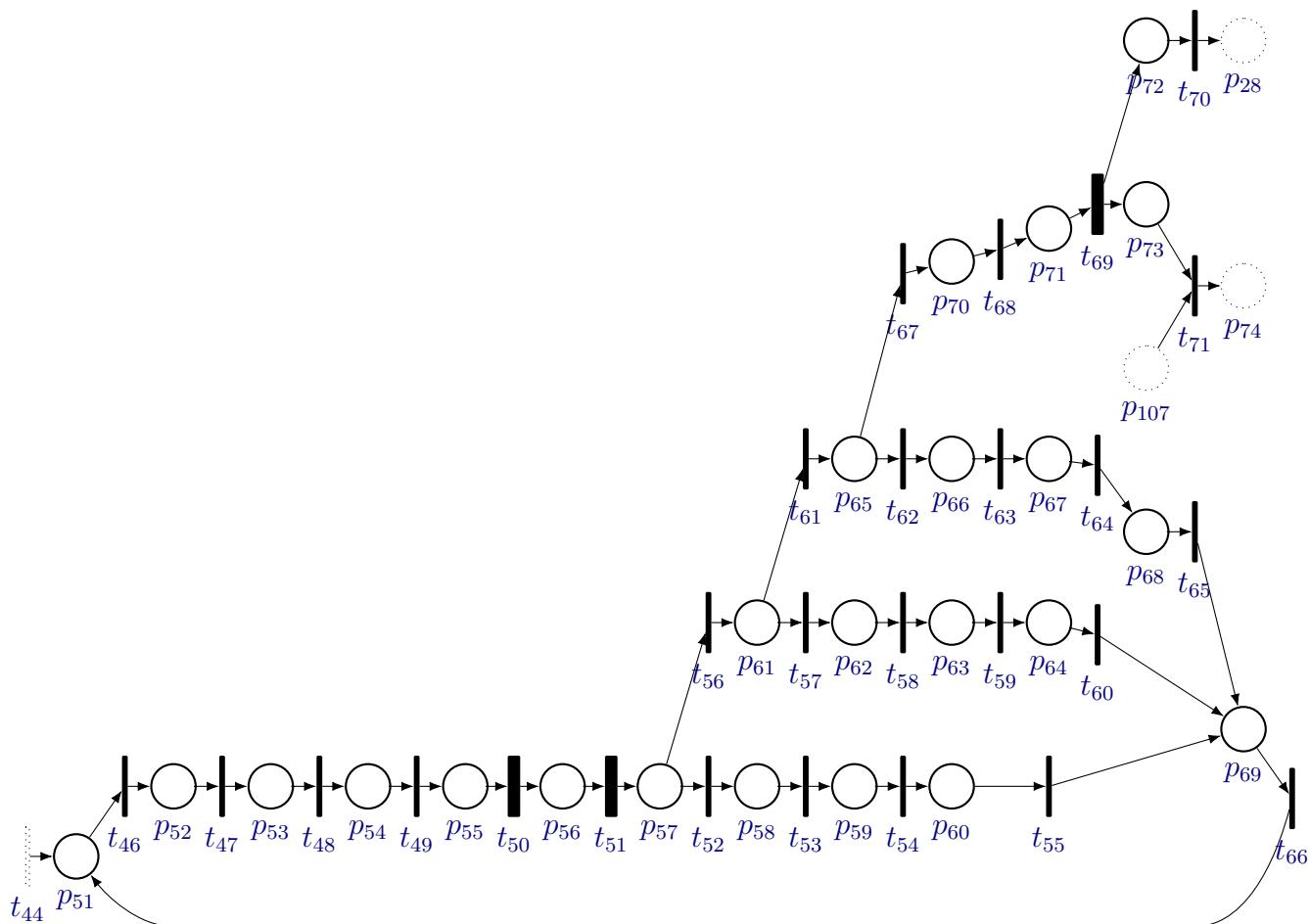


Figure 4.3: Petri net of plastic cube half sorting module.

#### 4.1.4 Arm From Conveyor Belt to Assembly Unit

This module uses the manipulator to take a piece present at the end of the conveyor belt and puts it in the assembly holder of the Assembly Unit. It places a metal piece and then a plastic piece, so they can be assembled to form a cube using the press. The corresponding Petri net and tables can be seen in Figure 4.4 and Tables 4.7 and 4.8.

Table 4.7: Arm From Conveyor Belt to Press Module Transitions.

Transitions	Meaning
$t_{72}$	Arm Raised
$t_{73}$	T=1.5s
$t_{74}$	T=1.5s and Arm Lowered
$t_{75}$	T=1.5s and Arm Raised
$t_{76}$	T=1.5s and Arm Raised
$t_{77}$	ARMCOUNTER <= PRESS_ANGLE
$t_{78}$	T=1.5s and Arm Raised
$t_{79}$	T=1.5s and Arm Lowered
$t_{80}$	T=1.5s
$t_{81}$	T=1.5s and Arm Raised
$t_{82}$	HALFPIECECOUNTER=1, Assembly Unit Holder Extended and Safety Door Opened
$t_{83}$	T=1.5s, HALFPIECECOUNTER=0 and Raised Arm
$t_{84}$	ARMCOUNTER >= BELT_ANGLE_CCW
$t_{85}$	

Table 4.8: Arm From Conveyor Belt to Press Module Places.

Places	Meaning
$p_{74}$	Raise Arm
$p_{75}$	Raise and Extend Arm, and Turn Vacuum On
$p_{76}$	Extend Arm and Turn Vacuum On
$p_{77}$	Raise and Extend Arm and Turn Vacuum On
$p_{78}$	Raise Arm and Turn Vacuum On
$p_{79}$	Raise Arm, Turn Vacuum On and Rotate Arm CW
$p_{80}$	Raise and Extend Arm and Turn Vacuum On
$p_{81}$	Extend Arm and Turn Vacuum On
$p_{82}$	Extend Arm
$p_{83}$	Raise and Extend Arm
$p_{84}$	Raise Arm
$p_{85}$	Raise Arm and Rotate Arm CCW
$p_{86}$	Raise Arm and HALFPIECE-COUNTER:=HALFPIECECOUNT+1

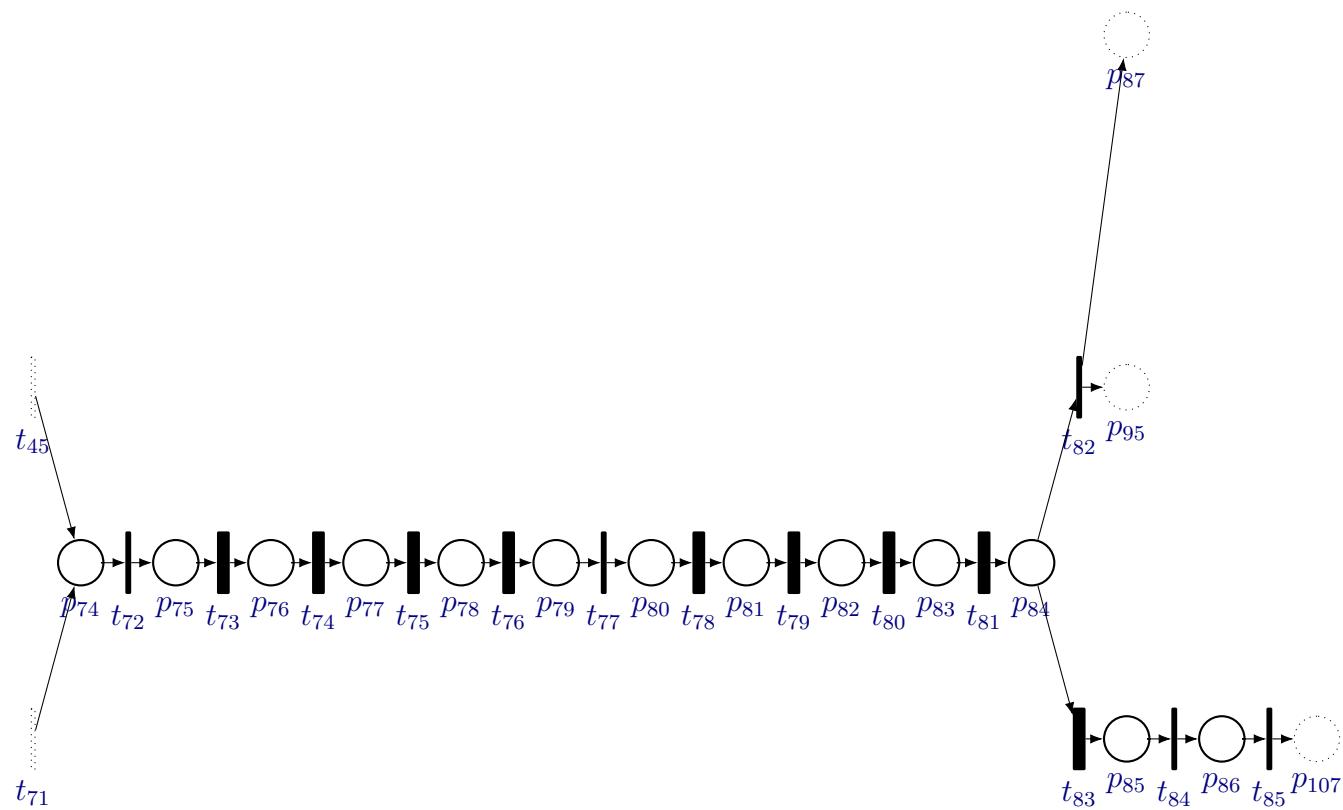


Figure 4.4: Petri net of manipulator taking a cube half from conveyor belt to assembly unit module.

#### 4.1.5 Assembly Unit

This module serves to press the two pieces, mounting a cube. Once both pieces are placed in the Assembly Unit Holder, it is retracted, the safety door is closed and the press is lowered, forming the cube. Then the press is raised, the door is opened, and the holder extended, waiting for the cube to be removed by the manipulator. The corresponding Petri net and tables can be seen in Figure 4.5 and Tables 4.9 and 4.10.

Table 4.9: Assembly Unit Module Transitions.

Transitions	Meaning
$t_{86}$	T=1s and Assembly Unit Holder Retracted
$t_{87}$	T=1s and Safety Door Closed
$t_{88}$	T=1s
$t_{89}$	T=1s
$t_{90}$	T=1s and Safety Door Opened
$t_{91}$	T=1s and Assembly Unit Holder Extended
$t_{92}$	
$t_{93}$	T=1.5s and Arm Extended

Table 4.10: Assembly Unit Module Places.

Places	Meaning
$p_{87}$	Retract Assembly Unit Holder *
$p_{88}$	Close Safety Door *
$p_{89}$	Lower Press *
$p_{90}$	Raise Press *
$p_{91}$	Open Safety Door *
$p_{92}$	Extend Assembly Unit Holder *
$p_{93}$	Cube Ready
$p_{94}$	Extend Arm and Turn Vacuum On
$p_{95}$	Raise and Extend Arm

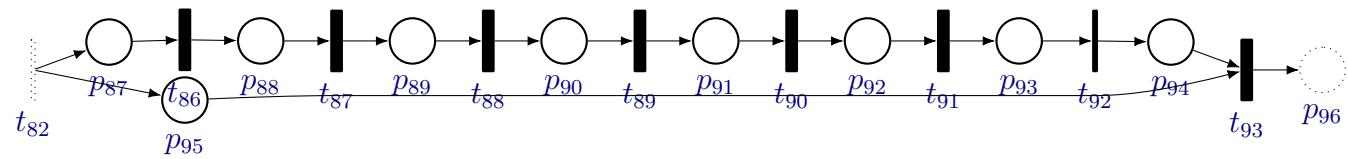


Figure 4.5: Petri net of assembly unit module.

#### 4.1.6 Arm From Assembly Unit To Storage Unit

This module uses the manipulator to move the cube from the Assembly Unit Holder to the storage device of the Storage Unit. Because of the way the system was assembled and the distances between the units, an additional encoder similar to **Storage Unit Horizontal Encoder** was placed just besides it, in order to help the alignment of the arm with the Storage Device, this encoder is called **Storage Unit Arm Alignment Encoder**, and is present in the logic of this module. The corresponding Petri net and tables can be seen in [Figure 4.5](#) and [Tables 4.11](#) and [4.12](#).

Table 4.11: Arm From Press To Storage Unit Module Transitions.

Transitions	Meaning
$t_{94}$	T=1.5s and Arm Lowered
$t_{95}$	Arm Raised, Storage Unit Right and Inferior Limit Switches
$t_{96}$	Storage Unit Arm Alignment Encoder
$t_{97}$	ARMCOUNTER $\leq$ STORAGE_ANGLE
$t_{98}$	T=2s
$t_{99}$	T=2s
$t_{100}$	Arm Lowered
$t_{101}$	Arm Raised, Storage Unit Right and Inferior Limit Switches
$t_{102}$	Inductive Sensor Arm
$t_{103}$	T=1s
$t_{104}$	ARMCOUNTER $\leq$ BELT_ANGLE_CW

Table 4.12: Arm From Press To Storage Unit Module Places.

Places	Meaning
$p_{96}$	Extend Arm e Turn Vacuum On
$p_{97}$	Raise and Extend Arm and Turn Vacuum On
$p_{98}$	Reset HALFPIECECOUNTER*, Raise and Extend Arm, Turn Vacuum On and Move Storage Unit to the Left
$p_{99}$	Raise and Extend Arm, Turn Vacuum On and Rotate Arm CW
$p_{100}$	Raise and Extend Arm and Turn Vacuum On
$p_{101}$	Extend Arm and Turn Vacuum On
$p_{102}$	Extend Arm
$p_{103}$	Raise and Extend Arm
$p_{104}$	Turn Arm CCW
$p_{105}$	Turn HSC Off ( Arm Stopped )
$p_{106}$	Turn Arm CW
$p_{107}$	Arm Stopped facing conveyor belt

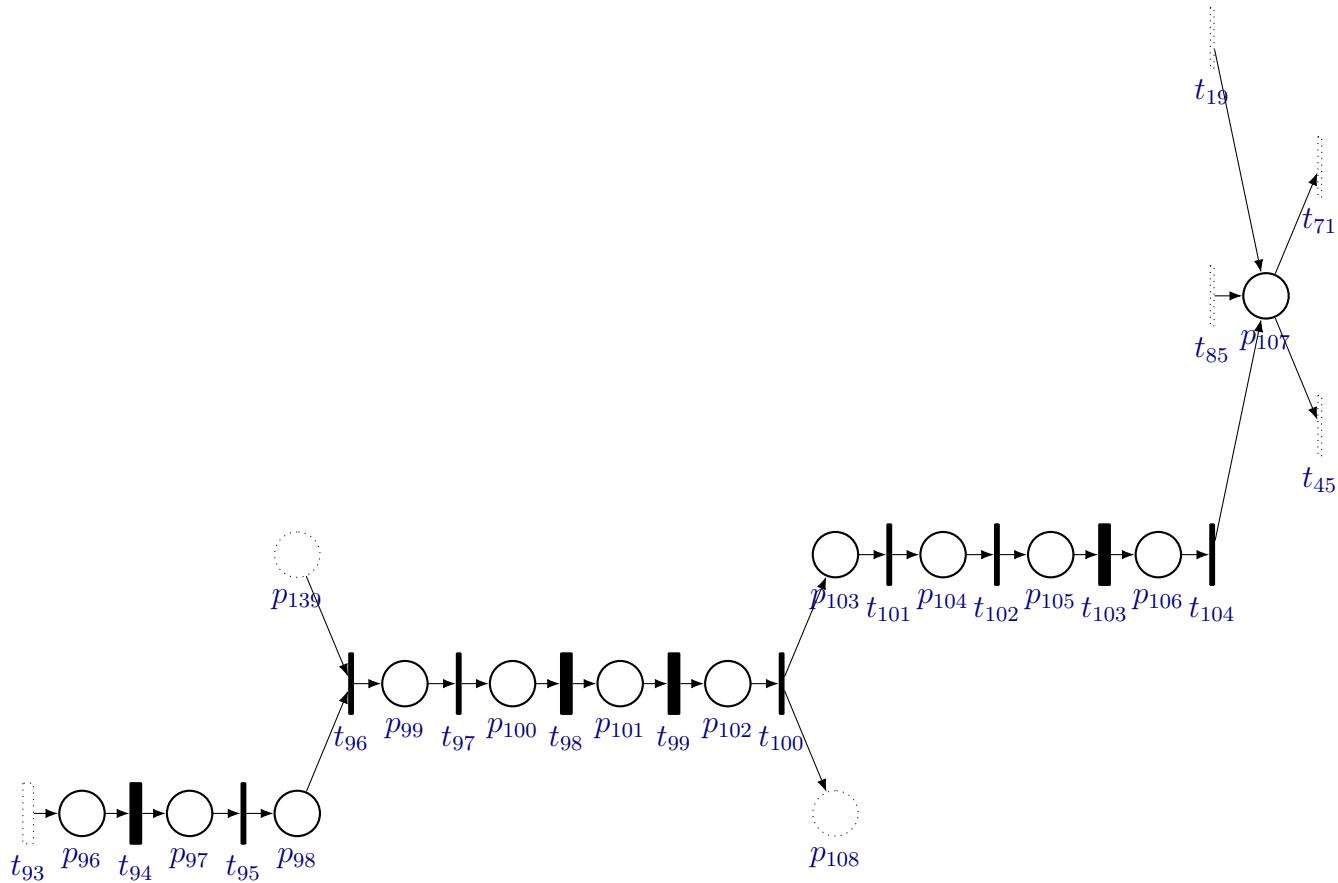


Figure 4.6: Petri net of manipulator taking cube from assembly unit to storage module.

#### 4.1.7 Storage Unit Positioning (y Axis)

This module sets the vertical position of the Storage Device. Once the cube is in the Storage Device, it is raised until the device is vertically aligned with corresponding store space. The order of storage in the rack is from top to bottom, right to left. The corresponding Petri net and tables can be seen in [Figure 4.7](#) and [Tables 4.13](#) and [4.14](#).

Table 4.13: Storage Unit (Y axis) Module Transitions.

Transitions	Meaning
$t_{105}$	T=2s
$t_{106}$	Storage Unit Right Limit Switch
$t_{107}$	COUNTER2=0
$t_{108}$	COUNTER3=4 and Vertical Encoder
$t_{109}$	COUNTER3<=4 and Vertical Encoder
$t_{110}$	COUNTER2=1
$t_{111}$	COUNTER3=3 and Vertical Encoder
$t_{112}$	COUNTER3<=3 and Vertical Encoder
$t_{113}$	COUNTER2=2
$t_{114}$	COUNTER3=2 and Vertical Encoder
$t_{115}$	COUNTER3<=2 and Vertical Encoder
$t_{116}$	COUNTER2=3
$t_{117}$	COUNTER3=1 and Vertical Encoder
$t_{118}$	COUNTER3<=1 and Vertical Encoder
$t_{119}$	Vertical Encoder
$t_{120}$	

Table 4.14: Storage Unit (Y axis) Module Places.

Places	Meaning
$p_{108}$	Cube on Storage Unit
$p_{109}$	Move Storage Unit to the Right
$p_{110}$	
$p_{111}$	Move Storage Unit Upwards
$p_{112}$	Move Storage Unit Upwards
$p_{113}$	Move Storage Unit Upwards
$p_{114}$	Move Storage Unit Upwards
$p_{115}$	COUNTER3:=COUNTER3+1
$p_{116}$	RESET COUNTER3*
$p_{117}$	

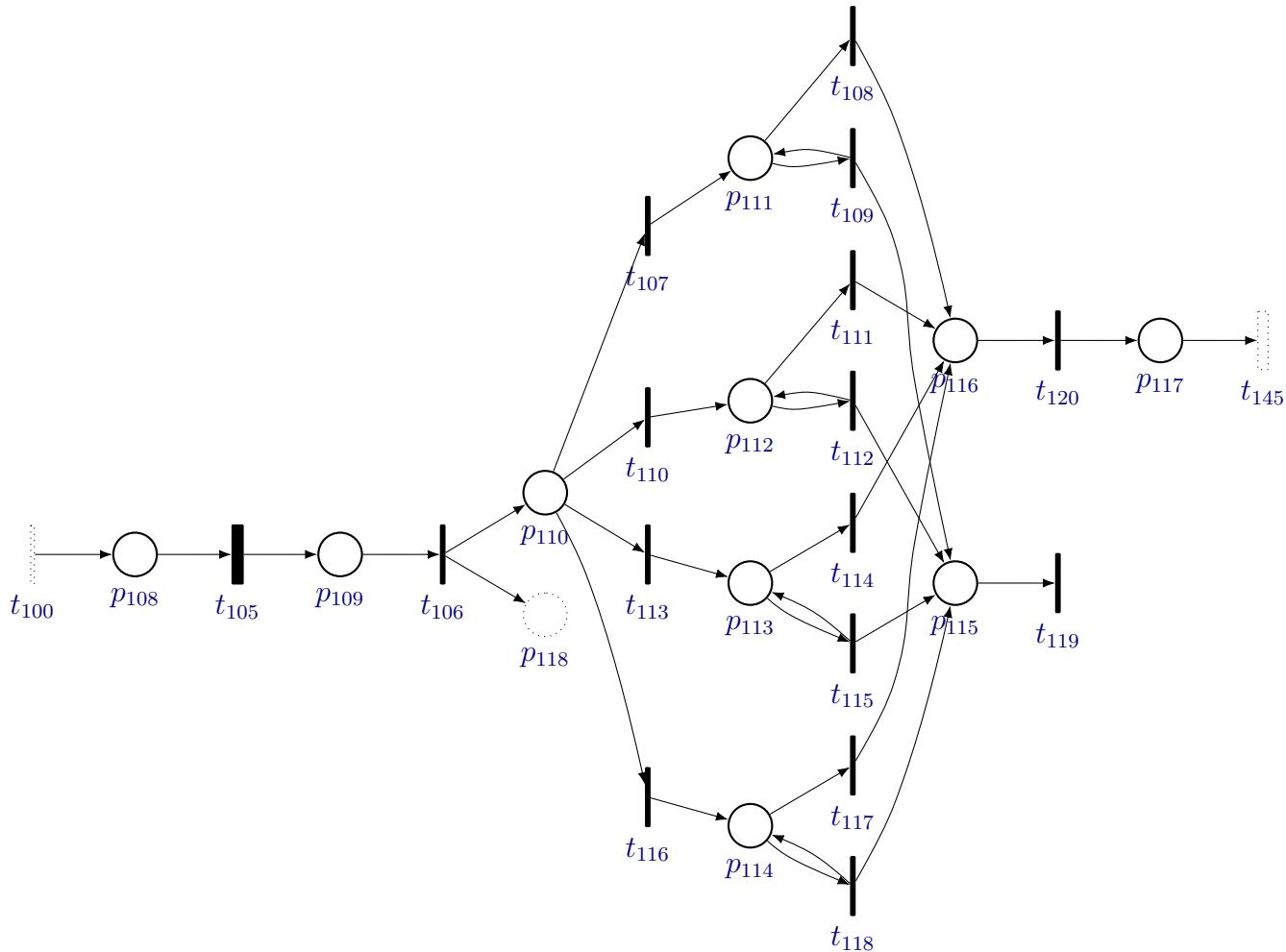


Figure 4.7: Petri net of storage unit positioning module (y axis).

#### 4.1.8 Storage Unit Positioning (x Axis)

This module sets the horizontal position of the Storage Device. It happens simultaneously with the last module. Instead of raising the Storage Device, this module makes it move from right to left until it is horizontally aligned with the corresponding store space. The corresponding Petri net and tables can be seen in Figure 4.8 and Tables 4.15 and 4.16.

Table 4.15: Storage Unit (X axis) Module Transitions.

Transitions	Meaning
$t_{121}$	COUNTER4=1
$t_{122}$	COUNTER5=1 and Horizontal Encoder
$t_{123}$	COUNTER5<=1 and Horizontal Encoder
$t_{124}$	COUNTER4=2
$t_{125}$	COUNTER5=2 and Horizontal Encoder
$t_{126}$	COUNTER5<=2 and Horizontal Encoder
$t_{127}$	COUNTER4=3
$t_{128}$	COUNTER5=3 and Horizontal Encoder
$t_{129}$	COUNTER5<=3 and Horizontal Encoder
$t_{130}$	COUNTER4=4
$t_{131}$	COUNTER5=4 and Horizontal Encoder
$t_{132}$	COUNTER5<=4 and Horizontal Encoder
$t_{133}$	COUNTER4=5
$t_{134}$	COUNTER5=5 and Horizontal Encoder
$t_{135}$	COUNTER5<=5 and Horizontal Encoder
$t_{136}$	COUNTER4=6
$t_{137}$	COUNTER5=6 and Horizontal Encoder
$t_{138}$	COUNTER5<=6 and Horizontal Encoder
$t_{139}$	COUNTER4=7
$t_{140}$	COUNTER5=7 and Horizontal Encoder
$t_{141}$	COUNTER5<=7 and Horizontal Encoder
$t_{142}$	
$t_{143}$	
$t_{144}$	Horizontal Encoder

Table 4.16: Storage Unit (X axis) Module Places.

Places	Meaning
$p_{118}$	COUNTER1:=COUNTER1+1 e
$p_{119}$	COUNTER4:=COUNTER4+1
$p_{120}$	Move Storage Unit to the Left
$p_{121}$	Move Storage Unit to the Left
$p_{122}$	Move Storage Unit to the Left
$p_{123}$	Move Storage Unit to the Left
$p_{124}$	Move Storage Unit to the Left
$p_{125}$	Move Storage Unit to the Left
$p_{126}$	COUNTER5:=COUNTER5+1
$p_{127}$	Reset COUNTER5*
$p_{128}$	Reset COUNTER4* , COUNTER2:=COUNTER2+1
$p_{129}$	

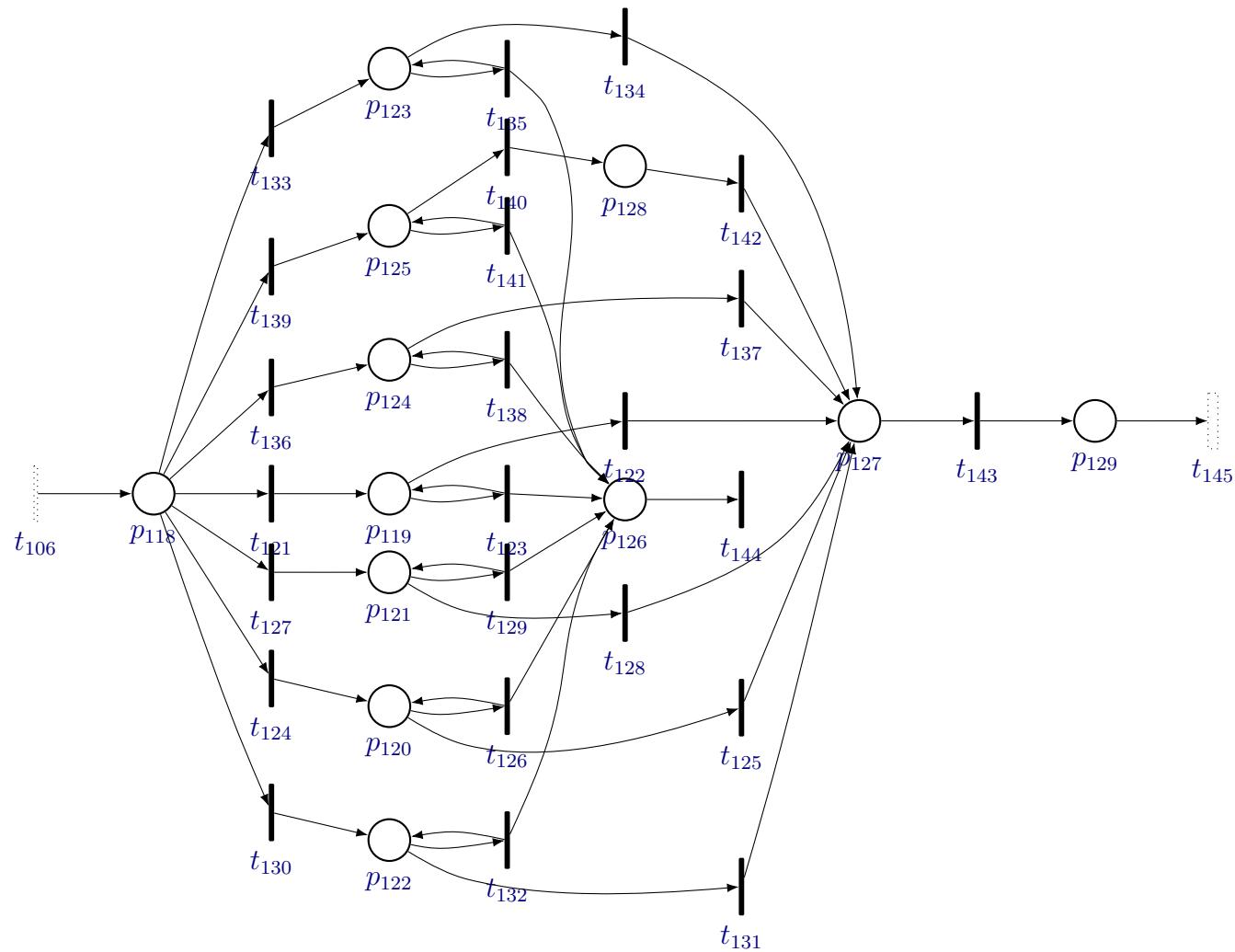


Figure 4.8: Petri net of storage unit positioning module (x axis).

#### 4.1.9 Cube Storage

This module has as objective to store the cube in the store space with which the storage device is vertically and horizontally aligned. The corresponding Petri net and tables can be seen in [Figure 4.9](#) and [Tables 4.17](#) and [4.18](#).

Table 4.17: Cube Storage Module Transitions.

Transitions	Meaning
$t_{145}$	T=2s
$t_{146}$	T=3s
$t_{147}$	T=0.25s
$t_{148}$	T=3s
$t_{149}$	T=7s
$t_{150}$	Storage Unit Right Limit Switch
$t_{151}$	Storage Unit Inferior Limit Switch
$t_{152}$	
$t_{153}$	COUNTER1<28
$t_{154}$	COUNTER1=28
$t_{155}$	COUNTER1=28

Table 4.18: Cube Storage Module Places.

Places	Meaning
$p_{130}$	Extend Storage Unit
$p_{131}$	Extend Storage Unit and Move Storage Unit Downwards
$p_{132}$	Extend Storage Unit
$p_{133}$	Piece Stored
$p_{134}$	Move Storage Unit to the Right
$p_{135}$	Storage Unit Ready ( horizontal )
$p_{136}$	Move Storage Unit Downwards
$p_{137}$	Storage Unit Ready ( vertical )
$p_{138}$	
$p_{139}$	Storage Unit Ready
$p_{140}$	Reset COUNTER1, COUNTER2, COUNTER3, COUNTER4 and COUNTER5*

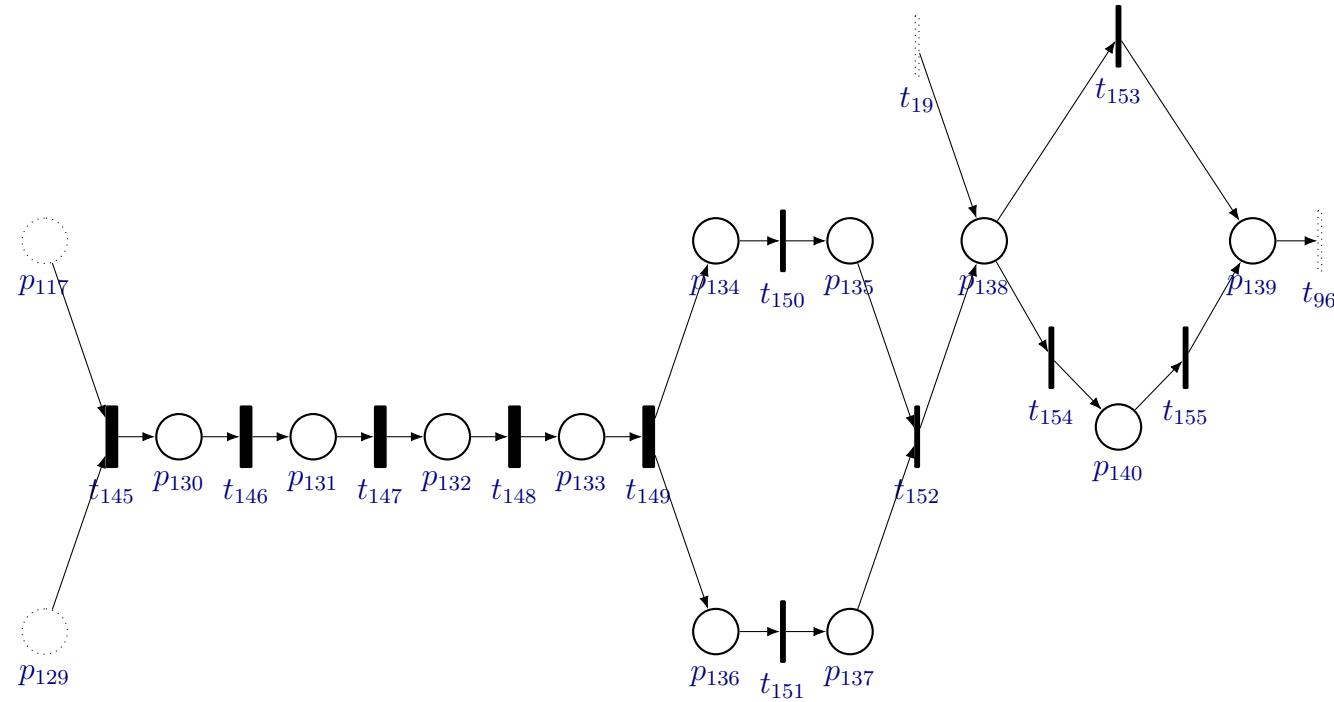


Figure 4.9: Petri net of cube storage module.

#### 4.1.10 Arm Stop Logic

Since the arm is controlled by *single acting pneumatic cylinders*, if in any moment the inputs of these cylinders is powered off voluntarily or not, they are going to retract, and because of the layout of the system, the way it was assembled, this can cause some incidents that can damage the arm. So, in order to prevent these kinds of accidents when the arm is powered off voluntarily, this module was created, turning the arm in a safe position before turning it off. With aim of stopping the system and consequentially the arm in any part of the process, a virtual button was created in the implementation phase. A transition with this button is created from every place in all other modules to the first place in this module, for organisation's sake all transitions are represented in the Petri net as the transition  $t_{156}$ .

Some angles were chosen to divide the areas where the arm can be, and the rotation logic is different for each one of them. These can be seen in Figure 4.10.

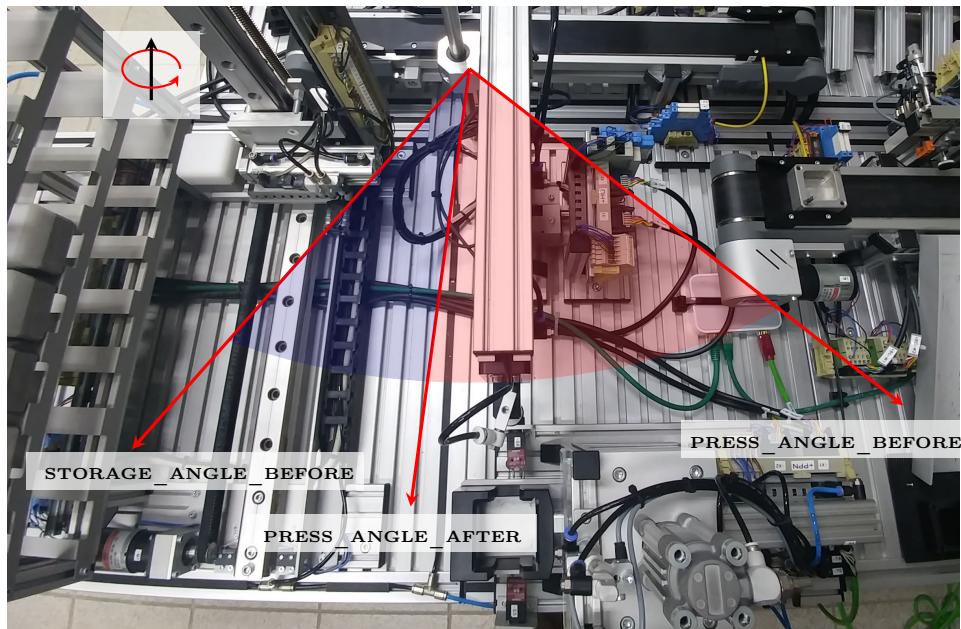


Figure 4.10: Arm Stop Logic Angles

The corresponding Petri net and tables can be seen in Figure 4.9 and Tables 4.17 and 4.18.

Table 4.19: Arm Stop Logic Module Transitions.

Transitions	Meaning
$t_{156}$	Stop Button
$t_{157}$	ARMCOUNTER < STORAGE_ANGLE_BEFORE
$t_{158}$	Arm Raised and Extended
$t_{159}$	ARMCOUNTER $\geq$ STORAGE_ANGLE_BEFORE (ARMCOUNTER $\geq$ STORAGE_ANGLE_BEFORE and ARMCOUNTER < PRESS_ANGLE_AFTER) or ARMCOUNTER $\geq$ PRESS_ANGLE_BEFORE
$t_{160}$	ARMCOUNTER $\geq$ PRESS_ANGLE_BEFORE
$t_{161}$	Arm Raised and Retracted
$t_{162}$	Inductive Sensor Arm
$t_{163}$	ARMCOUNTER $\geq$ PRESS_ANGLE_AFTER and ARMCOUNTER < PRESS_ANGLE_BEFORE
$t_{164}$	Arm Retracted
$t_{165}$	Arm Retracted

Table 4.20: Arm Stop Logic Module Places.

Places	Meaning
$p_{141}$	
$p_{142}$	Raise and Extend Arm
$p_{143}$	Raise, Extend Arm and Turn CCW
$p_{144}$	Raise Arm
$p_{145}$	Raise Arm and Turn CCW
$p_{146}$	

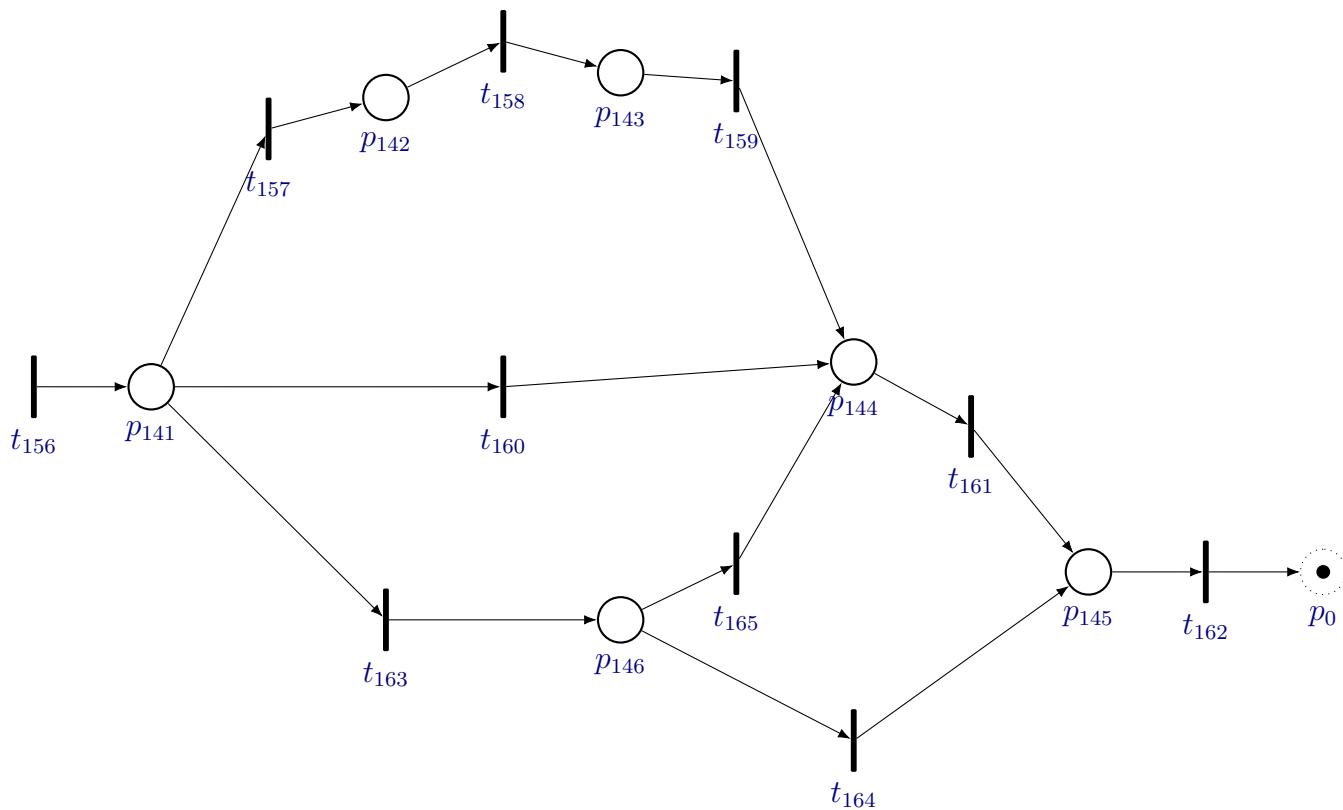


Figure 4.11: Petri net of manipulator Stop Logic module.

## 4.2 Implementation of the Control

As said in other chapters, the implementation of the control in this work be implemented using [PLCs](#). The units shown in [chapter 3](#) were divided in two parts and each part was connected to a Siemens [PLC](#) S7-1500, as the one shown in [Figure 4.12](#).



Figure 4.12: Siemens [PLC](#) S7-1500

The first [PLC](#) was connected with both magazines, the conveyor belt and the sorting unit. As those units are used to select the kind of pieces, this [PLC](#) is identified as Selection. In order to program the Ladder logic it is needed to create tags to represent every input and output, so in [Tables 4.21](#) and [4.22](#) we can see the correspondence between the name of the input/output, the address in which it is connected and the name of the tag created to represent it in the Ladder Logic.

Table 4.21: Inputs Selection PLC

Input	Address	Tag
MAG 1 Cylinder Extended	I0.0	I_MAG1EXT
MAG 1 Cylinder Retracted	I0.1	I_MAG1RET
MAG 1 Empty	I0.2	I_MAG1EMPT
MAG 2 Cylinder Extended	I0.3	I_MAG2EXT
MAG 2 Cylinder Retracted	I0.4	I_MAG2RET
MAG 2 Empty	I0.5	I_MAG2EMPT
Proximity Sensor Left Discharge Cylinder	I2.0	I_PSLD
Proximity Sensor Center Discharge Cylinder	I2.1	I_PSCD
Proximity Sensor Right Discharge Cylinder	I2.2	I_PSRD

Continued on next page

Continued from previous page

Input	Address	Tag
Relay	I2.3	I_RELAY1
Left Discharge Cylinder Extended	I1.0	I_LDCEXT
Left Discharge Cylinder Retracted	I1.1	I_LDCRET
Center Discharge Cylinder Extended	I1.2	I_CDCEXT
Center Discharge Cylinder Retracted	I1.3	I_CDCRET
Right Discharge Cylinder Extended	I1.4	I_RDCEXT
Right Discharge Cylinder Retracted	I1.5	I_RDCRET
White Color Sensor	I1.6	I_WHIT
Metallic Sensor	I1.7	I_METAL
Proximity Sensor End Of Conveyor Belt	I0.6	I_PSEOC
Distance Sensor	IW4	I_DS

Table 4.22: Outputs Selection PLC

Output	Address	Tag
Extend MAG 1 Cylinder	Q1.0	O_MAG1EXT
Retract MAG 1 Cylinder	Q1.1	O_MAG1RET
Extend MAG 2 Cylinder	Q1.2	O_MAG2EXT
Retract MAG 2 Cylinder	Q1.3	O_MAG2RET
Extend Right Discharge Cylinder	Q0.2	O_RDCEXT
Extend Center Discharge Cylinder	Q0.1	O_CDCEXT
Extend Left Discharge Cylinder	Q0.0	O_LDCEXT
Conveyor Belt Forward	Q1.4	O_CBFW
Conveyor Belt Reverse	Q1.5	O_CBREV

As said in [chapter 3](#), the `Distance Sensor` outputs an integer so the variables `ConcUP` and `ConcDWN` were created using the following comparisons:

$$\text{ConcUP} = \text{Distance Sensor} \geq 1000 \text{ & } \text{Distance Sensor} < 10000 \quad (4.1)$$

$$\text{ConcDWN} = \text{Distance Sensor} \geq 10000 \quad (4.2)$$

The other units (Handling Unit, Assembly Unit and Storage Unit) are connected to the second plc, identified as Handling-Assembly-Storage. The Tables [4.23](#) and [4.24](#) identify the addresses and tags for this PLC.

Table 4.23: Inputs Handling-Assembly-Storage PLC

Input	Address	Tag
Safety Door Opened	I1.0	I_SDO
Safety Door Closed	I1.1	I_SDC
Assembly Unit Holder Extended	I1.2	I_AUHEXT
Assembly Unit Holder Retracted	I1.3	I_AUHRET
Inductive Sensor Arm	I0.2	I_INDARM
Arm Lowered	I0.4	I_ARMLOW
Arm Raised	I0.3	I_ARMHIG
Arm Retracted	I0.6	I_ARMRET
Arm Extended	I0.5	I_ARMEXT
Storage Unit Vertical Encoder	I2.0	I_SUVE
Storage Unit Inferior Limit Switch	I2.2	I_SUILS
Storage Unit Superior Limit Switch	I2.1	I_SUSLS
Storage Unit Extended	I2.3	I_SUEXT
Storage Unit Retracted	I2.4	I_SURET
Relay	I2.5	I_RELAY2
Storage Unit Horizontal Encoder	I1.4	I_SUHE
Storage Unit Right Limit Switch	I1.5	I_SURLS
Storage Unit Left Limit Switch	I1.6	I_SULLS
Storage Unit Arm Alignment Encoder	I1.7	I_SUARMALE

Table 4.24: Outputs Handling-Assembly-Storage PLC

Output	Address	Tag
Open Safety Door	Q0.6	O_SDO
Close Safety Door	Q0.7	O_SDC
Retract Assembly Unit Holder	Q1.1	O_AUHRET
Extend Assembly Unit Holder	Q1.0	O_AUHEXT
Lower Press	Q1.2	O_PRESSLOW
Raise Press	Q1.3	O_PRESSTHIG
Raise Arm	Q0.0	O_ARMHIG
Turn Vacuum Gripper ON	Q0.1	O_VACON
Extend Arm	Q0.2	O_ARMEXT

Continued on next page

Continued from previous page

Output	Address	Tag
Turn Arm CCW	Q0.3	O_ARMCCW
Turn Arm CW	Q0.4	O_ARMCW
Extend Storage Unit	Q0.5	O_SUEXT
Move Storage Unit Upwards	Q1.6	O_SUUP
Move Storage Unit Downwards	Q1.7	O_SUDWN
Move Storage Unit to the Right	Q1.5	O_SURIGHT
Move Storage Unit to the Left	Q1.4	O_SULEFT

To convert the CIPN from the section 4.1 in to LD the method presented in subsection 2.6.2. And in order to implement the connection between the two PLCs the method shown in subsection 2.6.3 was used. In order to configure the “Get” and “Put” blocks and consequentially the connection between the two PLCs, the tutorials shown in section 3.4 of ROCHA PEREIRA (2019) was used.

For brevity’s sake the ladder logic was concealed, but can easily be found at the following link <https://github.com/Accacio/docsTCC/tree/master/PLC/TCC>, where all files of the TIA Project used in this work is stored.

# Chapter 5

## Identification

In this chapter, the identification process of the controlled system will be explained. This identification process can be divided in two parts: the data acquisition, where the inputs/outputs of the system are acquired, and the model identification, where the acquired data is used in the identification algorithm, [algorithm 1](#), and the identified model is generated.

In the next sections these two parts will be described

### 5.1 Data Acquisition

For the data acquisition, there are some ways to acquire the values of the inputs/outputs, but most of them are divided in two categories, one where the data is continuously registered, and the other one where the data is buffered and registered in batches from time to time. The first one is usually used for online processes, where the continuous flow of information is necessary, and processes that are repeated extensively, examples of these processes are control loops and failure detection modules. On the other hand, the second one usually serves for offline processes, processes that are computationally expensive or happen only once in a while. Modelling a big system and some more complex control loops can be examples of such processes.

Since the [algorithm 1](#) takes as its input a set of paths, all the data is acquired beforehand, a continuously acquisition is not necessary. The data can be acquired in batches, and once all the data is collected the algorithm can be run.

In order to acquire the data from a [PLC](#), the most straightforward way to do it is by using datalogs. The Siemens [PLC S7-1500](#) includes function blocks to use inside a [LD](#) to store custom data in a [Comma Separated Values .csv \(CSV\)](#) formatted file. This file

is saved in a SD card. To retrieve this file, the SD card can be connected to a PC, or it can be downloaded using a web browser if a Web Server is configured in the PLC.

The five blocks created to log data are called `DataLogCreate`, `DataLogOpen`, `DataLogWrite`, `DataLogClose` and `DataLogDelete`. They can be seen used in a LD in Figures 5.1, 5.2, 5.3, 5.4 and 5.5.

The `REQ` input in all blocks trigger the action to be performed in a datalog: create the datalog file, open it for writing, close it for writing, and deletes it, respectively.

The `ID` and `NAME` inputs are the way to identify the datalog file, they use a unique id number and a string. This string is used as the name of the .csv stored in the SD card.

The `DATA` input is a struct that contains the data to be stored in the .csv. This struct can contain variables of different types even of different sizes, for instance: it can contain a boolean, an int, a word, a string, and time.

The `HEADER` input is a string to be *prepended* to the first line of the .csv file, and it serves to identify each variable in the `DATA` input. As this string will be part of the .csv file, it is needed to include commas “,” inside it to separate the identifiers of the variables.

**N.B. 5.1** *As string type has variable size, it is important to take into account its maximum size, that is 256 Bytes, that means it can store up to 256 characters, considering the commas.*

The `TIMESTAMP` input is a boolean that if it is *true* the resulting .csv will have a timestamp column containing the time and date relative to the time when the data line was inserted.

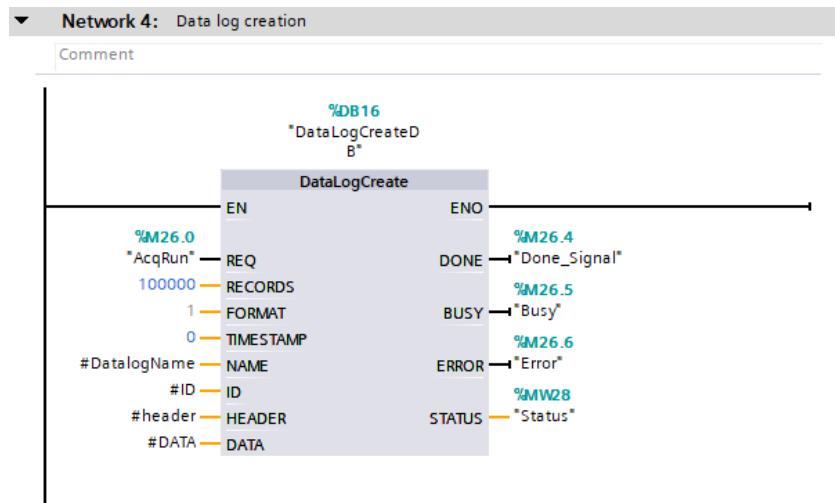


Figure 5.1: DataLogCreate block.

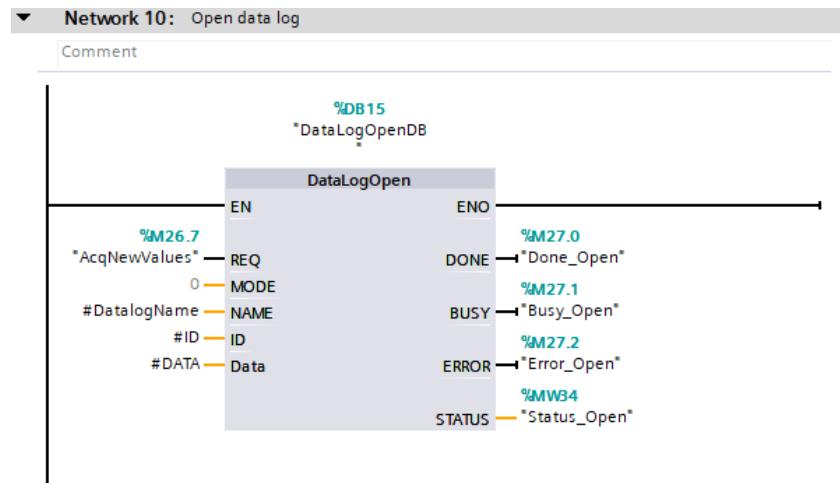


Figure 5.2: DataLogOpen block.

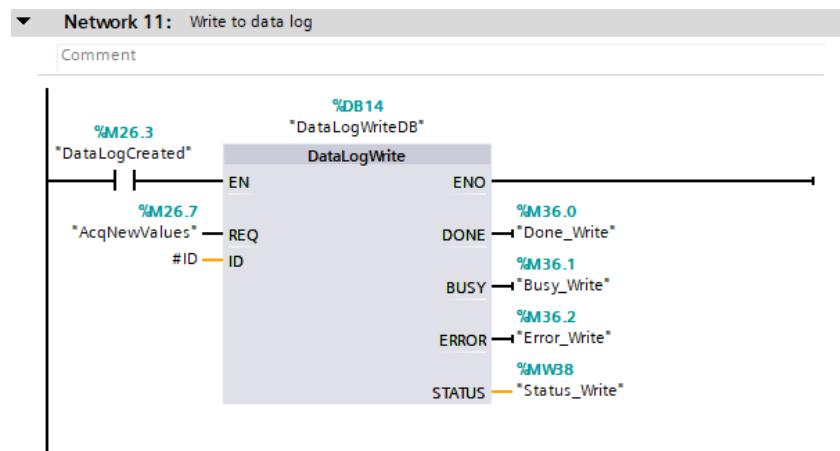


Figure 5.3: DataLogWrite block.

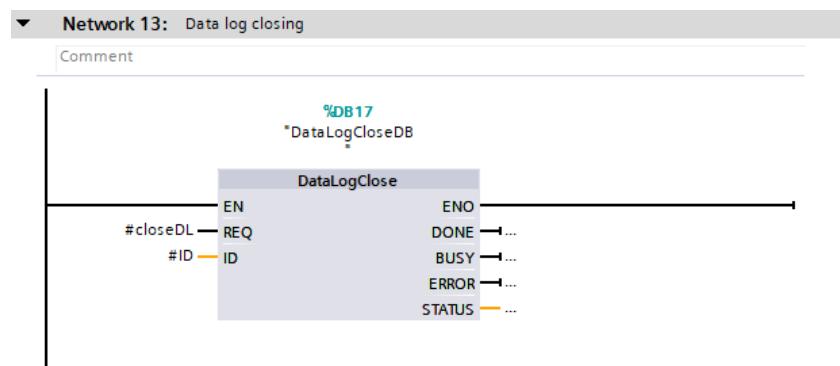


Figure 5.4: DataLogClose block.

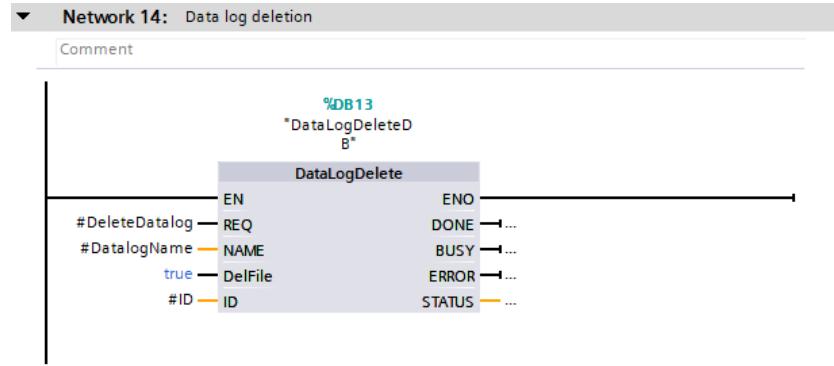


Figure 5.5: DataLogDelete block.

The outputs DONE,BUSY,ERROR and STATUS have as end to identify the status of each block, bu they were not used in this work.

As a way to organise the data used for all these blocks, a *DataBlock* was used. The Figure 5.6 show the *DataBlock* used and its structure. We can see in it the main variables used to create and write the log data: DATA, HEADER, ID and NAME.

More information about the blocks and its data can be seen in the Section 3.2 of SIEMENS (a).

	Name	Data type	Start value	Retain	Accessible f...	Writ...	Visible in ...	Setpoint	Supervision
1	Static			<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
2	IOVECNAME\$	Array[0..65] ...		<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
3	IOVEC	Array[0..64] of Bool		<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
4	AcqValues	Array[0..64] of Bool		<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
5	AcqValuesprev	Array[0..64] of Bool		<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
6	Data	Struct		<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
7	header	String	"	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
8	ID	DWord	16#0	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
9	Name	String	"	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	

Figure 5.6: Example of DataBlock used to log data.

Since some of the needs of the identification process are not satisfied by only using

the blocks themselves, as a way to implement these needs, a block was created that uses the datalog blocks together with others blocks, some of them are also user-defined.

The needs of the identification process is to store the input/output vectors of the controller, these being booleans, but store only when they change, so there are never two consecutive lines with the same *IOvector*. Another need is to start and stop easily the acquisition, and even delete the .csv file.

The block created to satisfy these needs is called *LOGDATA*. It can be seen in Figure 5.7 and it has 12 inputs. The **startAcq**, **stopAcq** and **DeleteDL** inputs serve to start and stop the acquisition and delete the .csv file, respectively. The **DatalogName** input serves to give the name of the file. **IOVECSIZE** tells the number of variables to be stored. As one of the needs is to store only *IOvectorss* different from its adjacent vectors, it is needed to store the last *IOvector* to compare it with a new one just acquired, so enter the two inputs **AcqValuesNew**, **AcqValuesPrev**. These two are arrays with size equivalent to the value input in **IOVECSIZE**. Their values are changed from inside the block via an update block. The **IOVEC** is also an input that is changed from inside the block and it is a temporary storage before it is sent to the variable connected to the **DATA** input, the latter input works exactly as the one shown in the original blocks, this variable that is effectively stored in the file.

Instead of writing the name of the variables in the **HEADER** input string, as in the original blocks, a **DATA\_HEADERS** input is created. This input is an array of size **IOVECSIZE** that contains the names of all variables to be stored in **DATA**. These names are concatenated with commas placed between them, and then this new string is assigned to the variable in put in **HEADER**, that is used as the header of the .csv. Remembering tha it is very important to take observation 5.1 on account.

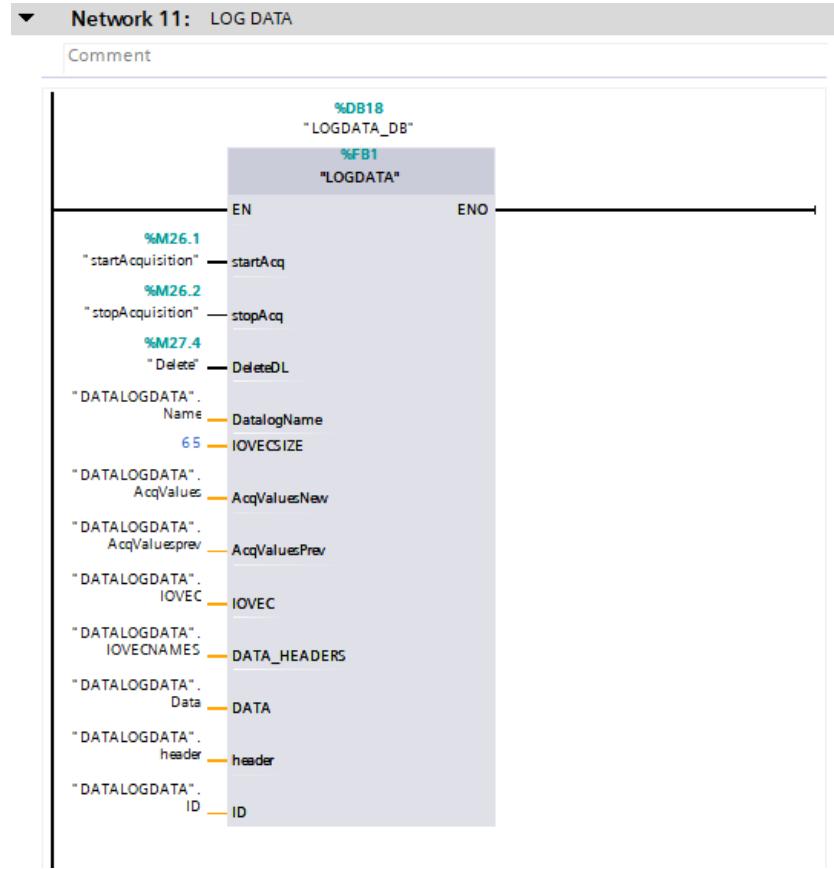


Figure 5.7: LOGDATA block.

An example of the data stored in this work can be seen in [Figure 5.8](#). As we can see the names of the variables inside the struct *Data* are the tags of the inputs and outputs of both [PLCs](#) shown in the [section 4.2](#).

TCC > Selection [CPU 1516-3 PN/DP] > Program blocks > Logging > DATALOGDATA [DB21]

	Name	Data type	Start value	Retain	Accessible f...	Write...	Visible in ...	Setpoint	Supervision
4	I_AcqValues	Array[0..64] of Bool		<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		
5	I_AcqValuesprev	Array[0..64] of Bool		<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		
6	I_Data	Struct		<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		
7	I_MAG1TEXT	Bool	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		
8	I_MAG1RET	Bool	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		
9	I_MAG1EMPT	Bool	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		
10	I_MAG2TEXT	Bool	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		
11	I_MAG2RET	Bool	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		
12	I_MAG2EMPT	Bool	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		
13	I_PSLD	Bool	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		
14	I_PSCD	Bool	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		
15	I_PSRD	Bool	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		
16	I_RELAY1	Bool	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		
17	I_LDCEXT	Bool	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		
18	I_LDCRET	Bool	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		
19	I_CDCEXT	Bool	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		
20	I_CDCRET	Bool	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		
21	I_RDCEXT	Bool	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		
22	I_RDCRET	Bool	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		
23	I_WHIT	Bool	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		
24	I_METAL	Bool	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		
25	I_PSEOC	Bool	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		
26	I_ConcDWN	Bool	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		
27	I_ConcUP	Bool	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		
28	O_MAG1TEXT	Bool	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		
29	O_MAG1RET	Bool	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		
30	O_MAG2TEXT	Bool	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		
31	O_MAG2RET	Bool	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		

Figure 5.8: Example of Data struct.

Inside this block some things are done in a certain order:

1. Copy AcqValuesNew to AcqValuesPrev
2. Update AcqValuesNew
3. Compare AcqValuesNew and AcqValuesPrev
4. prepare AcqValuesNew to be stored (If different from AcqValuesPrev)
5. Create and open datalog if asked
6. Write the data if asked
7. Close and delete datalog if asked

The last three steps are already depicted in Figures 5.1 to 5.5, the other ones will be shown in the next paragraphs.

To Update the values of `AcqValuesNew` the custom block called *UpdateValues* was created inside *LOGDATA*. This block can be seen in Figure 5.9, and in Figure 5.10 we can see the correspondence between the array and the tags.

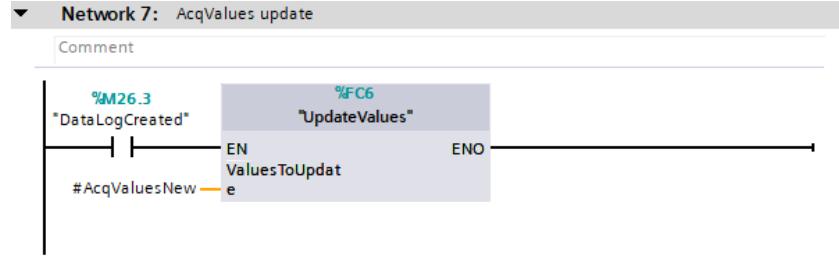


Figure 5.9: UpdateValues block.

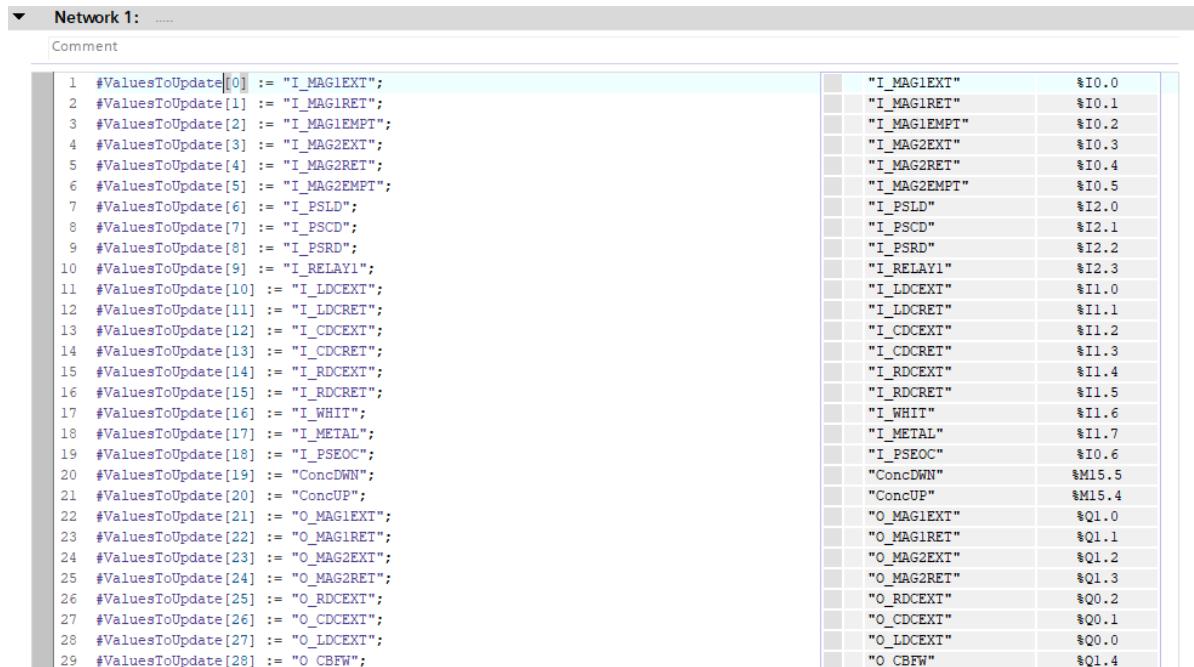


Figure 5.10: Code inside UpdateValues block.

In order to compare `AcqValuesNew` and `AcqValuesPrev`, a *CompareArrays* block was created where all the values are compared bitwise, since they are booleans, and if they are different the `AcqValuesNew` is copied to the temporary variable input to `Iovec`. This logic can be seen in [Figure 5.11](#).

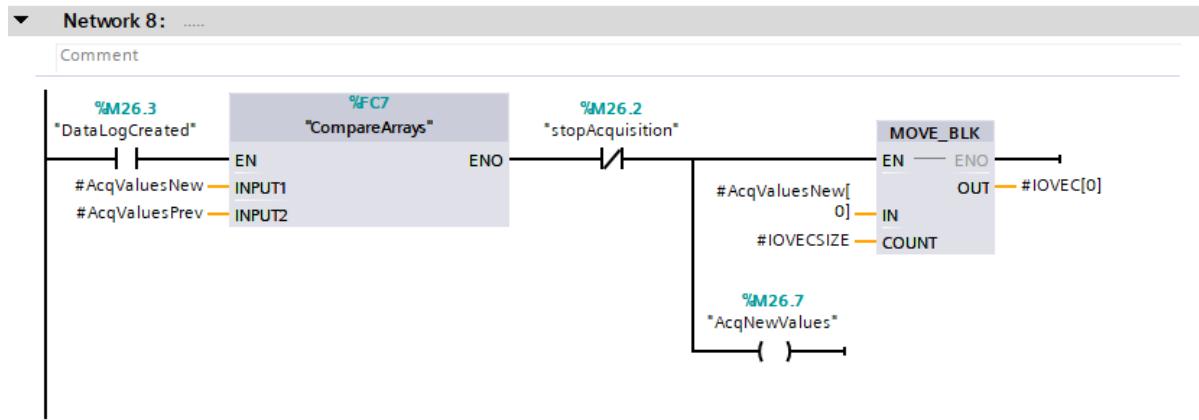


Figure 5.11: CompareArrays block.

And just before storing the data using the blocks, the data in the temporary variable **IOVEC** is copied to **DATA**, using the used defined block *PutInDataStruct* shown in Figure 5.12 its content can be seen in Figure 5.13.

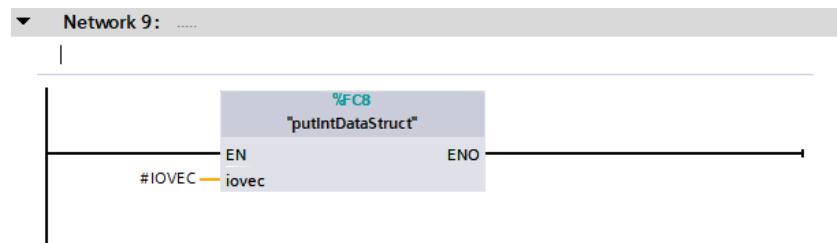


Figure 5.12: PutInDataStruct block.

Network 1: ....

Comment

```

1 "DATALOGDATA".Data.I_MAG1EXT := #iovec[0];
2 "DATALOGDATA".Data.I_MAG1RET := #iovec[1];
3 "DATALOGDATA".Data.I_MAG1EMPT := #iovec[2];
4 "DATALOGDATA".Data.I_MAG2EXT := #iovec[3];
5 "DATALOGDATA".Data.I_MAG2RET := #iovec[4];
6 "DATALOGDATA".Data.I_MAG2EMPT := #iovec[5];
7 "DATALOGDATA".Data.I_PSLD := #iovec[6];
8 "DATALOGDATA".Data.I_PSCD := #iovec[7];
9 "DATALOGDATA".Data.I_PSRD := #iovec[8];
10 "DATALOGDATA".Data.I_RELAY1 := #iovec[9];
11 "DATALOGDATA".Data.I_LDCEXT := #iovec[10];
12 "DATALOGDATA".Data.I_LDCRET := #iovec[11];
13 "DATALOGDATA".Data.I_CDCEXT := #iovec[12];
14 "DATALOGDATA".Data.I_CDCRET := #iovec[13];
15 "DATALOGDATA".Data.I_RDCEXT := #iovec[14];
16 "DATALOGDATA".Data.I_RDCRET := #iovec[15];
17 "DATALOGDATA".Data.I_WHIT := #iovec[16];
18 "DATALOGDATA".Data.I_METAL := #iovec[17];
19 "DATALOGDATA".Data.I_PSEOC := #iovec[18];
20 "DATALOGDATA".Data.ConcDN := #iovec[19];
21 "DATALOGDATA".Data.ConcUP := #iovec[20];
22 "DATALOGDATA".Data.O_MAG1EXT := #iovec[21];
23 "DATALOGDATA".Data.O_MAG1RET := #iovec[22];
24 "DATALOGDATA".Data.O_MAG2EXT := #iovec[23];
25 "DATALOGDATA".Data.O_MAG2RET := #iovec[24];
26 "DATALOGDATA".Data.O_RDCEXT := #iovec[25];
27 "DATALOGDATA".Data.O_CDCEXT := #iovec[26];

```

Figure 5.13: Code inside PutInDataStruct block.

**N.B. 5.2** Since the tags are divided between the 2 PLCs, in order to have all tags in a same PLC, “Get” and “Put” blocks were used along with a two Datablocks containing all the inputs and outputs of the Handling-Assembly-Storage PLC. The aspect of these datablocks can be seen in Figures 5.14a and 5.14b

TCC > Selection [CPU 1516-3 PN/DP] > Program blocks > Logging > IOVEC\_FROMPLC2 [DB19]

	Name	Data type	Offset	Start value	Retain	Accessible f...	Writab...	Visible in ...	Setpoint	Supervision	...
1	I_Static				<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
2	I_SDO	Bool	0.0	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
3	I_SDC	Bool	0.1	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
4	I_AUHEXT	Bool	0.2	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
5	I_AUHRET	Bool	0.3	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
6	I_INDARM	Bool	0.4	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
7	I_ARMLOW	Bool	0.5	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
8	I_ARMHIG	Bool	0.6	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
9	I_ARMRET	Bool	0.7	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
10	I_ARMEXT	Bool	1.0	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
11	I_SUVE	Bool	1.1	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
12	I_SUILS	Bool	1.2	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
13	I_SUSLS	Bool	1.3	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
14	I_SUEXT	Bool	1.4	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
15	I_SURET	Bool	1.5	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
16	I_RELAY2	Bool	1.6	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
17	I_SUHE	Bool	1.7	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
18	I_SURLS	Bool	2.0	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
19	I_SULLS	Bool	2.1	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
20	I_SUARMALE	Bool	2.2	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
21	O_SDO	Bool	2.3	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
22	O_SDC	Bool	2.4	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
23	O_AUHRET	Bool	2.5	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
24	O_AUHEXT	Bool	2.6	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
25	O_PRESSLOW	Bool	2.7	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
26	O_PRESHIGH	Bool	3.0	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
27	O_ARMHIG	Bool	3.1	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
28	O_VACON	Bool	3.2	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
29	O_ARMEXT	Bool	3.3	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
30	O_ARMCCW	Bool	3.4	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
31	O_ARMCW	Bool	3.5	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
32	O_SUEXT	Bool	3.6	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
33	O_SUUP	Bool	3.7	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
34	O_SUDWN	Bool	4.0	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
35	O_SURIGHT	Bool	4.1	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
36	O_SULEFT	Bool	4.2	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	

(a) IOVEC\_FROMPLC2 DataBlock.

TCC > Selection [CPU 1516-3 PN/DP] > Program blocks > Logging > IOVEC\_FROMPLC2 [DB19]

	Name	Data type	Offset	Start value	Retain	Accessible f...	Writab...	Visible in ...	Setpoint	Supervision	...
17	I_SUHE	Bool	1.7	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
18	I_SURLS	Bool	2.0	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
19	I_SULLS	Bool	2.1	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
20	I_SUARMALE	Bool	2.2	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
21	O_SDO	Bool	2.3	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
22	O_SDC	Bool	2.4	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
23	O_AUHRET	Bool	2.5	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
24	O_AUHEXT	Bool	2.6	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
25	O_PRESSLOW	Bool	2.7	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
26	O_PRESHIGH	Bool	3.0	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
27	O_ARMHIG	Bool	3.1	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
28	O_VACON	Bool	3.2	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
29	O_ARMEXT	Bool	3.3	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
30	O_ARMCCW	Bool	3.4	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
31	O_ARMCW	Bool	3.5	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
32	O_SUEXT	Bool	3.6	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
33	O_SUUP	Bool	3.7	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
34	O_SUDWN	Bool	4.0	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
35	O_SURIGHT	Bool	4.1	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
36	O_SULEFT	Bool	4.2	false	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	

(b) IOVEC\_FROMPLC2 DataBlock - Continuing.

Figure 5.14: Inputs/Outputs from Handling-Assembly-Storage PLC.

**N.B. 5.3** Since this is the first version of the block, that are still lots of improvements to be made to optimise its use. Some variables can be changed, in order to be used as internal variables, instead of creating new variables in a DataBlock. And maybe some variables and even rungs can be removed or substituted.

## 5.2 Model Identification

As said previously, once the data is logged, the .csv file can be downloaded from a Web Server, if this one is configured, or from the SD card. Simply following the steps shown in [SIEMENS \(b\)](#), it is possible to configure a web server in the Siemens PLCS7-1500. And once it is configured, the file can be downloaded in different ways, most of them are shown in section 3.13 of [SIEMENS \(b\)](#). Two of them are by clicking the download button shown in the DataLogs web page or by a bash script, as the following:

```
wget --content-disposition -i "http://192.168.2.132/DataLogs?Action=LIST".
```

The address “192.168.2.132” should be changed by the address of the [PLC](#) in which the web server is running.

When the .csv is already copied to a PC, the identification algorithm can be run. This identification algorithm, [algorithm 1](#), was implemented in python, but as the algorithm takes modified paths as input, the paths are needed to be extracted from the .csv and then modified using the Equations [2.3](#) and [2.4](#), also implemented in python. As the identification is made using a black box approach, we do not have any previous information of what is considered a path in the file, so the first vector is considered as the initial vector, and every time it is repeats another branch is created.

So, as an example, if we had the observed data of the example from [MOREIRA and LESAGE \(2018\)](#), the one shown in [section 2.7](#), in a .csv file, instead of 3 paths, as

shown in the example, 4 paths would be created, as follows:

$$\begin{aligned}
 p_1 &= \left( \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, a, \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}, b, \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}, c, \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, d, \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, e, \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \right) \\
 p_2 &= \left( \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, g, \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, h, \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, b, \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}, c, \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, i, \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \right) \\
 p_3 &= \left( \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, j, \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}, l, \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \right) \\
 p_4 &= \left( \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, g, \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, h, \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, b, \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, i, \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}, m, \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, d, \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, n, \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \right)
 \end{aligned}$$

If we compare these paths with those shown in [section 2.7](#), we can see that  $p_3$  of this section is part of  $p_2$  from [section 2.7](#). This change in number of paths is reflected on the identified model. Running the identification algorithm with the extracted paths and using the same parameters,  $k = 1$  and  $k = 2$ , it was possible to identify the state transition diagrams seen in Figures [5.15](#) and [5.16](#). Comparing them with Figures [2.28](#) and [2.29](#), we can see the difference caused by the additional path in the identified model. A brief discussion about the way the paths are chosen will be made in [chapter 6](#).

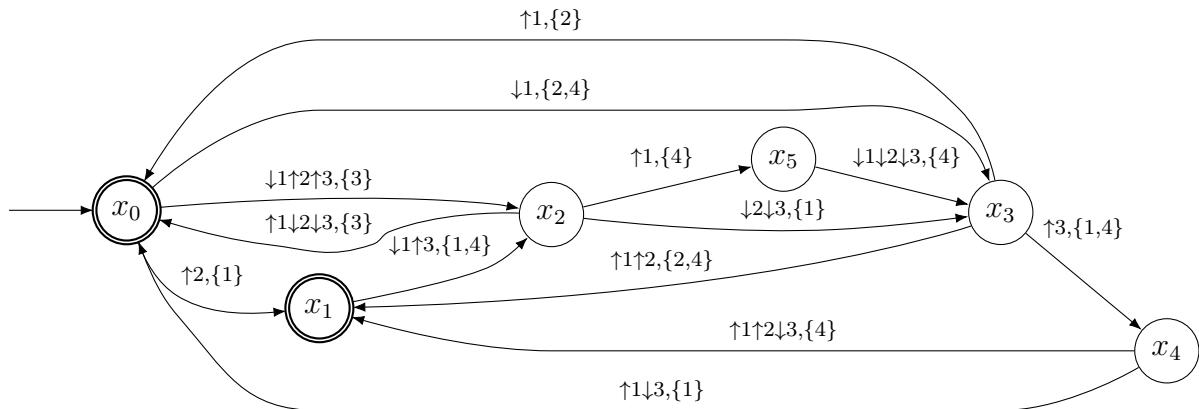


Figure 5.15: Identified model from paths extracted from `.csv` file using  $k = 1$ .

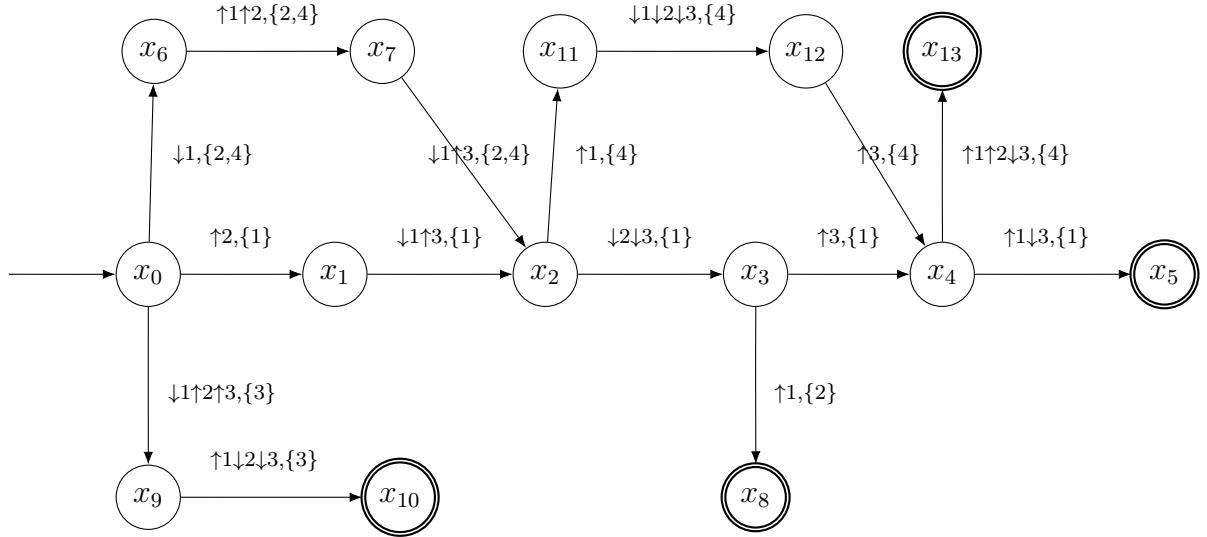


Figure 5.16: Identified model from paths extracted from .csv file using  $k = 2$ .

The tools created to implement the extraction of the paths and the identification algorithm will be commented in [section B.1](#), together with other useful tools used in this work.

# Chapter 6

## Results

In this chapter we will discuss the identified model generated by the algorithm 1 using different values of  $k$ , taking on account the number of paths extracted from the .csv file and the number of states generated by the identification algorithm.

### 6.1 Identified Model

As it was discussed in [chapter 2](#), in this work we assume that all sequences of events that have length  $n_0 + 1$  were observed, so  $L_{OrigNI}^{\leq n_0} = \emptyset$  can be true. In order to observe this sequences of events, the acquisition should be made for a sufficiently long time, thus the dataset was acquired from an experiment that lasted for 2 hours completing the cycle of filling the rack from top to bottom 4 times and a half. A time lapse of the process can be seen in <https://www.youtube.com/watch?v=ZtCCKJtA9pI>.

The acquisition started once the system was initialised, that means, when the system is ready to begin the process, this correspond to place  $p_{27}$  in [Figure 4.1](#).

The collected data<sup>1</sup> of this experiment that lasted for 2 hours has 19751 entries using 65 variables, the inputs/outputs of the system and the auxiliary variables ConcUP and ConcDWN.

Once this data was collected, the paths were extracted from the .csv and the result was kind of unexpected. The total number of paths using this raw data is equal to 2. This result can be easily explained by the behaviour of the system. As it is formed by different models, and some work simultaneously with others, the system has an important parallel behaviour. As these modules are not necessarily synchronised, it is very unlikely that

---

<sup>1</sup>Available at: [https://raw.githubusercontent.com/Accacio/docsTCC/master/data/2019-05-10/2019-05-10\\_1524.csv](https://raw.githubusercontent.com/Accacio/docsTCC/master/data/2019-05-10/2019-05-10_1524.csv)

all of them will return to their initial state at the same time, resulting in a few very long paths.

As expected, with a greater value of  $k$ , more states are identified, the Figure 6.1 show the variation of the number of states by changing the value of  $k$ .

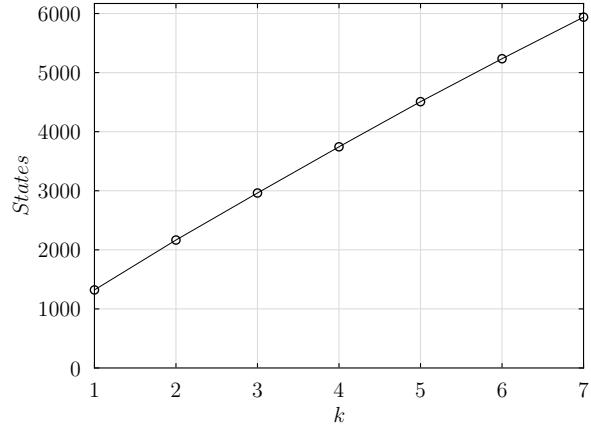


Figure 6.1: Number of states of identified model for different values of  $k$ .

As seen, the number of states for all values of  $k$  is greater than 1000. Models of this magnitude are very difficult to depict in state transition diagrams for two reasons; the legibility of such image would be terrible and the effort to draw it would be very time consuming (drawing it manually) or resource intensive (drawing using an automated tool). The automated tool known for drawing graphs has an upper bound around 350 nodes, so a tool was developed to automatically generate the list of  $f$  functions of the identified model.

The list of  $f$  functions generated using the original .csv file for  $k = 1$  and  $k = 2$  can be seen in <https://raw.githubusercontent.com/Accacio/docsTCC/master/figures/results/all/flistk1.tex> and <https://raw.githubusercontent.com/Accacio/docsTCC/master/figures/results/all/flistk2.tex> respectively.

We can also make a comparison between the exceeding language generated by the identified model using the DAOCT model and the Non-Deterministic Autonomous Automaton with Output (NDAAO) model, proposed by KLEIN *et al.* (2005). This comparison can show that even for a considerable large system with more than 60 inputs/outputs, the DAOCT is more tailored for fault detection, as the cardinality of its exceeding language is inferior to that of a NDAAO model of the same size (with a similar  $f$  function).

The Figure 6.2 shows this comparison using 2 values of  $k$ . In this case if we take, for example,  $k = 1$  and  $n = 12$  the exceeding language of NDAAO is 1018 and for DAOCT

it is 923. And if we take  $k = 2$ , both are 0 for  $n \leq 12$ . This mean in this case, with very long paths, both have a similar behaviour, but **DAOCT** still have a smaller exceeding language.

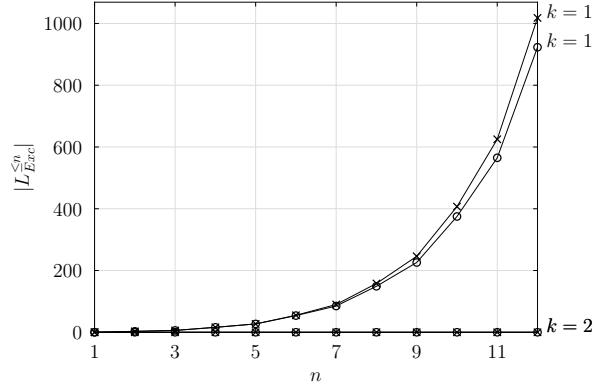


Figure 6.2: Comparison between the cardinality of the exceeding language generated by the DAOCT (o) and NDAAO (x) models, for  $k = 1$  and  $k = 2$ .

Since the original .csv file only could get only 2 paths extracted, an experiment was made in order to increase the number of paths and see how different the generated model would be.

The file was processed by a tool, where all vectors were sorted by the number of duplicates in the file. The vector with most duplicates was elected to be the new initial vector, consequentially the initial state of the new model.

A new .csv file was created from the original one. It was created by discarding all vectors from the beginning of the file up to the first appearance of the new initial vector, so this vector could be the first one to be processed in the path extraction.

Instead of the original 19751 entries, the new file had 19427 entries. This difference in number entries can be reflected in the number of generated states.

This new file, after the path extraction, resulted in 80 paths, 40 times the number of paths of the original.

Similarly to Figure 6.1, Figure 6.3 shows the variation of number of states with respect of the values of  $k$ . This figure can be used to see the relation between the change in the number of entries on the .csv file and the number of generated states.

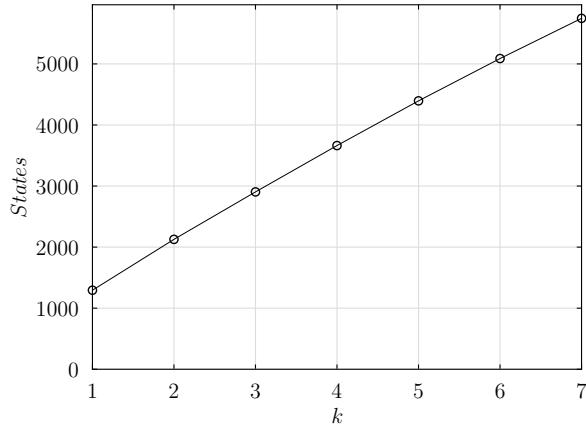


Figure 6.3: Number of states of identified model for different values of  $k$ .

Although both figures seem similar, both have the same order of magnitude for each  $k$ , the number of states diverges. If we put the values in a vector we can compare them. While for the original file the corresponding vector is [1321 2166 2962 3744 4508 5235 5939], for the modified is [1294 2127 2904 3663 4395 5088 5746]. This change can be caused, as said, by the difference in the number of entries in the .csv files.

Similarly, the list of  $f$  functions generated using the modified .csv file for  $k = 1$  and  $k = 2$  can be seen in <https://raw.githubusercontent.com/Accacio/docsTCC/master/figures/results/all/best/flistk1.tex> and <https://raw.githubusercontent.com/Accacio/docsTCC/master/figures/results/all/best/flistk2.tex> respectively.

The same comparison between the **DAOCT** and **NDAAO** was made and is shown in Figures 6.4a to 6.4c.

In this second case we can see that the difference between the language of the **DAOCT** and **NDAAO** models is more substantial. For example, for  $k = 1$  and  $n = 12$  the exceeding language of **NDAAO** is 465332 and for **DAOCT** it is 24866. For  $k = 2$  and  $n = 12$ , it is 1943 and 3, for **NDAAO** and **DAOCT** respectively. And if we take  $k = 3$  and  $n = 12$ , it is 712 for **NDAAO** and 0 **DAOCT**. With smaller and more numerous paths, we can see more clearly the difference between the exceeding language of the models. For instance if we want to detect correctly the failures of the system for sequences of length equal to or smaller than 12, using the **DAOCT** model it is needed only to use a  $k = 3$  while for the **NDAAO** it is needed to use a  $k$  greater than 7, for  $k = 7$  and  $n = 12$  the exceeding language of **NDAAO** is still equal to 47. Showing that **NDAAO** is more computationally expensive for the identification of a system.

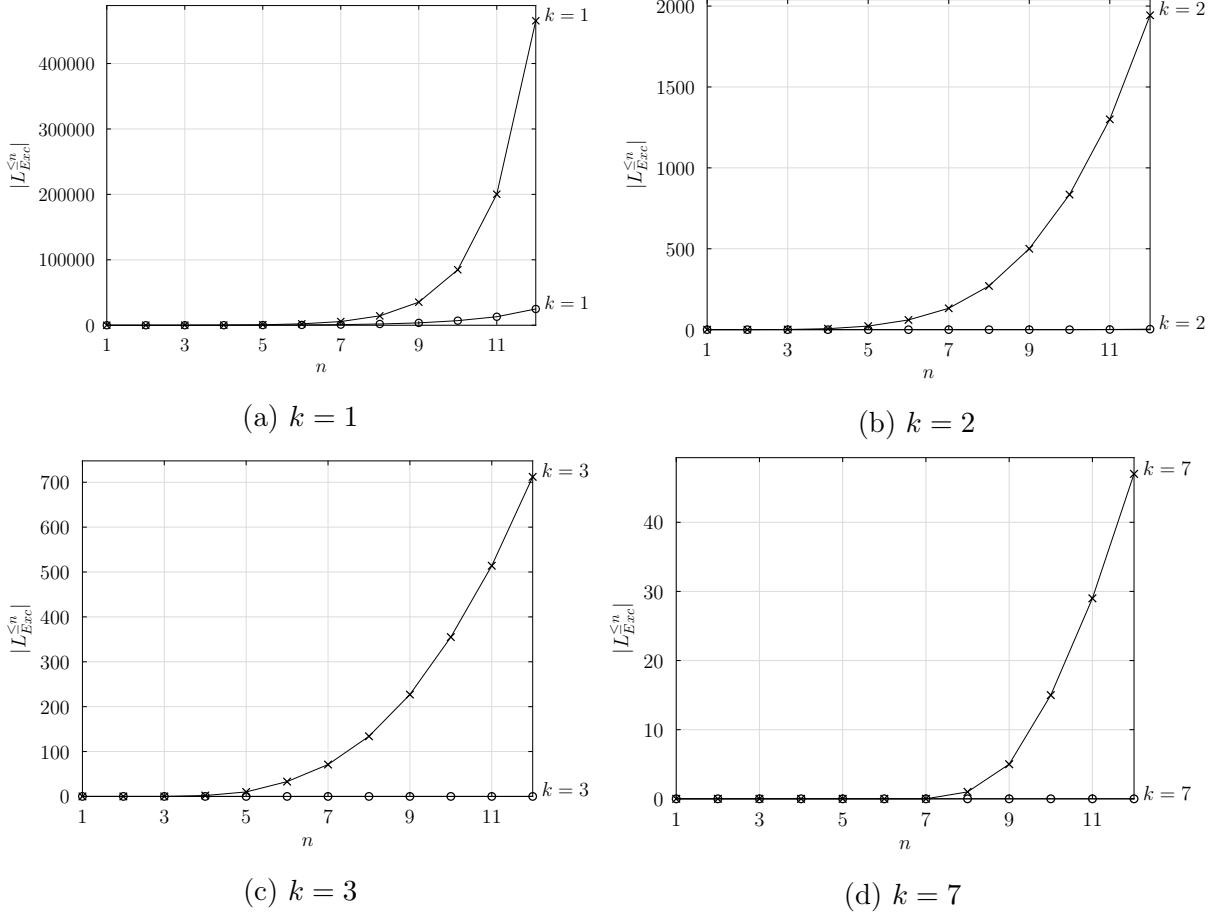


Figure 6.4: Comparison between the cardinality of the exceeding language generated by the DAOCT (o) and NDAAO (x) models.

Although the modified .csv generates more paths and shows a more considerable difference in the exceeding language generated by both models, the change of initial vector does not mean that it creates a more trustworthy model, that it represents better the system. The choice of this first vector of acquisition will be discussed in the next section in the form of an example.

## 6.2 About the choice of the first vector

Let us take as an example the following system:

### Example 6.1 (Conveyor Belt with 3 sensors)

*This simple system consists in a conveyor belt with three sensors  $S_1$ ,  $S_2$  and  $S_3$ . A scheme of the conveyor and its sensors can be seen in Figure 6.5. This conveyor is used to transport boxes, from the left to the right. The boxes are placed one at a time, so only a box can be over the conveyor, as it was a requisite for the design of the control. Once the box is over the conveyor and begin to be transported, it activates and deactivates  $S_1$ , then activates and deactivates  $S_2$  and finally activates and deactivates  $S_3$ . After  $S_3$  is deactivated another system know that the conveyor is empty and then it puts another box over the conveyor belt to restart its cycle. Since only a box is placed over the belt, it is impossible to 2 sensors to be activated the same time. And as the belt is always turned on, this system only has outputs (inputs to the controller), that are the signals of the three sensors  $S_1$ ,  $S_2$  and  $S_3$ .*

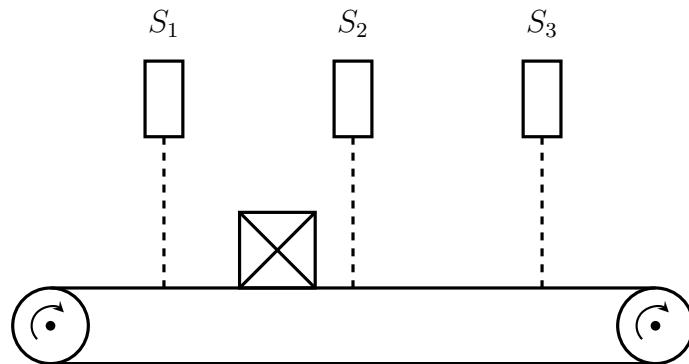


Figure 6.5: Scheme of the example 6.1.

If we make the data acquisition of this system and compose a vector with the values of  $S_1$ ,  $S_2$  and  $S_3$ , we will have the following repeating motif:

$$\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \quad (6.1)$$

This motif will be repeated multiple times on the .csv file forming cycles, and since it forms cycles the motif can be rewritten in how many ways as it has vertices, in this case it can be written in 6 ways. To reduce the complexity we will only discuss 2 ways of writing it, the first one shown in 6.1 and the second shown in 6.2. So, we can define two

datasets of acquisition, one beginning with  $[0 \ 0 \ 0]^T$  and other with  $[1 \ 0 \ 0]^T$ .

$$\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \quad (6.2)$$

If we take the first dataset, the one beginning with  $[0 \ 0 \ 0]^T$  and input it in the identification algorithm, the identified model, for  $k = 1$  can be seen in [Figure 6.6](#).

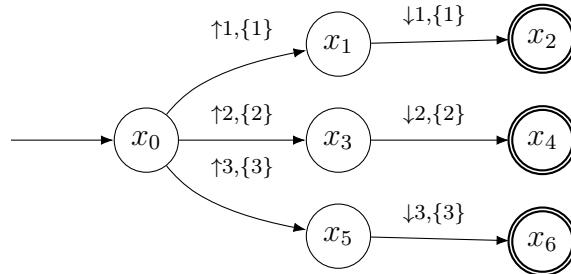


Figure 6.6: Identified model using  $[0 \ 0 \ 0]^T$  as initial state,  $k = 1$ .

As we can see in the arcs of the state transition diagram, three paths were extracted, this is caused by the way the motif is constructed. As  $[0 \ 0 \ 0]^T$  is considered the first vector and it repeats thrice throughout the motif, every time it is repeated another path is created.

But if we take the second dataset instead, the one beginning with  $[1 \ 0 \ 0]^T$  and input it in the identification algorithm, the identified model, for  $k = 1$  can be seen in [Figure 6.7](#).

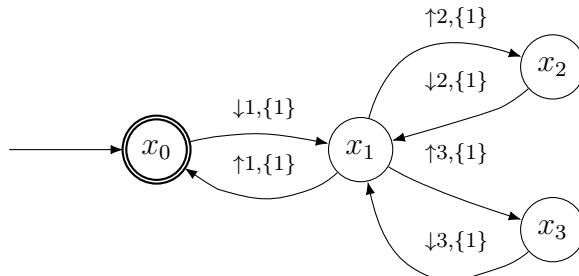


Figure 6.7: Identified model using  $[1 \ 0 \ 0]^T$  as initial state,  $k = 1$ .

As we can see only one path is created this time. In this figure we can see the representation of the vector  $[0 \ 0 \ 0]^T$  as a central hub, the state  $x_1$  in this state transition diagram, where all other states have arcs coming from or going to it. If we use a greater value of  $k$ , take  $k = 2$ , for instance, we can have a better vision of this unique path, see [Figure 6.8](#).

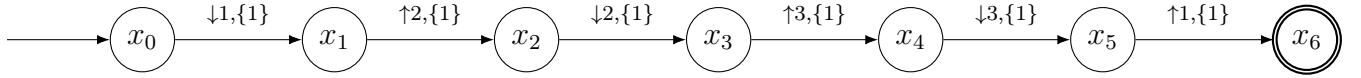


Figure 6.8: Identified model using  $[1 \ 0 \ 0]^T$  as initial state,  $k = 2$ .

In the first case, using  $[0 \ 0 \ 0]^T$  as the initial vector, two more paths were created when comparing with the second case, where  $[1 \ 0 \ 0]^T$  is used as the initial state. At a first glance it could seem that these 2 additional paths increase the information we have about the system, but actually, it does not. If we take the allowed sequences on this first case, we can see that the events  $\uparrow 2$  and  $\uparrow 3$  are allowed even before the event  $\uparrow 1$  is triggered, which is not part of the normal functioning of the system, described in example 6.1.

So, even with only one path, the second case, using  $[1 \ 0 \ 0]^T$  as initial state, represents better the system, since  $[1 \ 0 \ 0]^T$  happens only once in the motif making the extracted path to store almost all the sequence of events described in example 6.1.

As we can see the choice of the first vector plays a very important part on the extraction of paths and the assignment of the initial state of the identified system.

**N.B. 6.1** *An important remark to make is to show that we could only tell which identified system was more trustworthy because of the description of example 6.1. But once we have some information about the system, the approach ceases to be a black box one to be a grey box approach.*

This remark shows that the success of the DAOCT model is very dependent on a good choice of the initial vector, and consequentially the initial state of the system. But a question remains for a future work, how can we be sure if the initial state was well chosen once we do not know any of the system original behaviour? Maybe this kind of problem resides on the fact that input/output vectors are used to extract the paths and to create the events. An answer could be making it the other way around, developing an identification model that uses purely the events. The events are acquired in the first place and then they are used to extract the paths, and consequentially to identify the system.

# Chapter 7

## Conclusion

As proposed in the introduction, this work presents a methodology and tools for control, observation and identification of **DESs** in order to have a model to be used for fault detection. In this chapter a brief retrospective of the process of preparation of the methodology and the tools presented is made and the conclusions that can be drawn from the results generated by the application of the methodology to identify a didactic manufacture system.

Through the process of preparation of the methodology and tools, some issues were found and they are compiled in this chapter, together with some workarounds, that are proposed as new approaches to solve this issues in future works.

### 7.1 Concluding Remarks

So, in this work we could see the methodology to control, observe and identify a **DES**. First creating the control logic using a **CIPN**, and then implementing it in **LD** to be used in a Siemens **PLC** ([chapter 4](#)). After this, the observation of its inputs/outputs was made using Datalog blocks that saved the data in **.csv** files, and finally these **.csv** files were input in the identification algorithm generating a **DAOCT** model ([chapter 5](#)). In [chapter 6](#) we could see that if the system was observed for a long time and the initial state of observation was well chosen, then the **DAOCT** is a good candidate for modelling, if the aim of this modelling is fault-detection. The fact that the exceeding language of the **DAOCT** model drops to 0 more rapidly than other models, with a smaller value of the variable  $k$ , proves that it is less resource intensive than the others, even for relatively big systems, with more than 60 inputs/outputs with parallel behaviour.

## 7.2 Issues and Workarounds

An issue found in the implementation of the control is the use of LD to program the logic. Although LD is very used in the industry, as it is a visual language, it creates a difficulty for the automation of the conversion from Petri Net. An approach that can be used in future works would be to represent the Petri Net in a text format, petri net markup language for example (presented in WEBER and KINDLER (2003)), and create a tool that automatically converts this file to a text based language standardised by the IEC 61131-1, **Instruction List (IL)** or **ST**. Since IL is less and less used, ST would be the logical choice. Using a text based language increases portability of the code and it helps the development, since version control can be used in text files, allowing the collaboration of multiple people to edit the code if needed, and track who made the changes and when, increasing the maintainability of the code.

Another issue was about the observation. Although the acquisition of inputs/outputs using datalogs and saving the data in batches on .csv files can be used for the identification process, for fault-detection it is not optimal to use this approach, a better one would be to acquire the data in real time, by using some API, snap7 for example, or using **Supervisory Control and Data Acquisition (SCADA)** protocols. But if we use in a future work the function block created to log the data in [chapter 5](#), the *LOGDATA* block, it is recommended to optimise its contents. Some refactoring on the logic could be made, increasing its speed and removing some unnecessary variables that may be present.

As shown in [chapter 6](#), the didactic manufacture system used for the experiments have a considerable parallel behaviour, affecting the identified model, on the number of states and extracted paths. An approach that can be made in other works is to divide the observation of the system in its modules, and compare the multiple models generated by the identification algorithm with the one using the observation of the complete system.

Another issue shown in [chapter 6](#), is the choice of the first vector to be used as initial state in the identification algorithm. Here we propose for future works a study on how to find the optimal vector. Two scenarios could be considered: the first one taking a grey box approach, where some of the behaviour is previously known, by a simple description of the function of the system and another considering a black box approach with total ignorance of the behaviour of the system.

Another proposition for a future work is made in [chapter 6](#). Instead of using the observation of inputs/outputs of the system/controller to extract the paths and events of the system and to identify the system behaviour, a way out would be to observe the

events and use them to identify the system. Of course the DAOCT model would not be fit for it, and another model should be developed. This way, probably the problem with the choice of the initial state of the system would cease to exist.

# Bibliography

- ANTUNES FLORIANO, L. *Sincronização de Sistemas a Eventos Discretos Modelados por Redes de Petri Usando Lugares Comuns*. Undergraduate Project, Universidade Federal do Rio de Janeiro, 2019.
- CABRAL, F. G., MOREIRA, M. V. “Synchronous Codiagnosability of Modular Discrete-Event Systems”, *IFAC-PapersOnLine*, v. 50, n. 1, pp. 6831–6836, 2017.
- CASSANDRAS, C. G., LAFORTUNE, S. *Introduction to discrete event systems*. Springer Science & Business Media, 2009.
- DAVID, R., ALLA, H. L. *Du Grafet aux réseaux de Petri*. Hermes, 1989.
- DAVID, R., ALLA, H. *Discrete, continuous, and hybrid Petri nets*, v. 1. Springer, 2005.
- DAVIS, R., HAMSCHER, W. “Model-based reasoning: Troubleshooting”. In: *Exploring artificial intelligence*, Elsevier, pp. 297–346, 1988.
- KALOUPTSIDIS, N. *Signal processing systems: theory and design*, v. 28. Wiley-Interscience, 1997.
- KHALIL, W., DOMBRE, E. *Modeling, identification and control of robots*. Butterworth-Heinemann, 2004.
- KLEIN, S., LITZ, L., LESAGE, J.-J. “Fault detection of discrete event systems using an identification approach”, *IFAC Proceedings Volumes*, v. 38, n. 1, pp. 92–97, 2005.
- KUMAR, R., TAKAI, S. “Comments on “Polynomial Time Verification of Decentralized Diagnosability of Discrete Event Systems” versus “Decentralized Failure

Diagnosis of Discrete Event Systems”: Complexity Clarification”, *IEEE Transactions on Automatic Control*, v. 59, n. 5, pp. 1391–1392, 2014.

MOREIRA, M. V., BASILIO, J. C. “Bridging the gap between design and implementation of discrete-event controllers”, *IEEE Transactions on Automation Science and Engineering*, v. 11, n. 1, pp. 48–65, 2013.

MOREIRA, M. V., LESAGE, J.-J. “Enhanced Discrete Event Model for System Identification with the Aim of Fault Detection”, *IFAC-PapersOnLine*, v. 51, n. 7, pp. 160–166, 2018.

OLIVEIRA, V. D. S. L. *Protocolo de Comunicação Profinet para Redes de Automação*. Undergraduate Project, Universidade Federal do Rio de Janeiro, 2016.

OPPENHEIM, A. V., WILLSKY, A. S., NAWAB, S. “Signals and Systems (Prentice-Hall signal processing series)”, 1996.

PITANGA CLETO DE SOUZA, R. *Um Modelo Temporizado para a Identificação de Sistemas a Eventos Discretos*. Undergraduate Project, Universidade Federal do Rio de Janeiro, 2019.

RENAULT. *Règles de qualité logicielle pour la programmation des automatismes*. Technical report, Département Ingénierie Automatismes et Robotique. Groupe Renault, 2017. Available at: <[http://cnomo.com/fichiers/2/15981/Eb03j0010\\_A\\_fr%20.pdf?download=true](http://cnomo.com/fichiers/2/15981/Eb03j0010_A_fr%20.pdf?download=true)>.

ROCHA PEREIRA, A. P. *Automação de uma Planta Mecatrônica de Montagem e Armazenamento de Cubos Utilizando Comunicação entre Controladores Lógicos Programáveis*. Undergraduate Project, Universidade Federal do Rio de Janeiro, 2019.

ROTH, M., LESAGE, J.-J., LITZ, L. “An FDI method for manufacturing systems based on an identified model”, *IFAC Proceedings Volumes*, v. 42, n. 4, pp. 1406–1411, 2009.

SIEMENS. *S7-1500 Structure and Use of the CPU Memory*. SIEMENS, a. Available at: <[https://support.industry.siemens.com/cs/attachments/59193101/s71500\\_structure\\_and\\_use\\_of\\_the\\_PLC\\_memory\\_function\\_manual\\_en-US\\_en-US.pdf?download=true](https://support.industry.siemens.com/cs/attachments/59193101/s71500_structure_and_use_of_the_PLC_memory_function_manual_en-US_en-US.pdf?download=true)>.

SIEMENS. *S7-1500 Web server Function Manual*. SIEMENS, b. Available at: <[https://support.industry.siemens.com/cs/attachments/59193560/s71500\\_webserver\\_function\\_manual\\_en-US\\_en-US.pdf?download=true](https://support.industry.siemens.com/cs/attachments/59193560/s71500_webserver_function_manual_en-US_en-US.pdf?download=true)>.

VERAS, M. Z., CABRAL, F. G., MOREIRA, M. V. “Distributed Synchronous Diagnosability of Discrete-Event Systems”, *IFAC-PapersOnLine*, v. 51, n. 7, pp. 88–93, 2018.

WEBER, M., KINDLER, E. “The petri net markup language”. In: *Petri Net Technology for communication-based systems*, Springer, pp. 124–144, 2003.

# Appendix A

## Complete Petri Net

Table A.1: Complete Places.

Places	Meaning
$p_0$	System Stopped
$p_1, p_{31}$	Retract MAG1's Cylinder *
$p_2, p_{32}$	MAG1's Cylinder Retracted
$p_3, p_{54}$	Retract MAG2's Cylinder *
$p_4, p_{55}$	MAG2's Cylinder Retracted
$p_5, p_{38}, p_{64}$	Retract Right Discharge Cylinder *
$p_6$	Right Discharge Cylinder Retracted
$p_7$	Retract Center Discharge Cylinder
$p_8$	Center Discharge Cylinder Retracted
$p_9, p_{46}, p_{60}$	Retract Left Discharge Cylinder *
$p_{10}$	Left Discharge Cylinder Retracted
$p_{11}$	Turn Conveyor Belt On (Reverse)
$p_{12}$	No Pieces On Conveyor Belt
$p_{13}$	Reset Variables
$p_{14}$	Raise Press
$p_{15}$	Open Safety Door
$p_{16}$	Extend Assembly Unit Holder
$p_{17}$	Assembly Unit Ready
$p_{18}$	Arm Lowered and Retracted, and Storage Unit Retracted

Continued on next page

Continued from previous page

Places	Meaning
$p_{19}, p_{109}, p_{134}$	Move Storage Unit to the Right
$p_{20}$	Storage Unit ready ( horizontal )
$p_{21}$	Move Storage Device Downwards
$p_{22}$	Storage Unit ready ( vertical )
$p_{23}$	Rotate Arm CCW
$p_{24}, p_{105}$	Turn HSC Off ( Arm Stopped )
$p_{25}$	Rotate Arm CW
$p_{26}, p_{107}$	Arm Stopped facing conveyor belt
$p_{27}$	System Ready
$p_{28}$	MAG1 Empty
$p_{29}$	MAG1 Not Empty
$p_{30}$	Extend MAG1's Cylinder *
$p_{33}, p_{36}, p_{39}, p_{44}, p_{47}, p_{48}, p_{56}, p_{58}, p_{62}, p_{66}, p_{70}, p_{71}$	Turn Conveyor Belt On
$p_{34}, p_{42}, p_{57}, p_{69}, p_{110}, p_{117}, p_{129}, p_{138}, p_{141}, p_{146}$	
$p_{35}$	Plastic Half-cube
$p_{37}, p_{63}$	Extend Right Discharge Cylinder *
$p_{40}, p_{67}$	Extend Center Discharge Cylinder *
$p_{41}, p_{68}$	Retract Center Discharge Cylinder *
$p_{43}, p_{61}$	Metal Half-cube
$p_{45}, p_{59}$	Extend Left Discharge Cylinder *
$p_{49}$	Metal Half-cube Ready
$p_{50}, p_{73}$	Conveyor Belt Stopped
$p_{51}$	MAG2 Empty
$p_{52}$	MAG2 Not Empty
$p_{53}$	Extend MAG2's Cylinder *
$p_{65}$	White Half-Cube
$p_{72}$	Plastic Half-cube Ready
$p_{74}, p_{84}, p_{144}$	Raise Arm
$p_{75}$	Raise and Extend Arm, and Turn Vacuum On
$p_{76}, p_{81}, p_{94}, p_{101}$	Extend Arm and Turn Vacuum On

Continued on next page

Continued from previous page

Places	Meaning
<i>p<sub>77</sub>, p<sub>80</sub>, p<sub>97</sub>, p<sub>100</sub></i>	Raise and Extend Arm and Turn Vacuum On
<i>p<sub>78</sub></i>	Raise Arm and Turn Vacuum On
<i>p<sub>79</sub></i>	Raise Arm, Turn Vacuum On and Rotate Arm CW
<i>p<sub>82</sub>, p<sub>102</sub></i>	Extend Arm
<i>p<sub>83</sub>, p<sub>95</sub>, p<sub>103</sub>, p<sub>142</sub></i>	Raise and Extend Arm
<i>p<sub>85</sub></i>	Raise Arm and Rotate Arm CCW
<i>p<sub>86</sub></i>	Raise Arm and HALFPIECE-COUNTER:=HALFPIECECOUNTER+1
<i>p<sub>87</sub></i>	Retract Assembly Unit Holder *
<i>p<sub>88</sub></i>	Close Safety Door *
<i>p<sub>89</sub></i>	Lower Press *
<i>p<sub>90</sub></i>	Raise Press *
<i>p<sub>91</sub></i>	Open Safety Door *
<i>p<sub>92</sub></i>	Extend Assembly Unit Holder *
<i>p<sub>93</sub></i>	Cube Ready
<i>p<sub>98</sub></i>	Reset HALFPIECECOUNTER*, Raise and Extend Arm, Turn Vacuum On and Move Storage Unit to the Left
<i>p<sub>99</sub></i>	Raise and Extend Arm, Turn Vacuum On and Rotate Arm CW
<i>p<sub>104</sub></i>	Turn Arm CCW
<i>p<sub>106</sub></i>	Turn Arm CW
<i>p<sub>108</sub></i>	Cube on Storage Unit
<i>p<sub>111</sub>, p<sub>112</sub>, p<sub>113</sub>, p<sub>114</sub></i>	Move Storage Unit Upwards
<i>p<sub>115</sub></i>	COUNTER3:=COUNTER3+1
<i>p<sub>116</sub></i>	RESET COUNTER3*
<i>p<sub>118</sub></i>	COUNTER1:=COUNTER1+1 e COUNTER4:=COUNTER4+1
<i>p<sub>119</sub>, p<sub>120</sub>, p<sub>121</sub>, p<sub>122</sub>, p<sub>123</sub>, p<sub>124</sub>, p<sub>125</sub></i>	Move Storage Unit to the Left
<i>p<sub>126</sub></i>	COUNTER5:=COUNTER5+1
<i>p<sub>127</sub></i>	Reset COUNTER5*

Continued on next page

Continued from previous page

Places	Meaning
$p_{128}$	Reset COUNTER4*, COUNTER2:=COUNTER2+1
$p_{130}, p_{132}$	Extend Storage Unit
$p_{131}$	Extend Storage Unit and Move Storage Unit Downwards
$p_{133}$	Piece Stored
$p_{135}$	Storage Unit Ready ( horizontal )
$p_{136}$	Move Storage Unit Downwards
$p_{137}$	Storage Unit Ready ( vertical )
$p_{139}$	Storage Unit Ready
$p_{140}$	Reset COUNTER1, COUNTER2, COUNTER3, COUNTER4 and COUNTER5*
$p_{143}$	Raise, Extend Arm and Turn CCW
$p_{145}$	Raise Arm and Turn CCW

Table A.2: Complete Transitions.

Transitions	Meaning
$t_0$	Initialization Button
$t_1$	MAG1's Cylinder Retracted
$t_2$	MAG2's Cylinder Retracted
$t_3, t_{30}, t_{60}$	Right Discharge Cylinder Retracted
$t_4, t_{34}, t_{65}$	Center Discharge Cylinder Retracted
$t_5, t_{39}, t_{55}$	Left Discharge Cylinder Retracted
$t_6, t_{18}, t_{21}, t_{40}, t_{45}, t_{47}, t_{66},$ $t_{71}, t_{85}, t_{92}, t_{120}, t_{142}, t_{143},$ $t_{152}$	
$t_9$	Safety Door Opened
$t_{10}$	Assembly Unit Holder Extended
$t_{11}$	Storage Unit Retracted and Arm Lowered and Retracted
$t_{12}, t_{106}, t_{150}$	Storage Unit Right Limit Switch
$t_{13}, t_{151}$	Storage Unit Inferior Limit Switch
$t_{15}, t_{102}, t_{162}$	Inductive Sensor Arm

Continued on next page

Continued from previous page

Transitions	Meaning
$t_{17}, t_{104}$	ARMCOUNTER $\leq$ BELT_ANGLE_CW
$t_{19}$	Start Button
$t_{20}$	$\overline{\text{MAG1 Empty}}$
$t_{22}$	$\uparrow$ MAG1's Cylinder Extended
$t_{23}$	$\uparrow$ MAG1's Cylinder Retracted
$t_{26}, t_{56}$	$\overline{\text{Metallic Sensor}}$
$t_{27}, t_{57}, t_{61}$	$\overline{\text{White Color Sensor}}$
$t_{28}, t_{37}, t_{53}$	$\uparrow$ Proximity Sensor Left Discharge Cylinder
$t_{29}, t_{59}$	Right Discharge Cylinder Extended
$t_{31}$	White Color Sensor
$t_{32}, t_{63}$	$\uparrow$ Proximity Sensor Center Discharge Cylinder
$t_{33}, t_{64}$	Center Discharge Cylinder Extended
$t_{35}, t_{52}$	Metallic Sensor
$t_{36}, t_{67}$	Concavity Downwards
$t_{38}, t_{54}$	Left Discharge Cylinder Extended
$t_{41}, t_{62}$	Concavity Upwards
$t_{42}, t_{68}$	$\uparrow$ Proximity Sensor End Of Conveyor Belt
$t_{44}, t_{70}$	$\downarrow$ Proximity Sensor End Of Conveyor Belt
$t_{46}$	$\overline{\text{MAG2 Empty}}$
$t_{48}$	$\uparrow$ MAG2's Cylinder Extended
$t_{49}$	$\uparrow$ MAG2's Cylinder Retracted
$t_{58}$	$\uparrow$ Proximity Sensor Right Discharge Cylinder
$t_{72}$	Arm Raised
$t_{77}$	ARMCOUNTER $\leq$ PRESS_ANGLE
$t_{82}$	HALFPIECECOUNTER=1, Assembly Unit Holder Extended and Safety Door Opened
$t_{84}$	ARMCOUNTER $\geq$ BELT_ANGLE_CCW
$t_{95}, t_{101}$	Arm Raised, Storage Unit Right and Inferior Limit Switches
$t_{96}$	Storage Unit Arm Alignment Encoder
$t_{97}$	ARMCOUNTER $\leq$ STORAGE_ANGLE
$t_{100}$	Arm Lowered

Continued on next page

Continued from previous page

Transitions	Meaning
$t_{107}$	COUNTER2=0
$t_{108}$	COUNTER3=4 and Vertical Encoder
$t_{109}$	COUNTER3<=4 and Vertical Encoder
$t_{110}$	COUNTER2=1
$t_{111}$	COUNTER3=3 and Vertical Encoder
$t_{112}$	COUNTER3<=3 and Vertical Encoder
$t_{113}$	COUNTER2=2
$t_{114}$	COUNTER3=2 and Vertical Encoder
$t_{115}$	COUNTER3<=2 and Vertical Encoder
$t_{116}$	COUNTER2=3
$t_{117}$	COUNTER3=1 and Vertical Encoder
$t_{118}$	COUNTER3<=1 and Vertical Encoder
$t_{119}$	Vertical Encoder
$t_{121}$	COUNTER4=1
$t_{122}$	COUNTER5=1 and Horizontal Encoder
$t_{123}$	COUNTER5<=1 and Horizontal Encoder
$t_{124}$	COUNTER4=2
$t_{125}$	COUNTER5=2 and Horizontal Encoder
$t_{126}$	COUNTER5<=2 and Horizontal Encoder
$t_{127}$	COUNTER4=3
$t_{128}$	COUNTER5=3 and Horizontal Encoder
$t_{129}$	COUNTER5<=3 and Horizontal Encoder
$t_{130}$	COUNTER4=4
$t_{131}$	COUNTER5=4 and Horizontal Encoder
$t_{132}$	COUNTER5<=4 and Horizontal Encoder
$t_{133}$	COUNTER4=5
$t_{134}$	COUNTER5=5 and Horizontal Encoder
$t_{135}$	COUNTER5<=5 and Horizontal Encoder
$t_{136}$	COUNTER4=6
$t_{137}$	COUNTER5=6 and Horizontal Encoder
$t_{138}$	COUNTER5<=6 and Horizontal Encoder
$t_{139}$	COUNTER4=7

Continued on next page

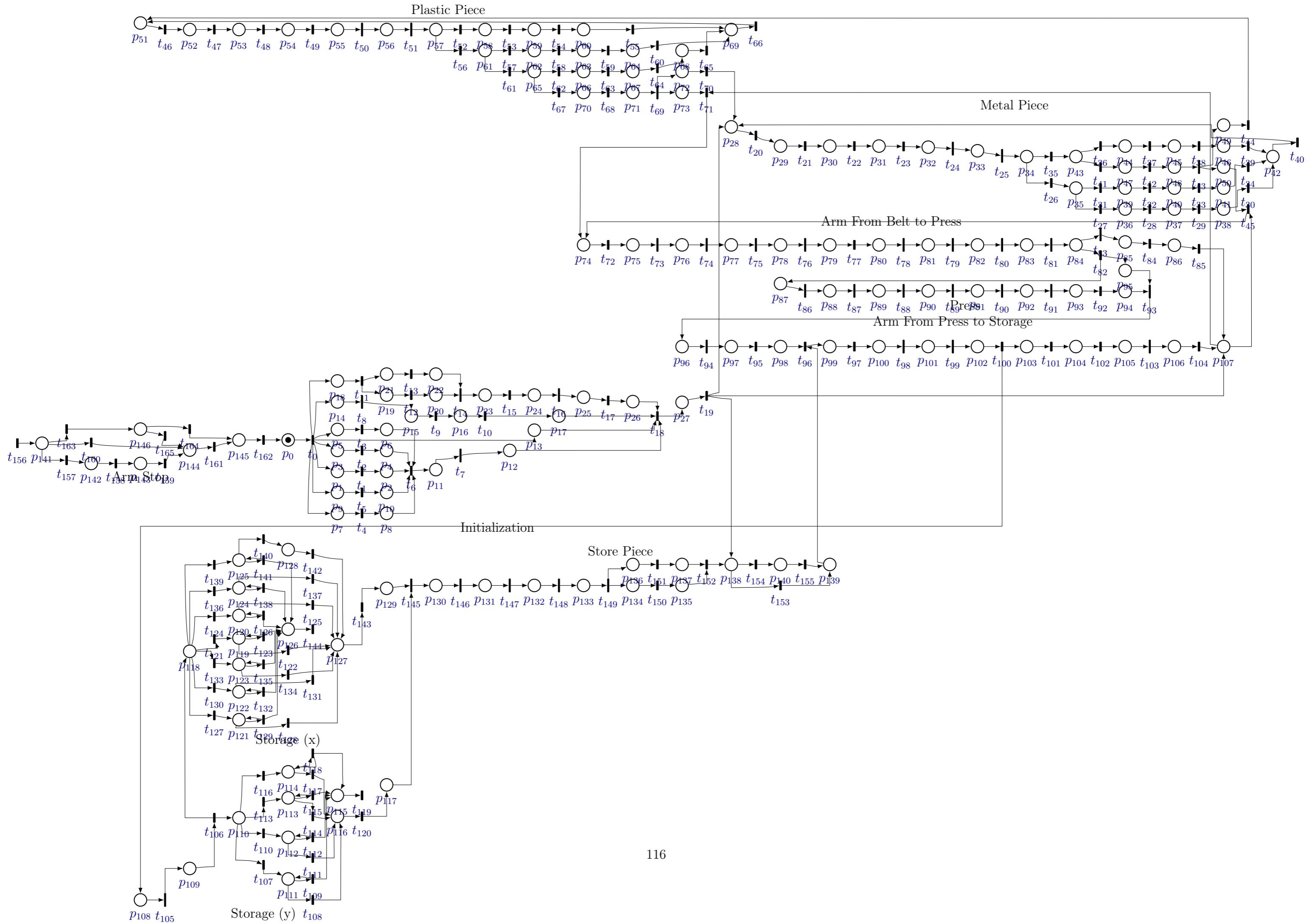
Continued from previous page

Transitions	Meaning
$t_{140}$	COUNTER5=7 and Horizontal Encoder
$t_{141}$	COUNTER5<=7 and Horizontal Encoder
$t_{144}$	Horizontal Encoder
$t_{153}$	COUNTER1<28
$t_{154}, t_{155}$	COUNTER1=28
$t_{156}$	Stop Button
$t_{157}$	ARMCOUNTER < STORAGE_ANGLE_BEFORE
$t_{158}$	Arm Raised and Extended
$t_{159}$	ARMCOUNTER >= STORAGE_ANGLE_BEFORE (ARMCOUNTER >= STORAGE_ANGLE_BEFORE and ARMCounter < PRESS_ANGLE_AFTER) or ARMCounter >= PRESS_ANGLE_BEFORE
$t_{160}$	ARMCounter >= PRESS_ANGLE_AFTER and ARMCounter < PRESS_ANGLE_BEFORE
$t_{161}$	Arm Raised and Retracted
$t_{163}$	ARMCounter >= PRESS_ANGLE_AFTER and ARMCounter < PRESS_ANGLE_BEFORE
$t_{164}$	Arm Retracted
$t_{165}$	Arm Retracted
$t_7$	T=12s
$t_8$	T=2.5s
$t_{14}, t_{98}, t_{99}, t_{105}, t_{145}$	T=2s
$t_{16}, t_{88}, t_{89}, t_{103}$	T=1s
$t_{24}, t_{43}, t_{50}, t_{69}$	T=0.5s
$t_{25}, t_{51}$	↑ Presence T=0.5s
$t_{73}, t_{80}$	T=1.5s
$t_{74}, t_{79}, t_{94}$	T=1.5s and Arm Lowered
$t_{75}, t_{76}, t_{78}, t_{81}$	T=1.5s and Arm Raised
$t_{83}$	T=1.5s, HALFPIECECOUNTER=0 and Raised Arm
$t_{86}$	T=1s and Assembly Unit Holder Retracted
$t_{87}$	T=1s and Safety Door Closed
$t_{90}$	T=1s and Safety Door Opened
$t_{91}$	T=1s and Assembly Unit Holder Extended
$t_{93}$	T=1.5s and Arm Extended

Continued on next page

Continued from previous page

Transitions	Meaning
$t_{146}, t_{148}$	T=3s
$t_{147}$	T=0.25s
$t_{149}$	T=7s



# Appendix B

## Tools

In this chapter, the most important tools used in this work will be presented. All the tools that were developed for this thesis are available at <https://github.com/Accacio/docsTCC/tree/master/tools>. The development of these tools was made using Ubuntu 18.04, wrapping some linux and unix programs/utilities, 100% compatibility with other operating systems/platforms was not the primary objective of this part of the work, but can be performed in some future work.

### B.1 `daoct`

To implement the `algorithm 1`, as seen in MOREIRA and LESAGE (2018), a script was created by Ryan Pitanga as part of his undergraduate thesis, PITANGA CLETO DE SOUZA (2019). His code was partially reimplemented, so it could be used as a command line tool based in common Linux utils (using stdin and stdout<sup>1</sup>, very useful to pipe<sup>2</sup> processes). Another modification, was to change the csv input file format, figure B.2, so the program could be generic, the names of the variables (inputs and outputs) are in the header, making the program more generic. One extra feature was added, now the automaton generated by the algorithm can be printed to stdout in the dot<sup>3</sup> file format, this output can be treated by one of the further described scripts to draw the automata shown on this thesis. Also there is an option to print the automaton in a list of *f*, as seen in B.4 An example of the output of the help option can be seen in Figure B.1

---

<sup>1</sup><http://man7.org/linux/man-pages/man3/stdin.3.html>

<sup>2</sup><http://man7.org/linux/man-pages/man2/pipe.2.html>

<sup>3</sup>format used by the program graphviz (<https://graphviz.org/>) to draw graphs

```
[ accacio@ accacio/tools/daoct ] daoct -h
Usage: daoct [OPTION] -i FILE ...
       daoct [OPTION] -s

-h, --help           give this help list
--debug            emit debuggin messages
--version          print program version
-i, --input          chooses input file
-s, --stdin          chooses stdin as input
-k                 chooses variable k value
-g, --graphviz        output automaton drawing in dot language
-f, --ffunction       output automaton in f function

Report bugs to raccacio@poli.ufrj.br
[ accacio@ accacio/tools/daoct ]
```

Figure B.1: daoct help dialog.

An example of a csv input and the script's different outputs can be seen in the following figures:

```
[ accacio@ accacio/tools/daoct ] cat sampleData.csv
SeqNo,Input1,Output1
1,0,1
2,0,0
3,1,0
4,0,0
5,0,1
6,0,0
7,1,0
8,0,0
9,0,1
10,0,0
11,1,0
//END,,,
[ accacio@ accacio/tools/daoct ]
```

Figure B.2: daoct input csv file.

```
[ accacio@ accacio/tools/daoct ] cat sampleData.csv | daoct -gs
digraph a {
rankdir=LR;
ratio=fill
graph [pad="0.5", nodesep="0.25", ranksep="0.2"];
node [shape=circle];
margin=0;
size="11.7,8.3!";
init [style=invis]
init -> x0
x2 [shape=doublecircle];
x4 [shape=doublecircle];
x0 -> x1 [texlbl=" ↓Output1 \\ \{ 0, 1 \} "]
x1 -> x2 [texlbl=" ↑Input1 \\ \{ 0, 1 \} "]
x2 -> x3 [texlbl=" ↓Input1 \\ \{ 0 \} "]
x3 -> x4 [texlbl=" ↑Output1 \\ \{ 0 \} "]
}
[ accacio@ accacio/tools/daoct ]
```

Figure B.3: daoct graphviz output.

```
[ accacio@ accacio/tools/daoct ] cat sampleData
f( x0 , ↓Output1 ) = x1
f( x1 , ↑Input1 ) = x2
f( x2 , ↓Input1 ) = x3
f( x3 , ↑Output1 ) = x4
[ accacio@ accacio/tools/daoct ]
```

Figure B.4: daoct *f* output.

## B.2 dot2automata

To visualize the output of the `daotc` program, the script `dot2automata` was created. It's basically a wrapper of the `dot2tex` program, that is capable of transforming a dot file in a tex file with the tikz syntax. `dot2automata` pre-process the dot file so the tikz output can be drawn using automaton style, with states and marked states, similar to graphs used in MOREIRA and LESAGE (2018). An example of the output of the help option can be seen in Figure B.5

```
[ accacio@ tools/dot2automata ] dot2automata -h
Usage: dot2automata [OPTIONS] FILE
Create Automata tikz file and/or pdf using dot2tex and latex.

When FILE is -, read standard input

-h, --help           give this help list
-d, --debug          emit debuggin messages
--version           print program version
-y,                 overwrites tex file
-p                 exports pdf file
-o, --output         chooses output name for pdf

Report bugs to raccacio@poli.ufrj.br
[ accacio@ tools/dot2automata ] █
```

Figure B.5: `dot2automata` Help.

As we can see there is an option to output a pdf file, so we can have a preview of the image. The tikz figure can be included in a latex and resized using the `tikzscale` package. So including a file as in Listing B.1 can result in the Figure B.8

Listing B.1: Include tikz file.

```
\begin{figure}[H]
\centering
\includegraphics [width=\textwidth]{tools/dot2automata/sampleData.tikz}
\caption{dot2automata output.}
\end{figure}
```

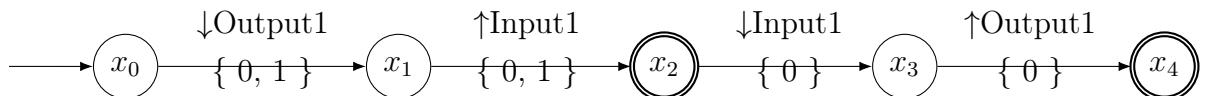


Figure B.6: `dot2automata` output.

## B.3 dot2petri

The script `dot2petri` is a similar to `dot2automata`, the same working principle but a different objective, this time the objective is to visualize Petri Nets. It's help dialog it's almost the same of `dot2automata`:

```
[ accacio@ tools/dot2petri ] dot2petri -h
Usage: dot2petri [OPTIONS] FILE
Create Petri Net tikz file and/or pdf using dot2tex and latex.

When FILE is -, read standard input

-h, --help           give this help list
-d, --debug          emit debuggin messages
--version           print program version
-y,                 overwrites tex file
-p                 exports pdf file
-o, --output         chooses output name for pdf

Report bugs to raccacio@poli.ufrj.br
[ accacio@ tools/dot2petri ]
```

Figure B.7: `dot2petri` Help.

The input dot file has a syntax slightly different from ‘plain vanilla’ dot language, as we can see in the following listing:

Listing B.2: `dot2petri` input dot file.

```
digraph A {
rankdir=LR;
ratio=fill
graph [pad="0.5", nodesep="0.25", ranksep="0.2"];

p0m3
p1
ep3
t1
tt2
et5
ett6

p0m3 -> t1 [label="3"]
p1 -> t1 [style="inhibitor"]
p1 -> tt2
p1 -> et5
t1 -> p1
tt2 -> ep3
tt2 -> p0m3 [label="3"]
ett6 -> p0m3

}
```

Using this modified syntax it is easy to define places, marked places, transitions, timed transitions, and different kinds of arcs. Places are defined using ‘p’ followed by an identification number, marked places are similar to places but have the letter ‘m’ and a number appended, this number represents how many tokens are in this place. Transitions are defined with a simple ‘t’ followed by its identification number and timed transitions are created using ‘tt’ and the id. The arcs can be defined using ‘->’ between two tags (between places and transitions), an inhibitor arc can be created changing the style of the arc, and a label can be used to define the number of tokens that are needed to trigger the transition or the number of tokens inserted in a place. A tikz style for places and transitions that are external to the current Petri Net are drawn using dotted lines, so we can see where different parts interconnect themselves.

Such dot files can be created in two ways: manually writing them or using another script called `petrimal2dot` present in the same repository. This other script converts a file in `petrimal`, created using the Platform Independent Petri net Editor 2 (PIPE2)<sup>4</sup> to the dot format. PIPE2 is a very powerful tool to design Petri Nets, since it is possible to simulate the net and it can generate reachability graphs, but at its current version, it lacks of a good way to export the graph besides eps non-vector graphic, enter the `petrimal2dot` and `dot2petri` scripts.

The code shown in [Listing B.2](#) used as input for the `dot2petri` script outputs a tikz file, that included in a similar fashion to the one shown in [Listing B.1](#), can result in the following figure:

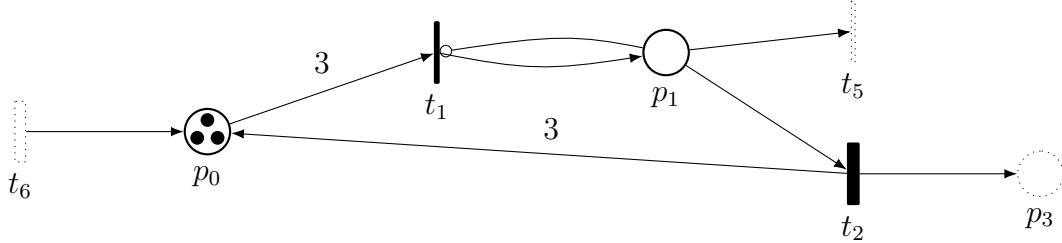


Figure B.8: `dot2petri` output.

## B.4 Other Scripts

A couple of other scripts were created to ease the building process. The `linkPetriNets` and `linkTables`, for instance, that together can link tex tables to tikz petri net figures, so they can hyperlinked in digital format, as seen in [Figure 4.1](#). And

---

<sup>4</sup><http://pipe2.sourceforge.net/>

other scripts that can pre-process the csv tables before sending to `daoct`, as in `treatCSV` and `removeVarsFromData`.