Universidade Federal
do Rio de Janeiro
Escola Politécnica

IDENTIFICATION OF A MECHATRONIC SYSTEM

Rafael Accácio 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
Julho de 2019

IDENTIFICATION OF A MECHATRONIC 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.

Prof. Lilian Kawakami Carvalho, D.Sc.

Prof. Gustavo da Silva Viana, D.Sc.

RIO DE JANEIRO, RJ – BRASIL
JULHO DE 2019

Nogueira, Rafael Accácio

Identificação de uma Planta Mecatrônica de Manufatura/Rafael Accácio Nogueira. – Rio de Janeiro: UFRJ/ Escola Politécnica, 2019.

XVIII, 118 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. 102 – 104.

1. Identification. 2. Discrete Event Systems. 3. Automata. 4. Planta mecatronica de manufatura inglês. 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 Mechatronic 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 MECHATRONIC SYSTEM

Rafael Accácio Nogueira

July/2019

Advisor: Marcos Vicente de Brito Moreira

Course: Automation and Control Engineering

This work has as primary objective to propose tools and present the implementation of a method for identification of discrete events systems using the Deterministic Automaton with Outputs and Conditional Transitions (DAOCT) model, which can be used to fault detection. In order to accomplish this, the control of a mechatronic 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.

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 UMA PLANTA MECATRÔNICA DE MANUFATURA

Rafael Accácio Nogueira

Julho/2019

Orientador: Marcos Vicente de Brito Moreira

Curso: Engenharia de Controle e Automação

Este trabalho tem como objetivo propor ferramentas e mostrar a implementação de um método 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 uma planta mecatrônica de manufatura, 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 será descrito nesse trabalho e o modelo identificado será discutido.

Contents

List of Figures

List of Tables

List of Acronyms

CCW
: Counter Clockwise

CIPN
: Control Interpreted Petri Net

CIPNs
: Control Interpreted Petri Nets

CSV
: Comma Separated Values

CW
: Clockwise

DAOCT
: Deterministic Automaton with Outputs and Conditional Transitions

DES
: Discrete Event System

DOF
: Degrees of Freedom

FBD
: Function Block Diagram

IL
Instruction List

LCA
Control and Automation Laboratory

LD
Ladder Diagram

NDAAO
Non-Deterministic Autonomous Automaton with Output

PLC
Programmable Logic Controller

SCADA
Supervisory Control and Data Acquisition

SCL
Structured Control Language

SFC
Sequential Function Chart

ST
Structured Text

UFRJ
Federal University of Rio de Janeiro

List of Symbols

R : Set of path indices
X : Set of states
X_f : Set of final states
Ω : Set of IO vectors
Σ : Set of events
λ : State output function
θ : Path estimation function
f : Deterministic transition function
x_0 : Initial State

Chapter 1

Introduction

In a world where the majority of the population lives in industrial societies, and machines take part in 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 causes are 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 this thing should work[1], because when we know how it should work, we can distinguish when it is and when it is not.

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

As great part of the manufacture facilities uses discrete sensors and actuators, as

[1] "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)

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 as 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); CARVALHO *et al.* (2017); KLEIN *et al.* (2005); KUMAR and TAKAI (2014); SAMPATH *et al.* (1995); VERAS *et al.* (2018); VIANA and BASILIO (2019) can be used as examples.

The procedure for detection and isolation of failure events proposed in SAMPATH *et al.* (1995) is based on the complete system behaviour. Although this procedure is used for small systems, applying the same procedure on larger systems can be challenging. The difficulty on the implementation of the procedure for large systems is caused by the concurrent behaviour they can present. As the procedure is based on the system behaviour, it is necessary to know the system and also people that are familiar with discrete-event modelling techniques. So, other approaches were created in order to make the modelling process automatically repeatable and without the need of knowing the system, using observation of the system and modelling algorithms.

This work is based on MOREIRA and LESAGE (2018), that develops an algorithm to identify a model of the system using 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 Manufacturing System with a concurrent behaviour and a moderate number of inputs and outputs (over 60) and show that the identification method can be used on concurrent 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 Work Outline

Background

In chapter 2 a background to understand this work is presented. First are presented the basic principles of Discrete Event Systems and two ways of modelling them (Automata and Petri nets). After, a way to use Petri nets to design a controlled system and to

translate this control to Ladder Logic are depicted (as in MOREIRA and BASILIO (2013)). And finally we present the identification algorithm, DAOCT (as in MOREIRA and LESAGE (2018)).

Didactic Manufacturing System

In chapter 3 we present the manufacturing system, its devices, sensors and actuators.

Control Logic

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

Manufacturing System Identification

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

Identified Model

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

Conclusion

In chapter 7, the conclusions are drawn. The drawbacks presented during the implementation are collected, and other approaches on some specific parts of this work are proposed as future works.

Chapter 2

Background

In this chapter the main topics needed to understand this work are presented. 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

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

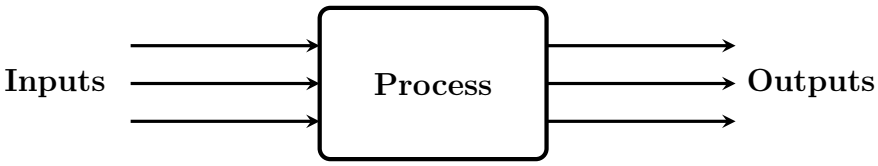


Figure 2.1: Input/Output Process model

The states of the system can be continuous or discrete, and the systems can be considered as Continuous, Discrete or 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 or by events, i.e., the states can change continuously with the time or instantaneously with the occurrence of events.

In this work, 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 CASSANDRAS 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 Σ. 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 Σ 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 terms "word" and "trace". We can also define the empty word, ϵ, that is, a word that is not formed by any event.

The operation 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 goes for the words "ten" and "net", that have different meanings in English.

We can also concatenate two words to create a different one. For instance, 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 ϵ, 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. Thus, given a word w its length is equal to $|w|$ and by definition $|\epsilon| = 0$.

Definition 2.1 (Language)
A Language defined over an alphabet Σ is formed of finite-length words generated from the concatenation of the events in Σ and ϵ.

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

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

2.4 Representation of Languages

Although languages can describe the behaviour of DESs, there are cases in which the language is countably infinite, what makes them not so simple to express the behaviour of the system. For this purpose, there are some other formalisms that are a more compact way of expressing the system's behaviour.

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 Σ, and we can create a set X composed of all states. From Σ 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 $f : X \times \Sigma \to X$. 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 z 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.

A representation of the transition function 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.

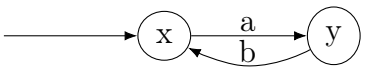


Figure 2.2: State Transition Diagram

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

Example 2.1 (Simple Automaton)
Let $\Sigma = \{a, b, g\}$, $X = \{x, y, z\}$ and the following transition functions (CASSANDRAS and LAFORTUNE, 2009):

$$f(x, a) = x \qquad f(x, g) = z$$
$$f(y, a) = x \qquad f(y, b) = y$$
$$f(z, b) = z \qquad f(z, a) = f(z, g) = y$$

We can represent this automaton with the diagram of Figure 2.3

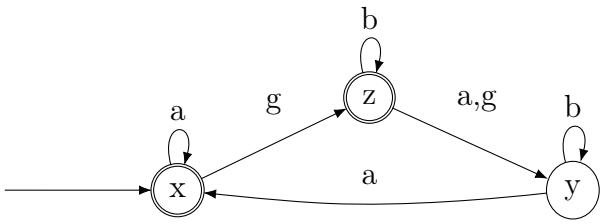


Figure 2.3: Diagram representing the automaton from example 2.1

We can also mark states that have some special meaning, as for instance, a final state. In this work, as in CASSANDRAS and LAFORTUNE (2009), the marked states are 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***
Σ *is the finite set of* ***events*** *associated with* G
$f : X \times \Sigma \to X$ *is the* ***transition function***
x_0 *is the* ***initial state***
$X_m \subseteq X$ *is the set of* ***marked states***

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 1960s. Differently from the automata representation, 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

Arcs are used to connect nodes and have arrowheads to identify the direction. 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. A Petri net is bipartite graph, which means that places can only connect to transitions and vice versa. In this work, as in DAVID and ALLA (2005) places are represented by circles and transitions by bars, as shown in Figure 2.4.

(a) A place. (b) A transition

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 these two sets, we can then define those functions. The first one represents the arcs from places to transitions, and is denoted as $Pre : P \times T \to \mathbb{N}$, the second one represents the arcs that connect transitions to places, denoted as $Post : P \times T \to \mathbb{N}$. Where $\mathbb{N} = \{0, 1, 2, \dots\}$ is the set of non-negative integers.

Example 2.2 (Simple Petri Net structure)
Given $P = \{p_0, p_1\}$, $T = \{t_0, t_1, t_2\}$ *and the following functions:*

$$
\begin{array}{cccc}
Pre(p_0, t_0) = 0 & Post(p_0, t_1) = 0 & Pre(p_1, t_0) = 0 & Post(p_1, t_1) = 2 \\
Post(p_0, t_0) = 1 & Pre(p_0, t_2) = 0 & Post(p_1, t_0) = 0 & Pre(p_1, t_2) = 1 \\
Pre(p_0, t_1) = 1 & Post(p_0, t_2) = 0 & Pre(p_1, t_1) = 0 & Post(p_1, t_2) = 0
\end{array}
$$

We can represent this Petri net structure with the diagram of Figure 2.5

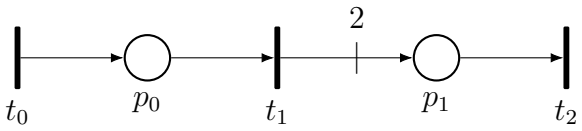
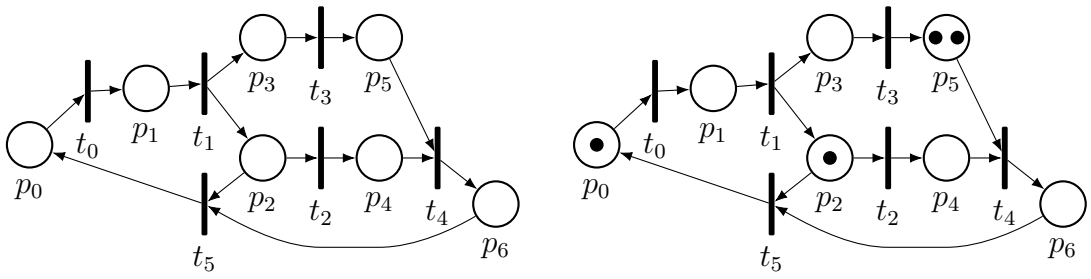


Figure 2.5: Diagram representing the Petri net structure from example 2.2

Marking

The marking is used as the mechanism to represent if the condition of occurrence of a determined event is met or not. The marking also represents the state of the system. The mechanism works as follows. Tokens can be assigned 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 $x : P \to \mathbb{N}$ 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.

Figures 2.6a and 2.6b show an unmarked and a marked Petri net graph, respectively.



(a) Unmarked (b) Marked

Figure 2.6: 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 Figure 2.6b is the following vector.

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

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 one state to other.

Firing Transitions

When an event associated with a transition t_j happens and t_j is enabled, t_j is fired and a new marking is reached. We can define the functions $I : T \to 2^P$ and $O : T \to 2^P$ that describe the set of places considered as inputs and outputs of a transition:

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

Definition 2.3 (Enabled transition)
A transition is enabled if

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

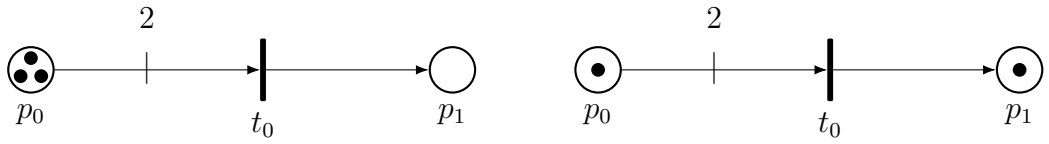
If $I(t_j) = \emptyset$, t_j is always enabled.

And we can define the dynamic of the Petri net as follows.

Definition 2.4 (Petri net dynamics)
It is possible to define a state transition *function, $f : \mathbb{N}^n \times T \to \mathbb{N}^n$, where n is the size 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 $\mathbf{x}'$:*

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

As an example we can take Figure 2.7:



(a) Before firing of t_0.

(b) After firing of t_0.

Figure 2.7: Example of Petri net Dynamic.

The state before firing transition t_0 is $\mathbf{x} = \begin{bmatrix} 3 & 0 \end{bmatrix}^T$ 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 $\mathbf{x}' = \begin{bmatrix} 1 & 1 \end{bmatrix}^T$.

A Petri Net is defined as follows.

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 in Definition 2.4.*

To make the connection between the Petri net and the events of the system, we can define a labeling function, $l : T^* \to \Sigma^*$ that makes the link between a sequence of firing transitions and a sequence of events.

Definition 2.6 (Labelled Petri net)
A Labelled 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*

Σ *is the set of* ***events***

l *is the* ***labelling*** *function*

Usually, the events are represented in the Petri net graph over its respective transition as shown in Figure 2.8. This system has an alphabet $\Sigma = \{a, b\}$ and labelling functions $l(t_0) = a$ and $l(t_1) = b$.

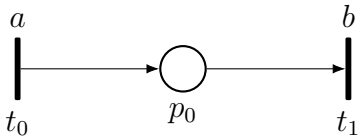


Figure 2.8: Labeled Petri net.

2.5 Control Interpreted Petri Nets

One of the important application of Petri nets, besides modelling a system, is its ability to model the control of a system. For this intent we use Control Interpreted Petri Nets (CIPNs). It is an extension of labelled Petri nets, in which we add actions to 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 the ability to delay the firing transitions based on time.

Definition 2.7 (Control Interpreted Petri net)
A Control Interpreted Petri net is defined as a thirteen-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*

Σ *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* Σ *and* C

D *is the set of* ***delays*** *associated to timed 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 $In : (P \times T) \to \mathbb{N}$ 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 2.9.

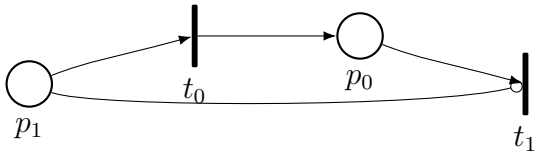


Figure 2.9: Example of Petri net with inhibitor arc.

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 $l_C : T^0 \to (\Sigma \times C)$ defines a pair of event and boolean condition from Σ 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 (σ_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, transition t_0 is fired when the condition c_0 is true and the event σ_0 happens, but obviously, if and only if this transition is enabled. The transition t_0 is represented graphically as shown in Figure 2.10

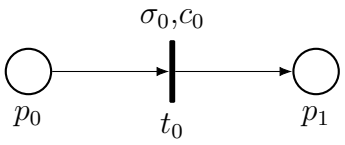


Figure 2.10: Representation of new labeling function

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

On the other hand, the labeling function $l_D : T^D \to D$, defines a delay for the transition to be fired. A timed transition $t_i \in T^D$ (a subset of T that represents the transitions with a time delay), has a corresponding delay d_i. As an example, consider a timed transition t_1 and $D = \{d_1\}$. Then, after the enablement of the transition, it takes d_1 time units in order to be fired. In this work, timed transitions are represented by white bars slightly larger than normal transitions. An example of this representation we can see in Figure 2.11

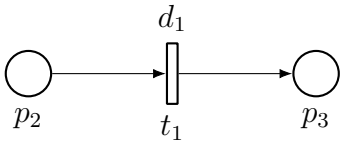


Figure 2.11: Representation of a timed transition.

Another labelling function is $l_A : P \to 2^A$, assigning a set of actions belonging to A to a place. Actions can be impulse or continuous actions. A continuous action happens when 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 1. Actions

are represented graphically as labels in places. Impulse actions are differed by a star (*) at its end.

In Figure 2.12, a representation of a place with both kinds of actions is presented, where F is continuous and $B^{\star}$.

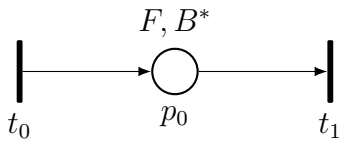


Figure 2.12: Representation of labeling of Actions.

Although these representations exist, in this work events, conditions and action labels are suppressed from the diagrams, and tables are added to the drawings showing the meaning of the transitions (firing events and conditions) and places (Actions). This choice was made because the CIPNs presented in this work are very large.

To illustrate how the tables are used to complement the information of the diagram, 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 2.13. 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*, a 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 re-initiate the loop.*

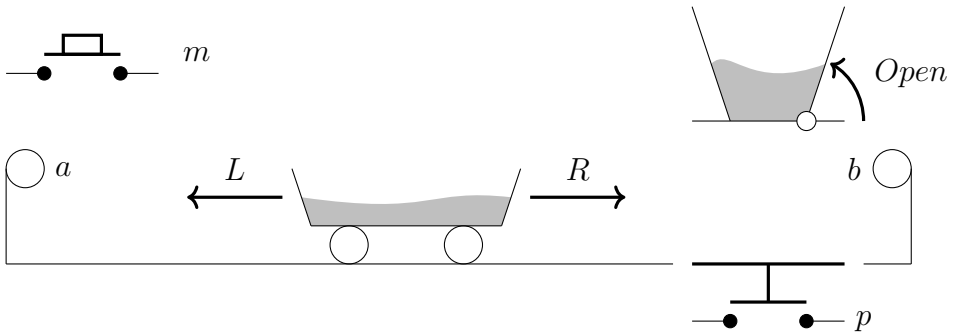


Figure 2.13: Example of System to be controlled by the Petri Net

From the description of the control it is possible to create a Control Interpreted Petri Net (CIPN) to represent it, as the one in Figure 2.14.

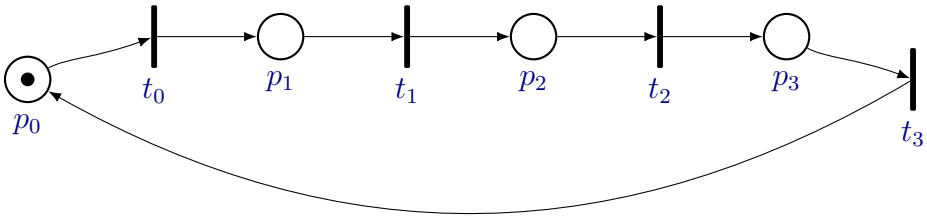


Figure 2.14: Example of Control Interpreted Petri Net to control system in Figure 2.13

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	↑ m (filling request)
t_1	↑ b (Right Limit Switch)
t_2	↑ p (Car is Full)
t_3	↑ a (Left Limit Switch)

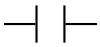
In this work as in the usual boolean notation, when just the name of a variable is given in a table it means the variable is equal to true, and when there is a bar in its top it is equal to false, so they determine conditions. E.g.: b and $\overline{b}$. And when a variable is preceded by $\uparrow$ and $\downarrow$, they determine events corresponding to its rising 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 IEC 61131 standard, defines in its third part (IEC 61131-3) the five languages to program PLCs: Ladder Diagram (LD), Function Block Diagram (FBD), Structured Text (ST), Instruction List (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.

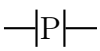
2.6.1 Ladder Logic

The ladder logic is based on two components, contacts and coils. Their terminals are interconnected to transmit boolean signals. 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 2.15 and 2.16

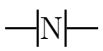


(a) Normally Open Contact.

(b) Normally Closed Contact.

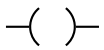


(c) Positive Edge Contact.



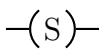
(d) Negative Edge Contact.

Figure 2.15: Types of Contacts.

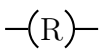


(a) Coil.

(b) Negated Coil.



(c) Set (latch) Coil.



(d) Reset (unlatch) Coil.

Figure 2.16: 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 open 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 rising ($\uparrow$) and falling edge ($\downarrow$) events and normally open 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 needed 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 2.17.

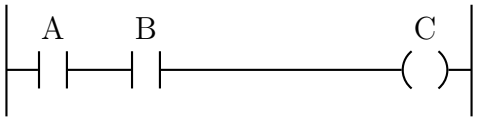


Figure 2.17: And logic in a Ladder rung.

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 2.18).

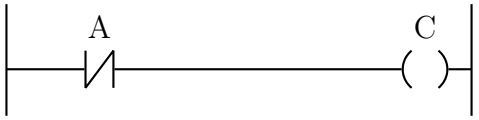


Figure 2.18: Not logic in a Ladder rung.

C will only by activated if A is $false$. $(C = \overline{A})$

Although all logic connectives can be constructed with this two connectors, the OR connector can be achieved by using contacts in parallel (Figure 2.19).

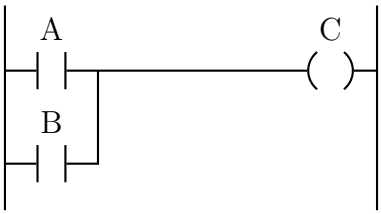
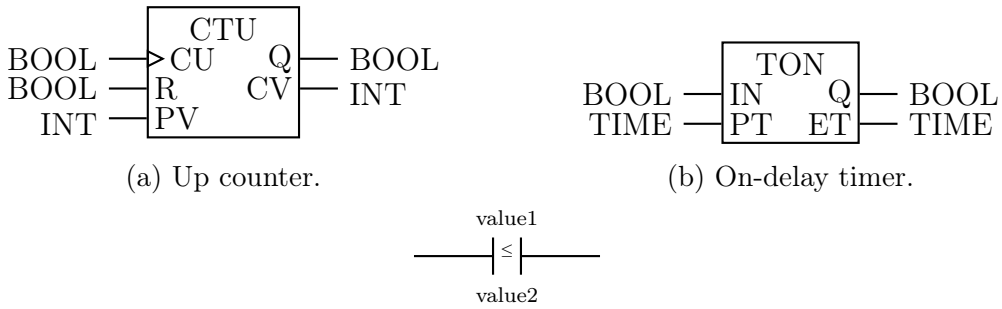


Figure 2.19: Or logic in a Ladder rung.

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:



(a) Up counter.

(b) On-delay timer.

(c) Less or equal comparator.

Figure 2.20: Examples of function blocks.

Up counters (Figure 2.20a) 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 2.20a) 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 2.20c, 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 Control Interpreted Petri Nets to Ladder Diagram

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

It consists in dividing the CIPN into 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 can 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 dynamics of the Petri net

4. A module of initialization
   It determines the initial marking of the net.

5. A module of actions
   It 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 edges of the inputs of the system, in this module, positive and negative edge contacts are used to detect the rising and falling edge events. The variables are stored in variables using coils and a variable is created for each event. For visibility's and organisation's sake a rung is used for each event, resulting $|\Sigma|$ rungs. This module is not necessary when using Siemens PLCs. So it will not be used in this work.

Firing Conditions

As said in section 2.5, a transition t_j fires, if it is 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 has 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 have at most one token for all reachable markings of the Petri net, a *boolean* variable can be used to store the number of tokens. In this case a 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 Petri net 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 change in the Petri net marking. These blocks, increase and decrease the values from the *int* variables that represent the marking of each place. If the capacity of the places is not greater than one, then the marking of those places can be represented by boolean variables. Set and Reset coils can be used to represent the marking of such places. Again, for organisation's sake a rung is associated for each transition, resulting $|T|$ rungs.

Initialization

The Initialization module works similarly to the dynamics module. A initial transition is created and when this transition is fired, the marking of the Petri net is changed to its initial marking. When the system is turned on, the condition for firing this initial transition is true and it is disabled immediately, so this transition can only be fired once, in the initialization phase.

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 to" comparators, but if it is represented by a *bool* variable, a normally open contact can be used.

Example

An example of this conversion can be given using the same CIPN from Example 2.3. The external events and firing condition modules are grouped in the same module in this work. The converted Ladder Logic is depicted in Figure 2.21.

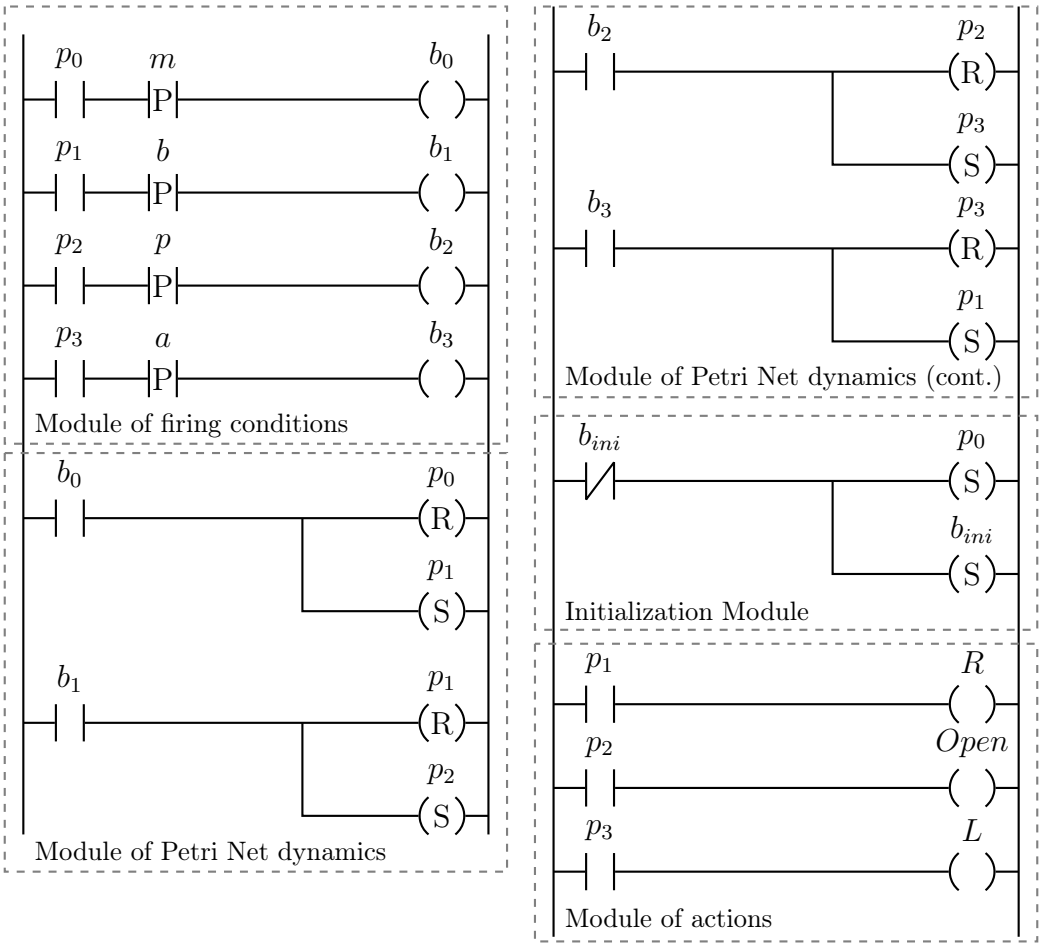
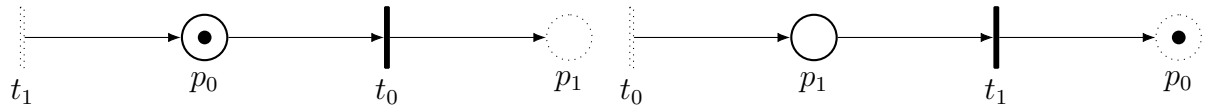


Figure 2.21: Example of Control Interpreted Petri Net converted to Ladder.

2.6.3 Control Interpreted Petri Net implemented in multiple PLCs

In some cases, the implementation of the control code must be carried out in several PLCs. In Figure 2.22, it is shown an example of a CIPN implemented between 2 PLCs. As we can see, in Figures 2.22a and 2.22b 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. Those transitions and places 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.



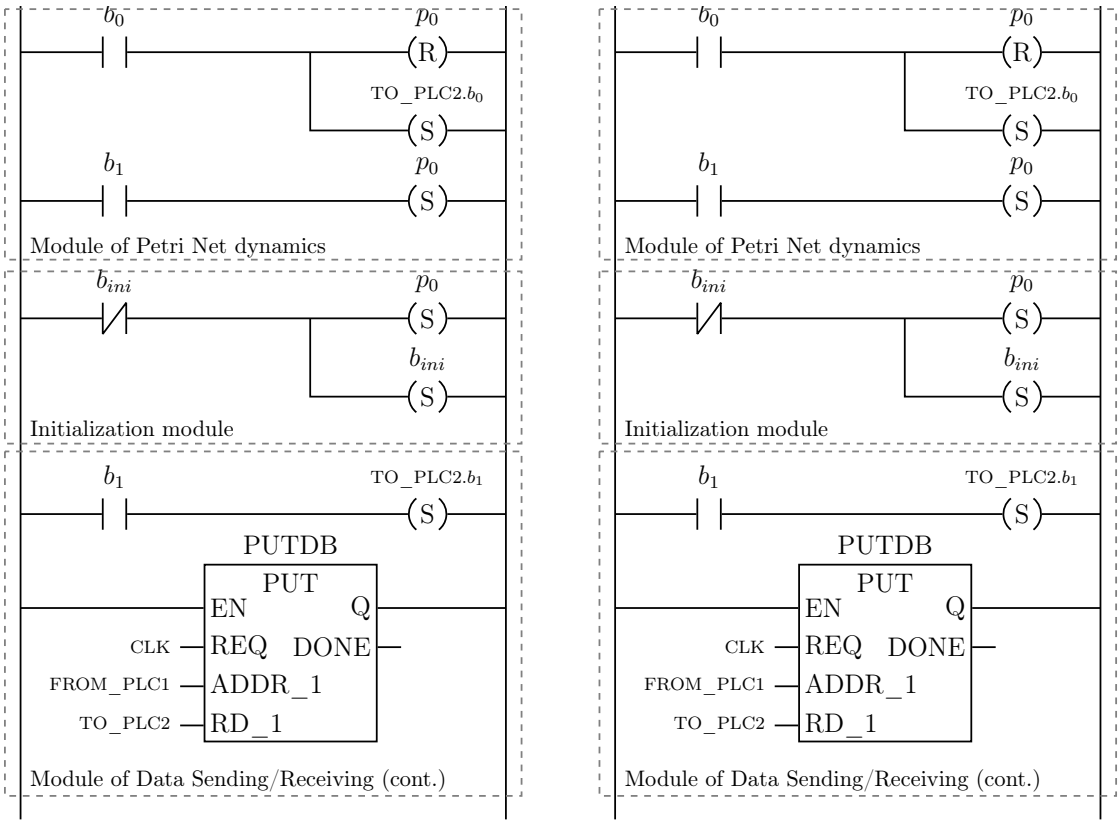
(a) Petri Net on PLC 1.

(b) Petri Net on PLC 2.

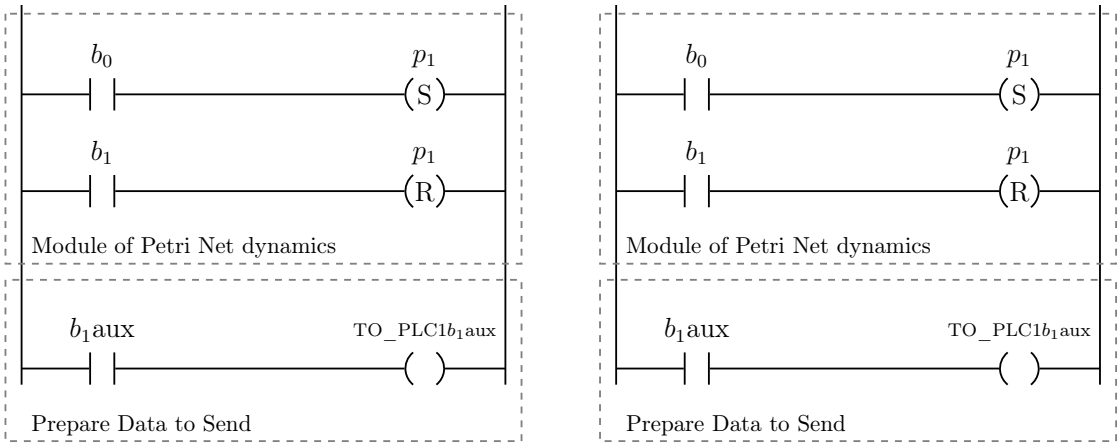
Figure 2.22: Example of Petri Net implemented in 2 PLCs.

In order to solve the problem of communication caused by the division, there are different ways, one of them is the method shown by FLORIANO (2019), where different sections are synchronised in a distributed manner using common places. In this work, a master/slave approach is used. CLP1 is considered as the master device and CLP2 as a slave. For the master, it is created another ladder module called "Data Sending/ Receiving", divided in 2 parts. The first part is implemented before all modules of the Ladder code, with the objective of getting all needed variables from other PLCs. The second part, that is implemented after all modules of the Ladder code, with the objective of sending variables to all other PLCs. On the other side, the slaves have 2 additional modules, one at the beginning of the Ladder code, called "Prepare Received Data" and another at the very end called "Prepare Data to Send". Those modules are created in order to avoid modification of variables in the middle of the program cycle. Change on variables used by the slaves PLCs caused by the master during the logic can entail unexpected behaviour.

The communication between PLCs can be accomplished by using Profinet protocol. Siemens PLCs 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 those blocks can be found on (FLORIANO, 2019; OLIVEIRA, 2016; PEREIRA, 2019). The complete implementation of the CIPN depicted in Figure 2.22 using LD is shown in Figure 2.23. In Figures 2.23a and 2.23b, we can see the conditions for transition $b0$ being transmitted from PLC1 to PLC2, and the conditions for transition $b1$ being transmitted from PLC2 to PLC 1. Since the dynamic of both conditions is divided in the two PLCs.



(a) Ladder Logic on PLC 1.



(b) Ladder Logic on PLC 2.

Figure 2.23: Example of Petri Net divided between 2 PLCs.

The logic presented, using the master/slave approach, works well when there are 2 PLCs. When there are more PLCs, this centralised approach creates a single point of failure. Since 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 FLORIANO (2019).

2.7 Identification

Once the control is implemented and the system is working as expected, the identification of the system can be carried out. In MOREIRA and LESAGE (2018), a new model for DES identification is proposed, called Deterministic Automaton with Outputs and Conditional Transitions (DAOCT). This model is created with the aim of fault detection based on the observation of the fault free behaviour of the system, as in ROTH *et al.* (2009) and KLEIN *et al.* (2005), but the use of path indices increases the efficiency for fault detection when compared with the latter articles.

The fault free observation is made by acquiring the observable signals of the system (controller inputs and outputs) for a sufficiently long period while the system works normally. Those signals can be seen on Figure 2.24.

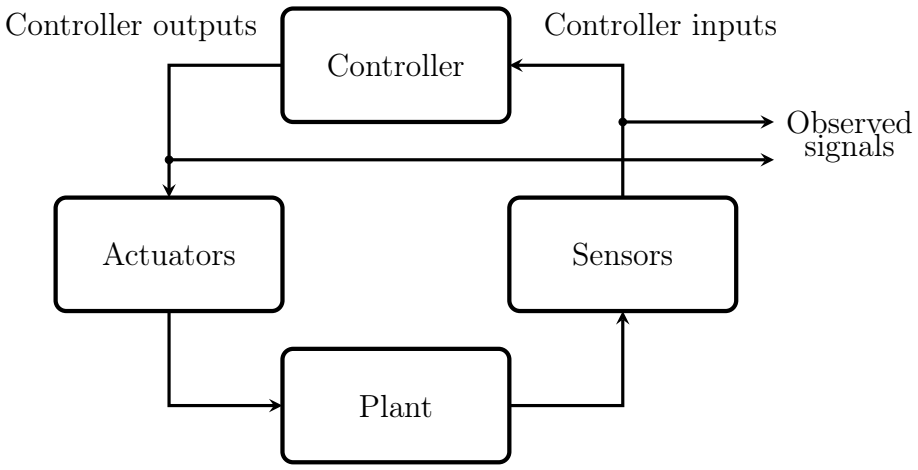


Figure 2.24: Observed Signals in a closed-Loop DES.

First, we assume the controller has m_i binary inputs and m_o binary outputs. We create an Input/Output vector called $\mathbf{u}$ as follows:

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

Vector $\mathbf{u}(t)$ represents the status of the system in an 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, σ. The vector $\mathbf{u}(t)$ is also 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 a sequence of l input/output vectors is observed, then the observed path of the system is $p = (\mathbf{u}_1, \sigma_1, \mathbf{u}_2, \sigma_2, \dots, \sigma_{l-1}, \mathbf{u}_l)$. So, if we observe multiple paths in the observation process, multiple paths can be created $p_i = (\mathbf{u}_{i,1}, \sigma_{i,1}, \mathbf{u}_{i,2}, \sigma_{i,2}, \dots, \sigma_{i,l_i-1}, \mathbf{u}_{i,l_i})$, for

$i = 1, \ldots, r$, 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 ω_i, defined as:

$$s_i = \sigma_{i,1}\sigma_{i,2}\ldots\sigma_{i,l_i-1}$$
$$\omega_i = \mathbf{u}_{i,1}\mathbf{u}_{i,2}\ldots\mathbf{u}_{i,l_i}$$

From the observed sequences of events s_i, we can define a language, called observed language, L_{Obs}:

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

Remark 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 ω_i, since it does not present any new information for the identification process.*

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

From L_{Orig} and L_{Iden}, another language can be defined, an exceeding language L_{Exc}. The exceeding language represents the part of the identified language that is not presented in the original language, $L_{Exc} = L_{Iden} \backslash L_{Orig}$. The relation between L_{Orig} and L_{Obs} is $L_{Obs} \subset L_{Orig}$. And from L_{Orig} and L_{Iden}, L_{OrigNI} can be defined. L_{OrigNI} represents the part of the original language that is not identified, $L_{OrigNI} = L_{Orig} \backslash L_{Iden}$. The relation between all languages presented is shown in Figure 2.25.

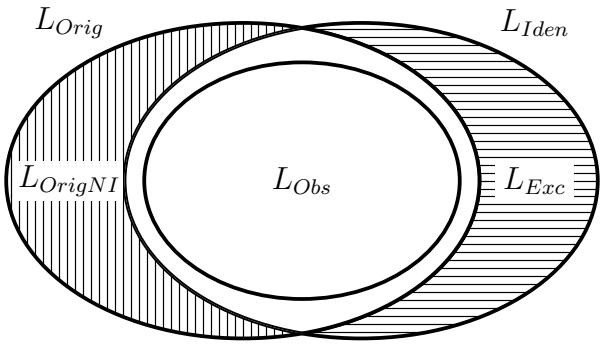


Figure 2.25: Relation between L_{Orig}, L_{OrigNI}, L_{Obs}, L_{Exc} and L_{Iden}

.

As L_{Exc} represents the part of the identified language that is not in 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 possible.

As L_{OrigNI} represents the part of the original system that is not identified, some sequences in the fault-free behaviour of the system will be detected as faults, generating false alarms. L_{OrigNI} must 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 estimate 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 $L_{Orig}^{\leq n_0} \backslash L_{Obs}^{\leq n_0} \approx \emptyset$, where $L_{Orig}^{\leq n_0}$ and $L_{Obs}^{\leq n_0}$ denote the languages formed of all sequences of events of length smaller than or equal to n_0 of L_{Orig} and L_{Orig}, respectively. 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 $L_{OrigNI}^{\leq n_0} = \emptyset$. Thus, the identified model must reduce the language $L_{Exc}^{\leq n_0}$.

2.7.1 Deterministic Automaton with Outputs and Conditional Transitions

In this subsection the modified automaton model proposed by MOREIRA and LESAGE (2018) is explained and the algorithm to construct it by the observed paths p_i is described.

Definition 2.8 (DAOCT)
A Deterministic Automaton with Outputs and Conditional Transitions, denoted by DAOCT, is a nine-tuple

$$DAOCT = (X, \Sigma, \Omega, f, \lambda, R, \theta, x_0, X_f)$$

where:

X *is the set of* ***states***
Σ *is the finite set of* ***events***
$\Omega \subset \mathbb{N}_1^{m_i+m_o}$ *is the set of* ***I/O vectors***
$f : X \times \Sigma^\star \to X$ *is the* ***deterministic transition*** *function*
$\lambda : X \to \Omega$ *is the* ***state output*** *function*
$R = 1, 2, \ldots, r$ *is the set of* ***path indices***
$\theta : X \times \Sigma \to 2^R$ *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 by Σ_i and Ω_i. Thus, Σ and Ω can be calculated by the union of these sets: $\Sigma = \bigcup_{i=1}^{r} \Sigma_i$ and $\Omega = \bigcup_{i=1}^{r} \Omega_i$. The states of the model are obtained from the vertices of the paths p_i, the I/O vectors. Each vertex is chosen as a new state. In some systems, certain I/O vectors are repeated in the same path, but with different preceding vectors. In order to differentiate these vectors, its preceding vectors are stored in a sequence of I/O vectors. To determine the number of vectors stored in this sequence a free parameter k is used. 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:

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

where

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

Remark 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 of all vertices of its corresponding path p_i.*

As an example of the computation of the paths p_i^k, let us consider the following paths, (MOREIRA and LESAGE, 2018):

$$p_1 = \left(\begin{bmatrix}1\\0\\0\end{bmatrix}, a, \begin{bmatrix}1\\1\\0\end{bmatrix}, b, \begin{bmatrix}0\\1\\1\end{bmatrix}, c, \begin{bmatrix}0\\0\\0\end{bmatrix}, d, \begin{bmatrix}0\\0\\1\end{bmatrix}, e, \begin{bmatrix}1\\0\\0\end{bmatrix} \right)$$

$$p_2 = \left(\begin{bmatrix}1\\0\\0\end{bmatrix}, g, \begin{bmatrix}0\\0\\0\end{bmatrix}, h, \begin{bmatrix}1\\1\\0\end{bmatrix}, b, \begin{bmatrix}0\\1\\1\end{bmatrix}, c, \begin{bmatrix}0\\0\\0\end{bmatrix}, i, \begin{bmatrix}1\\0\\0\end{bmatrix}, j, \begin{bmatrix}0\\1\\1\end{bmatrix}, l, \begin{bmatrix}1\\0\\0\end{bmatrix} \right)$$

$$p_3 = \left(\begin{bmatrix}1\\0\\0\end{bmatrix}, g, \begin{bmatrix}0\\0\\0\end{bmatrix}, h, \begin{bmatrix}1\\1\\0\end{bmatrix}, b, \begin{bmatrix}0\\1\\1\end{bmatrix}, i, \begin{bmatrix}1\\1\\1\end{bmatrix}, m, \begin{bmatrix}0\\0\\0\end{bmatrix}, d, \begin{bmatrix}0\\0\\1\end{bmatrix}, n, \begin{bmatrix}1\\1\\0\end{bmatrix} \right)$$

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

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

$$p_1^2 = \left(\begin{bmatrix}1\\0\\0\end{bmatrix}, a, \begin{bmatrix}1&1\\0&1\\0&0\end{bmatrix}, b, \begin{bmatrix}1&0\\1&1\\0&1\end{bmatrix}, c, \begin{bmatrix}0&0\\1&0\\1&0\end{bmatrix}, d, \begin{bmatrix}0&0\\0&0\\0&1\end{bmatrix}, e, \begin{bmatrix}0&1\\0&0\\1&0\end{bmatrix} \right)$$

$$p_2^2 = \left(\begin{bmatrix}1\\0\\0\end{bmatrix}, g, \begin{bmatrix}1&0\\0&0\\0&0\end{bmatrix}, h, \begin{bmatrix}0&1\\0&1\\0&0\end{bmatrix}, b, \begin{bmatrix}1&0\\1&1\\0&1\end{bmatrix}, c, \begin{bmatrix}0&0\\1&0\\1&0\end{bmatrix}, i, \begin{bmatrix}0&1\\0&0\\0&0\end{bmatrix}, j, \begin{bmatrix}1&0\\0&1\\0&1\end{bmatrix}, l, \begin{bmatrix}0&1\\1&0\\1&0\end{bmatrix} \right)$$

$$p_3^2 = \left(\begin{bmatrix}1\\0\\0\end{bmatrix}, g, \begin{bmatrix}1&0\\0&0\\0&0\end{bmatrix}, h, \begin{bmatrix}0&1\\0&1\\0&0\end{bmatrix}, b, \begin{bmatrix}1&0\\1&1\\0&1\end{bmatrix}, i, \begin{bmatrix}0&1\\1&1\\1&1\end{bmatrix}, m, \begin{bmatrix}1&0\\1&0\\1&0\end{bmatrix}, d, \begin{bmatrix}0&0\\0&0\\0&1\end{bmatrix}, n, \begin{bmatrix}0&1\\0&1\\1&0\end{bmatrix} \right)$$

The states of the system are obtained from these vertices. In order to do so, the labelling function is defined, $\tilde{\lambda} : X \to \Omega^k$. Where Ω^k is composed of all sequences of Ω 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 $x \in X$. Let us 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,. . . ,r

Output: DAOCT = (X,Σ,Ω,f,λ,R,θ,x_0,X_f)

Create an initial state x_0, and define $\lambda(x_0) = \tilde{\lambda}(x_0) = y_{1,1}$
$X = \{x_0\}, X_f = \emptyset, R = \emptyset$
for $i = 1$ **to** r **do**
    $R = R \cup \{i\}$
    for $j = 1$ **to** $l_i - 1$ **do**
        Find the State $x \in X$ such that $\tilde{\lambda}(x) = y_{i,j+1}$
        if $\tilde{\lambda}(s) \neq y_{i,j+1}$ *for all* $s \in X$ **then**
            Create state x' and define $\tilde{\lambda}(x') = y_{i,j+1}$
            $X = X \cup \{x'\}$
            $\lambda(x') = \tilde{\lambda}_l(x')$
        else
            Find $x' \in X$ such that $\tilde{\lambda}(x') = y_{i,j+1}$
        end
        $f(x, \sigma_{i,j}) = x'$
        Add i to $\theta(x, \sigma_{i,j})$
        if $j = l_i - 1$ **then**
            $X_f = X_f \cup \{x'\}$
        end
    end
end

In this algorithm, each state created represents an unique vertex of the modified paths, and the states marked as final states represent the last vertex of each modified path.

Using the modified paths p_1^1, p_2^1, p_3^1, and p_1^2, p_2^2, p_3^2 as inputs of algorithm 1, two models can be identified, one for $k = 1$ and another for $k = 2$. The state transition diagrams

for the identified models are represented in Figures 2.26 and 2.27 respectively.

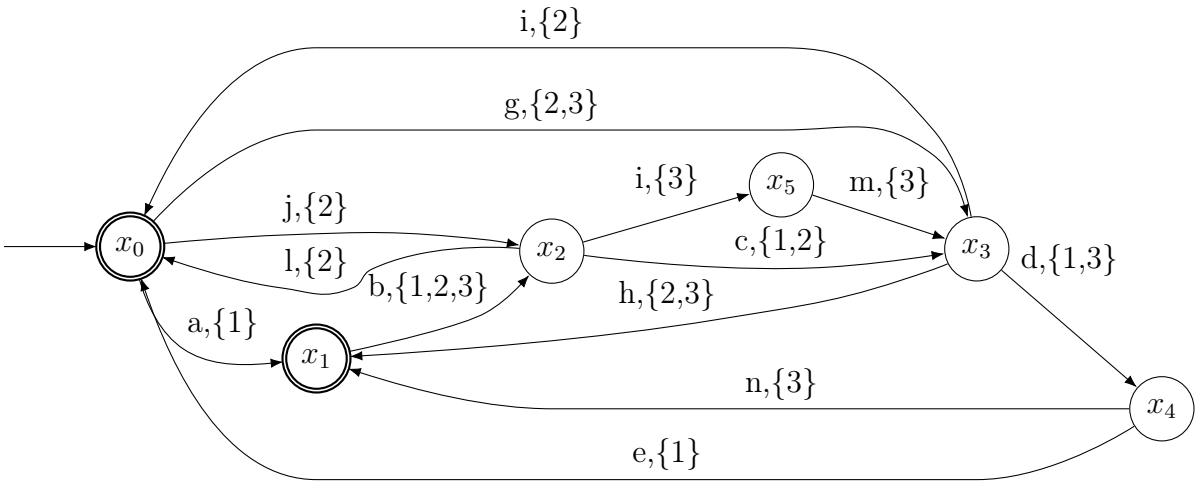


Figure 2.26: State transition diagram for identified model using $k = 1$.

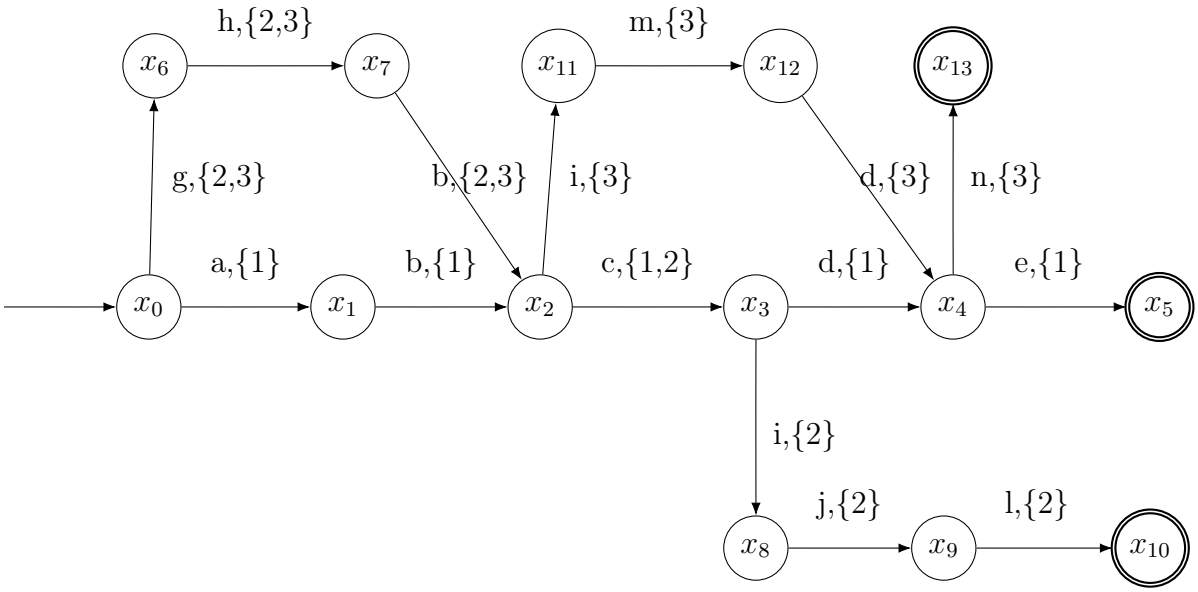


Figure 2.27: State transition diagram for identified model using $k = 2$.

As expected, with a greater value of k, more states are created in the identification process.

The path estimation function θ 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 θ

to consider sequence of events, instead of only one event. This new estimation function will be denoted as $\theta_s : X \times \Sigma^\star \to 2^R$, and it is defined recursively as:

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

Where ! denotes *is defined*.

The language generated by the identified DAOCT model is given by:

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

Using a similar logic, it is possible to define the language formed of all subsequences of events of length n generated by the identified model:

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

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

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

Assuming all sequences of events that have length $n_0 + 1$ have been observed, then $L_{\overline{Orig}}^{\leq n_0} \approx L_{\overline{Obs}}^{\leq n_0}$. If $n \leq n_0$, then $L_{\overline{Orig}}^{\leq n} \approx L_{\overline{Obs}}^{\leq n}$. Thus, $L_{\overline{Exc}}^{\leq n} = L^{\leq n}(DAOCT) \backslash L_{\overline{Obs}}^{\leq n}$, and this formula will be used to calculate $L_{\overline{Exc}}^{\leq n}$ throughout this work.

Chapter 3

Didactic Manufacturing System

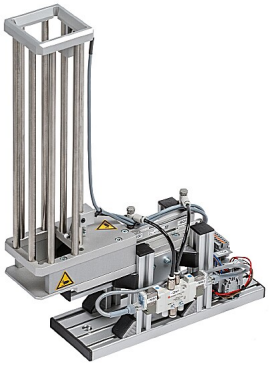
In this chapter, the system to be controlled and identified is presented. The mechatronic system is a didactic manufacturing system assembled from submodules fabricated by Christiani[1]. This manufacturing system is located at the Control and Automation Laboratory (LCA), located at the Federal University of Rio de Janeiro (UFRJ).

The manufacturing system is a cube assembly system, where the different cube halves shown in Figure 3.1 are put together to form cubes.

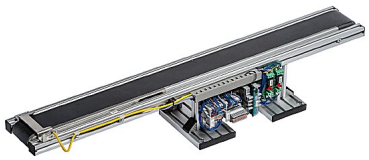
Figure 3.1: Cube halves.

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 and storing), 6 Units are used. These units can be seen in Figure 3.2.

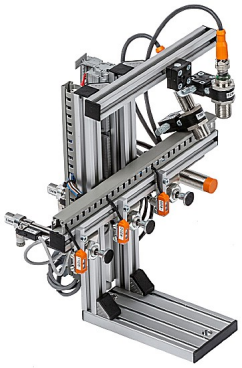
[1]All images from the Christiani modules are present on its sales catalog, available at `www.christiani.de`. All rights are reserved to Christiani.



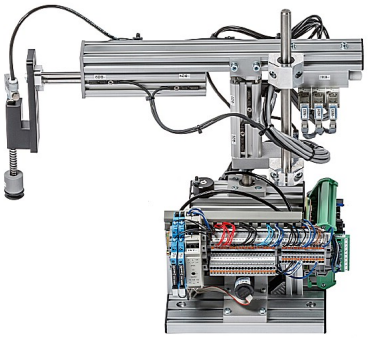
(a) Magazine Unit



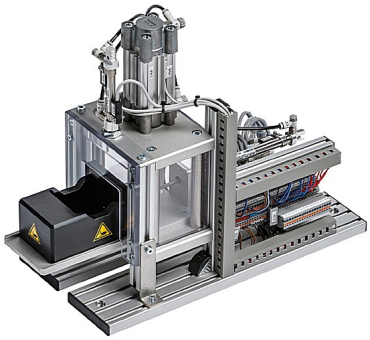
(b) Conveyor Belt.



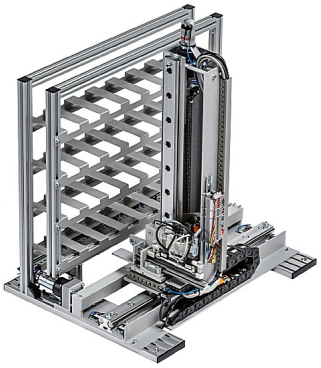
(c) Sorting Unit.



(d) Handling Unit.



(e) Assembly Unit.



(f) Storage Unit.

Figure 3.2: Units of the Manufacture System.

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

Remark 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 store the cube halves to be used. There are 2 types of magazines in the system, one to store pieces with connection pins inserted (the pins shown in Figure 3.1) and another to stock pieces without those pins. These magazine units can stack 8 and 10 pieces and will be denominated MAG 1 and MAG 2, respectively. Each magazine has a pneumatic cylinder and a limit switch sensor. The cylinder serves to extract a piece from the bottom of the stack, and the limit switch 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 sensors 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 limit switch of each magazine outputs a *true* value if the stack is empty and *false*, otherwise. Thus, the limit switches are called in this work `MAG 1/2 Empty`, and their localisation on the magazine can be seen in Figure 3.3.

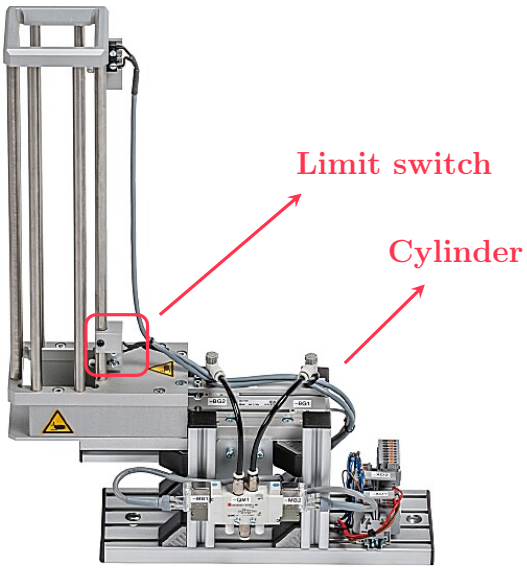


Figure 3.3: Magazine Unit.

3.2 Conveyor Belt

The conveyor belt transports the pieces from a unit to another. It has 2 inputs and 1 output. The inputs are used to turn the belt on, in two possible directions. The output is generated by a presence sensor located at the end of the belt (see Figure 3.4), and 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 is 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`. The input is called `Proximity Sensor End of Conveyor Belt`.

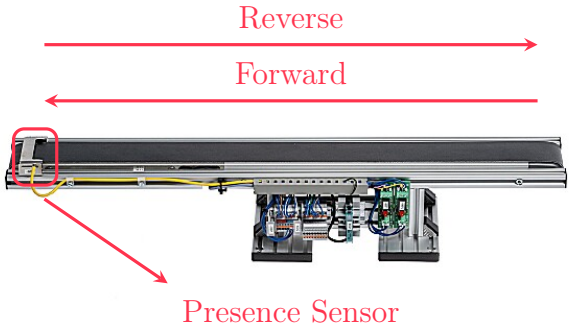


Figure 3.4: Conveyor Belt.

3.3 Sorting Unit

The sorting unit is used to sort the pieces according to their material and colour. In order to denominate 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 the half cube: a distance sensor to identify the orientation of the concavity of the piece, an optic sensor to identify the colour 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 this output is denoted as `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 denoted as `White Color Sensor`. The distance sensor outputs an integer value corresponding to the distance between the piece and the sensor, it is denoted as `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.

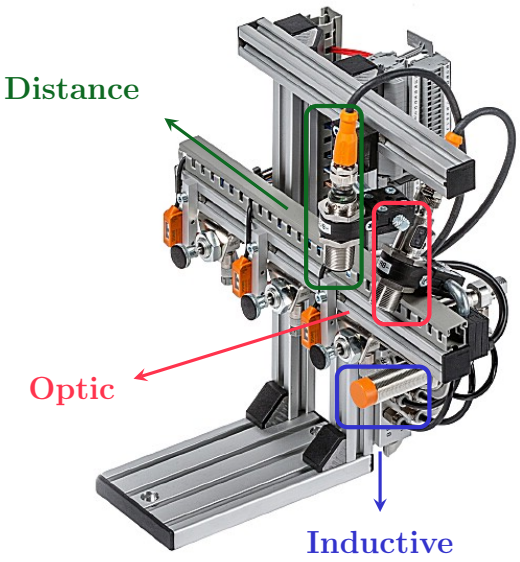


Figure 3.5: Sorting Unit - Identification.

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.

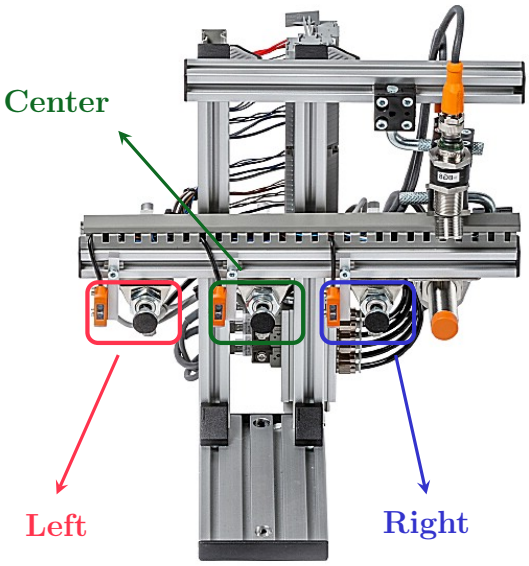


Figure 3.6: Sorting Unit - Discharging.

Differently from MAG 1/2, each cylinder is a *single acting pneumatic cylinder*. When

the corresponding input is equal to *true* it extends, and when it is *false* it is automatically retracted. A tag 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 tags 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 tag 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 another. 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 (ρ, ϕ, z), where ρ is the axial distance, ϕ 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, called "arm" throughout this work, there is a couple of pneumatic cylinders. The behaviour of these cylinders is similar to the behaviour of the cylinders in the sorting unit (*single acting pneumatic cylinder*). These cylinders are placed in each prismatic joint of the arm, thus raising (increasing the height z) and extending (increasing the axial distance ρ) 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 sensors to identify if the are retracted or extended, and tags 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 on the arm's base. This motor has 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 kinds of movements denoted `Turn Arm CW` and `Turn Arm CCW`. In order to identify what is considered CW and CCW it is needed to impose one of those rotation directions. In this work we have 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 ϕ 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 $\phi = 180^o$.

An encoder is used to measure the azimuth angle, but as the angular velocity of the arm is relatively high, and the resolution of the arm is very precise, the output of this encoder is connected to a High Speed Counter, in order to correctly estimate the angle. The configuration of the High Speed Counter can be seen in FLORIANO (2019); PEREIRA (2019).

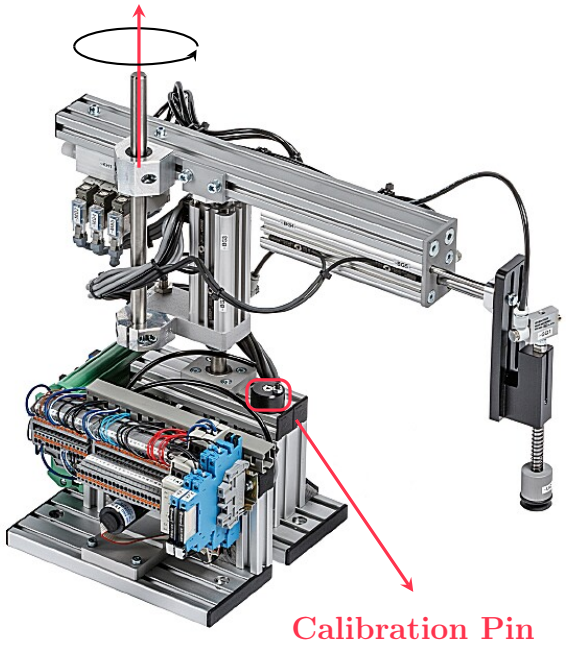


Figure 3.7: Handling Unit.

3.5 Assembly Unit

The assembly unit presses 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, and their respective outputs are called `Safety Door Opened` and `Safety Door Closed`.

Since there is a safety compartment, where the cubes are assembled, there exists a device with the purpose of transporting the pieces from outside to inside of the press 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 that indicate 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.

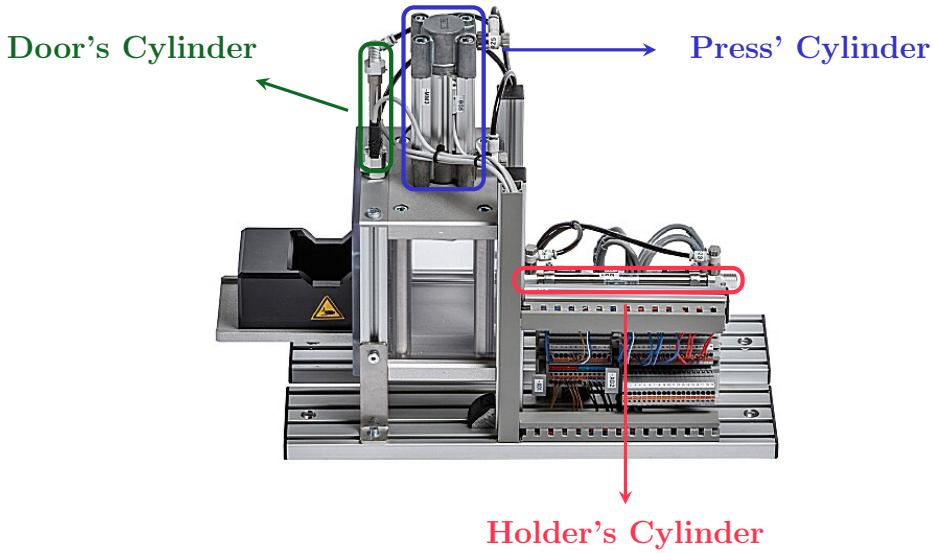


Figure 3.8: Assembly Unit.

3.6 Storage Unit

The storage unit 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 space to store 7 pieces, resulting in 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 that this holder is moved and not the rack itself. There is also another motor 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, Figure 3.9 shows what is considered the Right, Left, Top and Bottom of this unit.

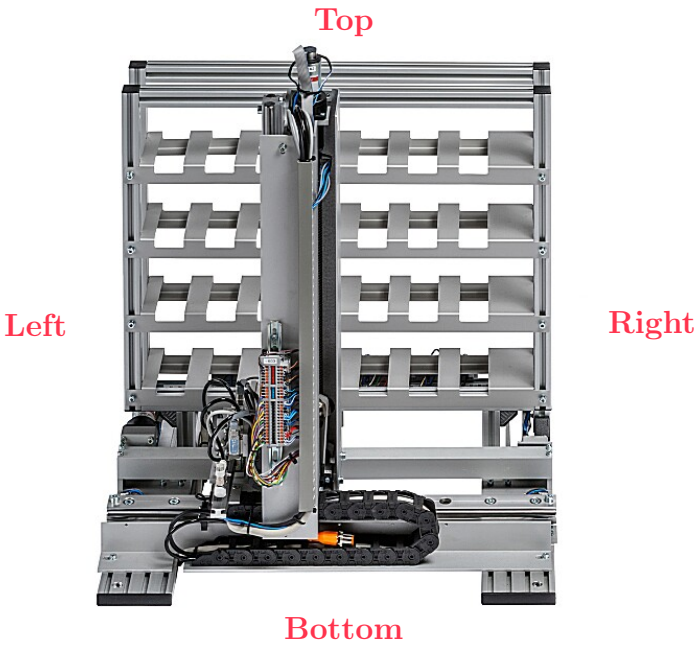


Figure 3.9: Storage Unit.

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, and when the 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 switches 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 indicate if the `Storage Device` is in one of the limits of the rack.

Chapter 4

Control Logic

A CIPN can be used to represent the control of the system, and then this Petri net can be converted into a LD, with the aim of being implemented in a PLC. So, this chapter 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 Control Interpreted Petri net for the manufacturing system

The logic of the control is to use the 6 units presented in chapter 3, to assemble cubes made of a plastic half cube on top of a metallic half cube. 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.

Remark 4.1 *Each Petri net shown in this chapter is a part of a complete Petri net. The complete one is presented in Appendix A. The dotted places/transitions represent places/transitions that belong to other parts of the complete Petri net.*

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

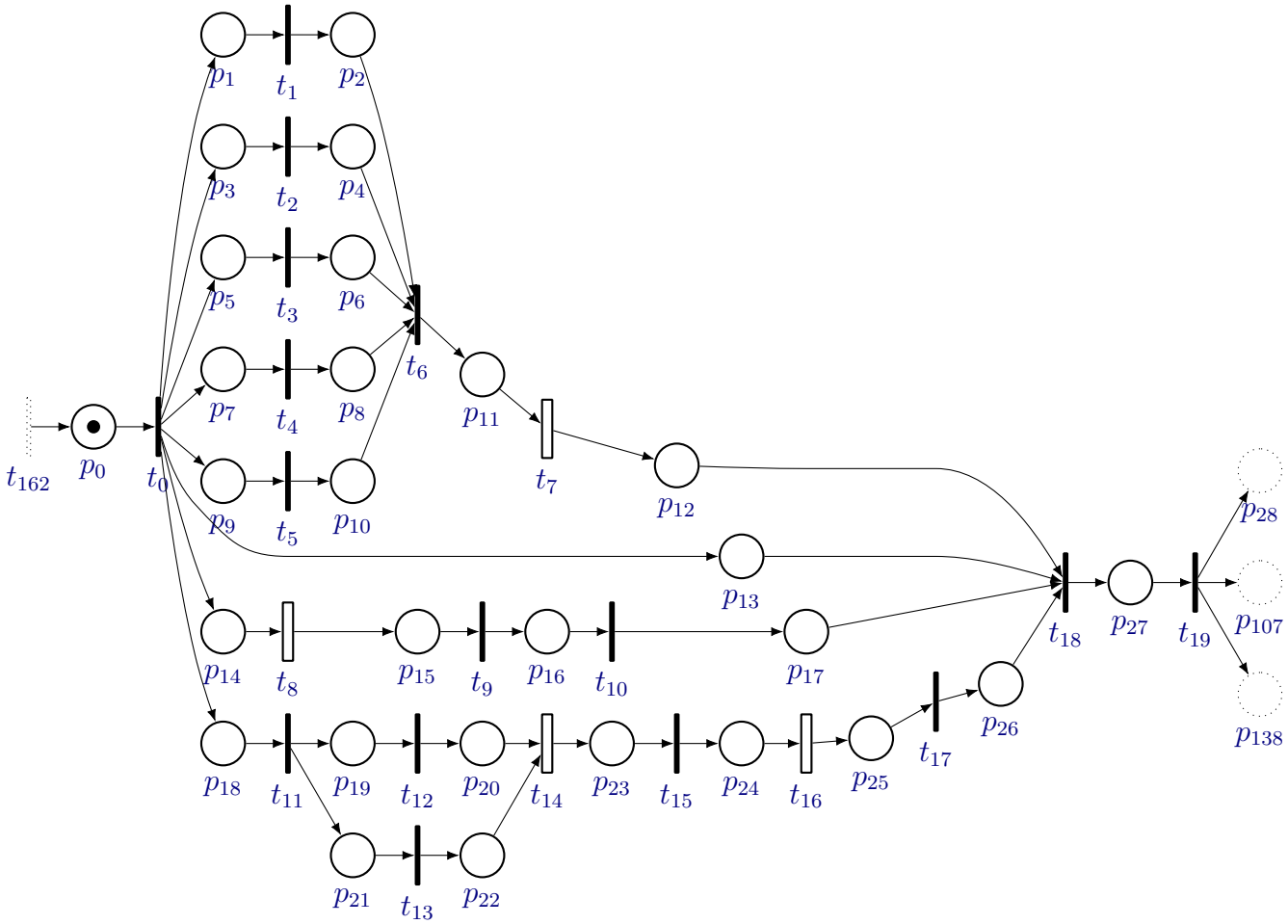


Figure 4.1: Petri net of Initialization module.

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 metallic piece with an upwards concavity the piece continues in the belt until it reaches the end of the belt, waiting to be picked by the arm. Otherwise, the piece is discarded 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}	$\overline{\text{MAG1 Empty}}$
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}	$\overline{\text{Metallic Sensor}}$
t_{27}	$\overline{\text{White Color Sensor}}$
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

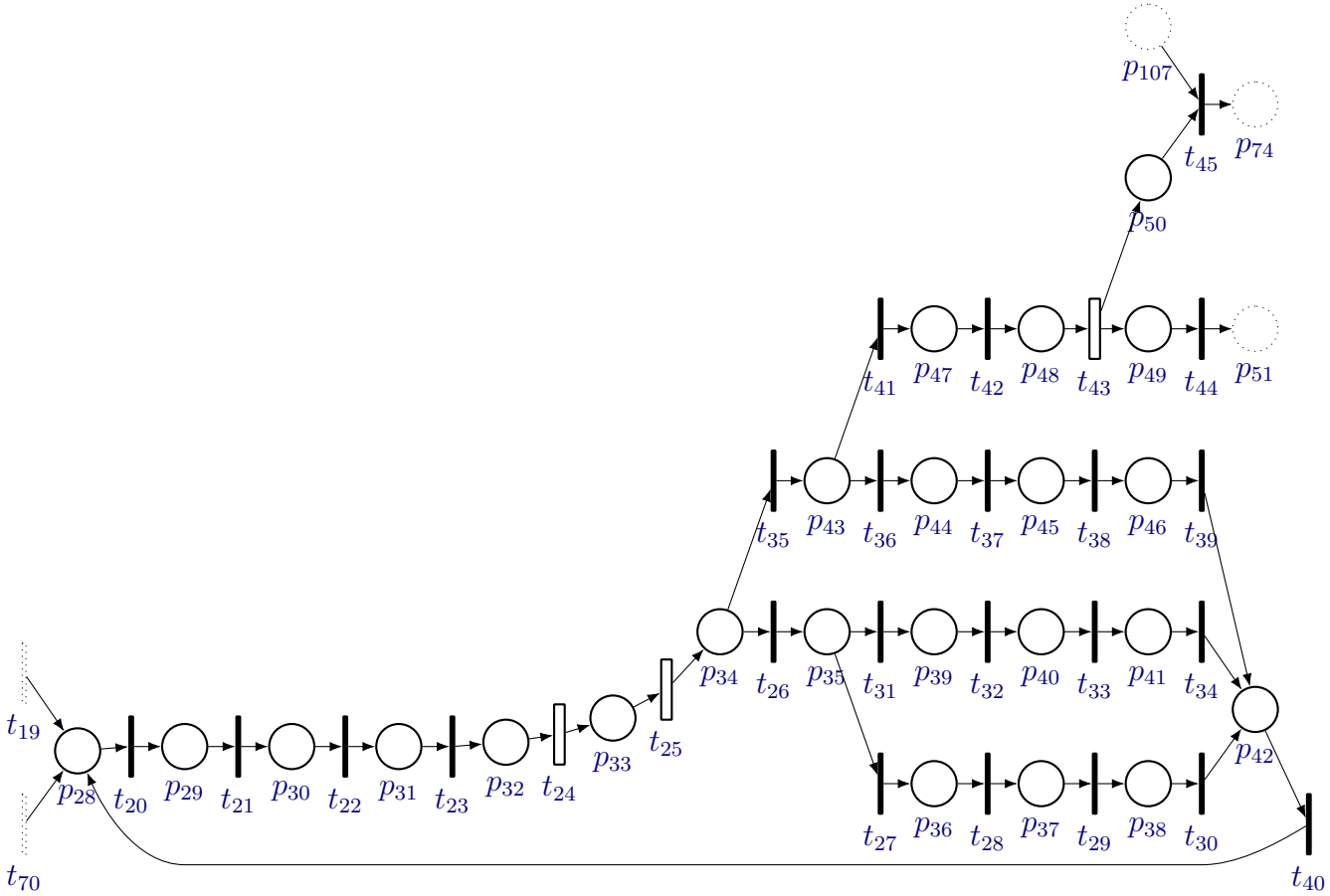


Figure 4.2: Petri net of metal cube half sorting module.

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. 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}	$\overline{\text{MAG2 Empty}}$
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}	$\overline{\text{Metallic Sensor}}$
t_{57}	$\overline{\text{White Color Sensor}}$
t_{58}	↑ Proximity Sensor Right Discharge Cylinder
t_{59}	Right Discharge Cylinder Extended
t_{60}	Right Discharge Cylinder Retracted
t_{61}	$\overline{\text{White Color Sensor}}$
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

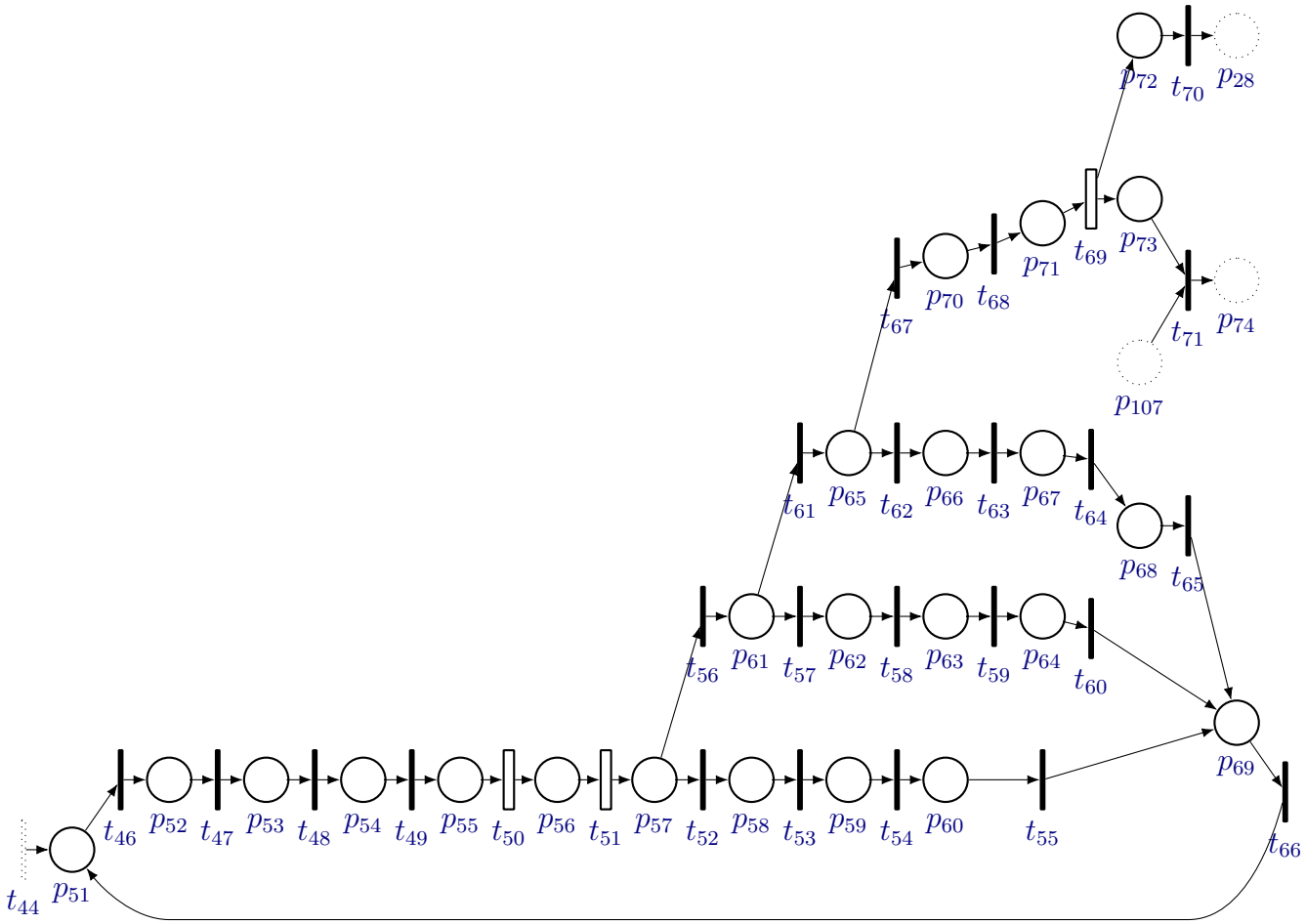


Figure 4.3: Petri net of plastic cube half sorting module.

4.1.4 Arm From Conveyor Belt to Assembly Unit

This module uses the manipulator to remove a piece from the end of the conveyor belt and place 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:=HALFPIECECOUNTER+1

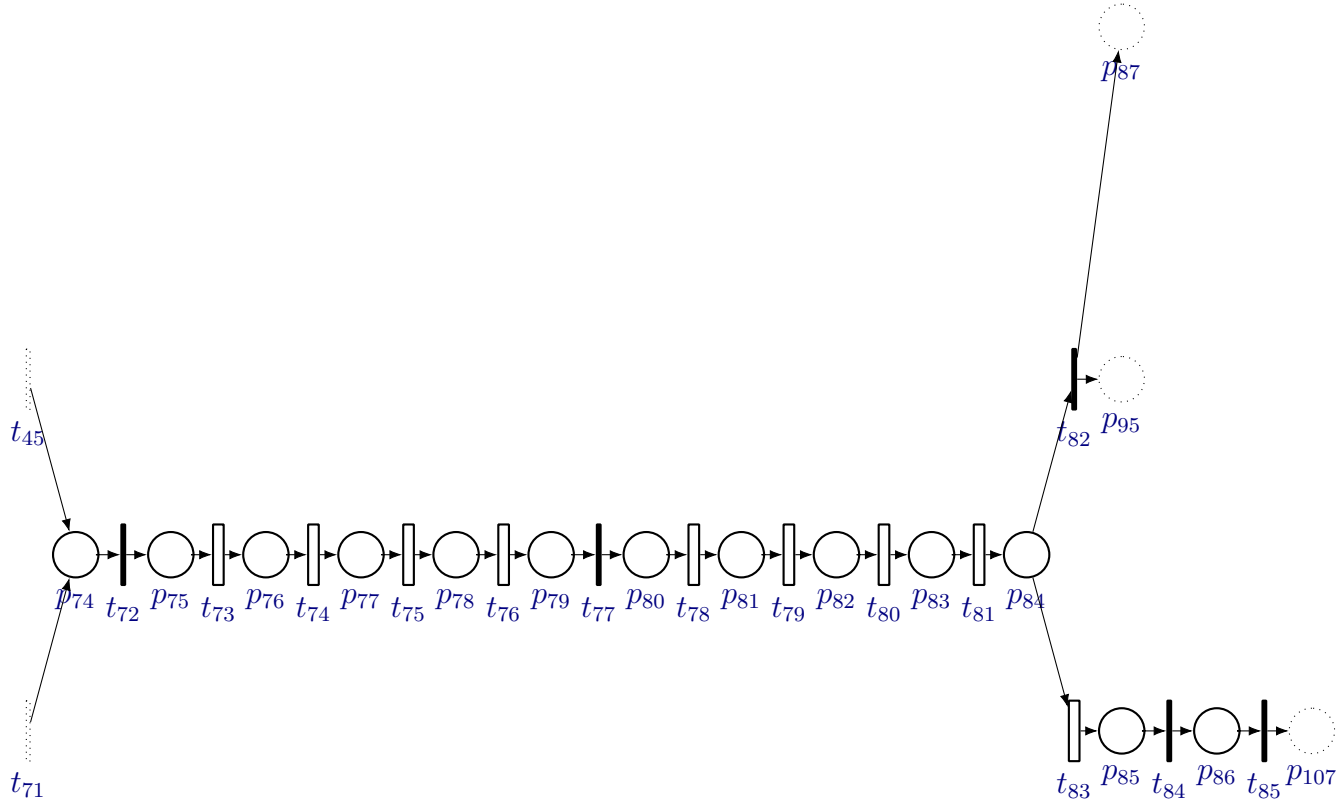


Figure 4.4: Petri net of manipulator taking a cube half from conveyor belt to assembly unit module.

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

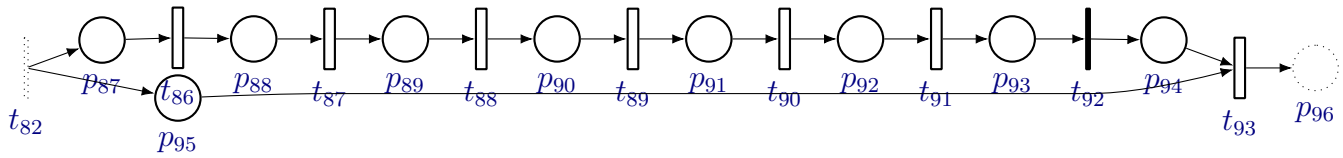


Figure 4.5: Petri net of assembly unit module.

4.1.6 Arm From Assembly Unit To Storage Unit

This module uses the arm to move the cube from the Assembly Unit Holder to the storage device of the Storage Unit. An additional encoder similar to `Storage Unit Horizontal Encoder` was placed just beside this same encoder. In order to help the alignment of the arm with the Storage Device. This new encoder is called `Storage Unit Arm Alignment Encoder`, and is presented 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 Alignement Encoder
t_{97}	ARMCOUNTER <= 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 <= 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

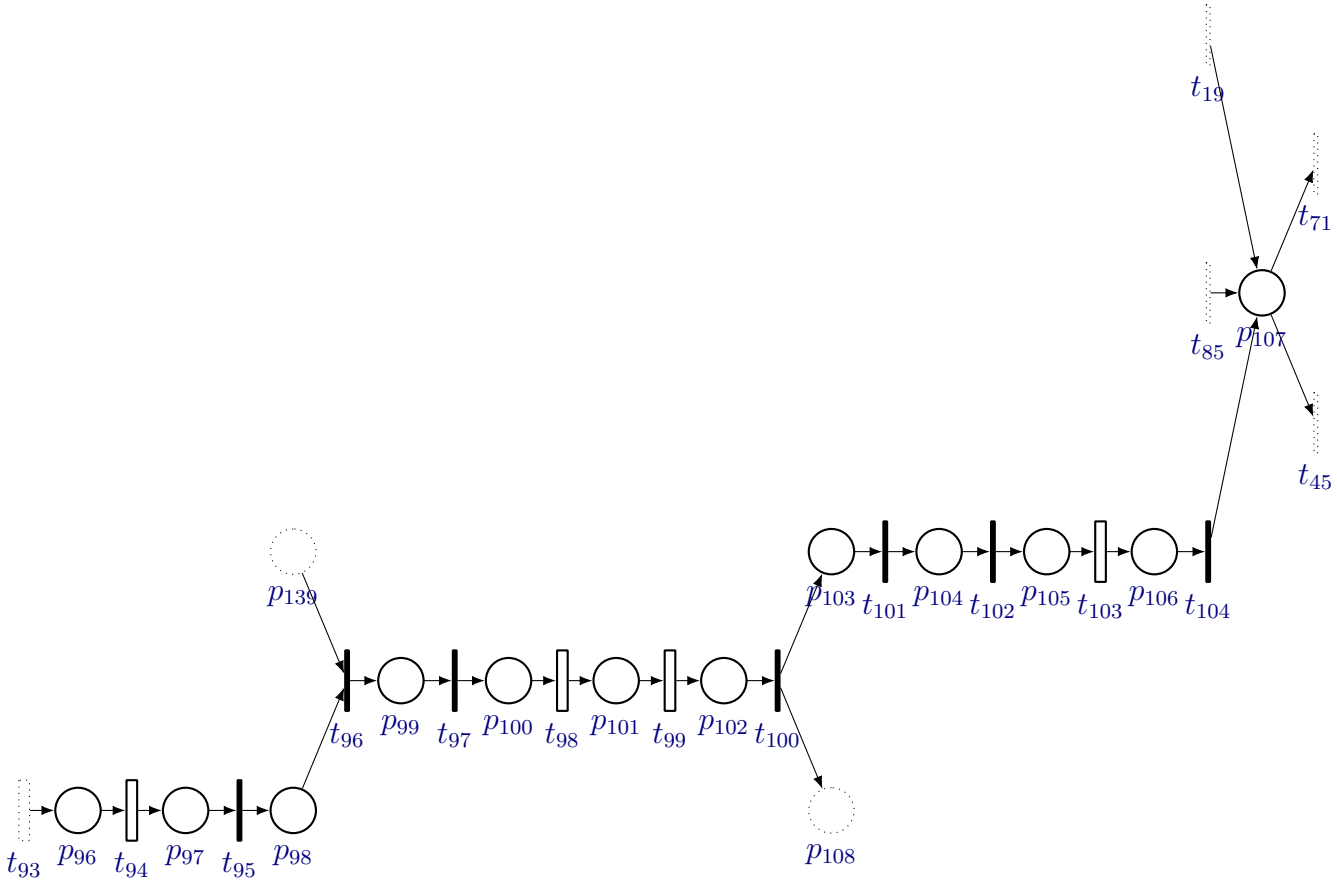


Figure 4.6: Petri net of manipulator taking cube from assembly unit to storage module.

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 the 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}	$\overline{\text{Vertical Encoder}}$
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}	

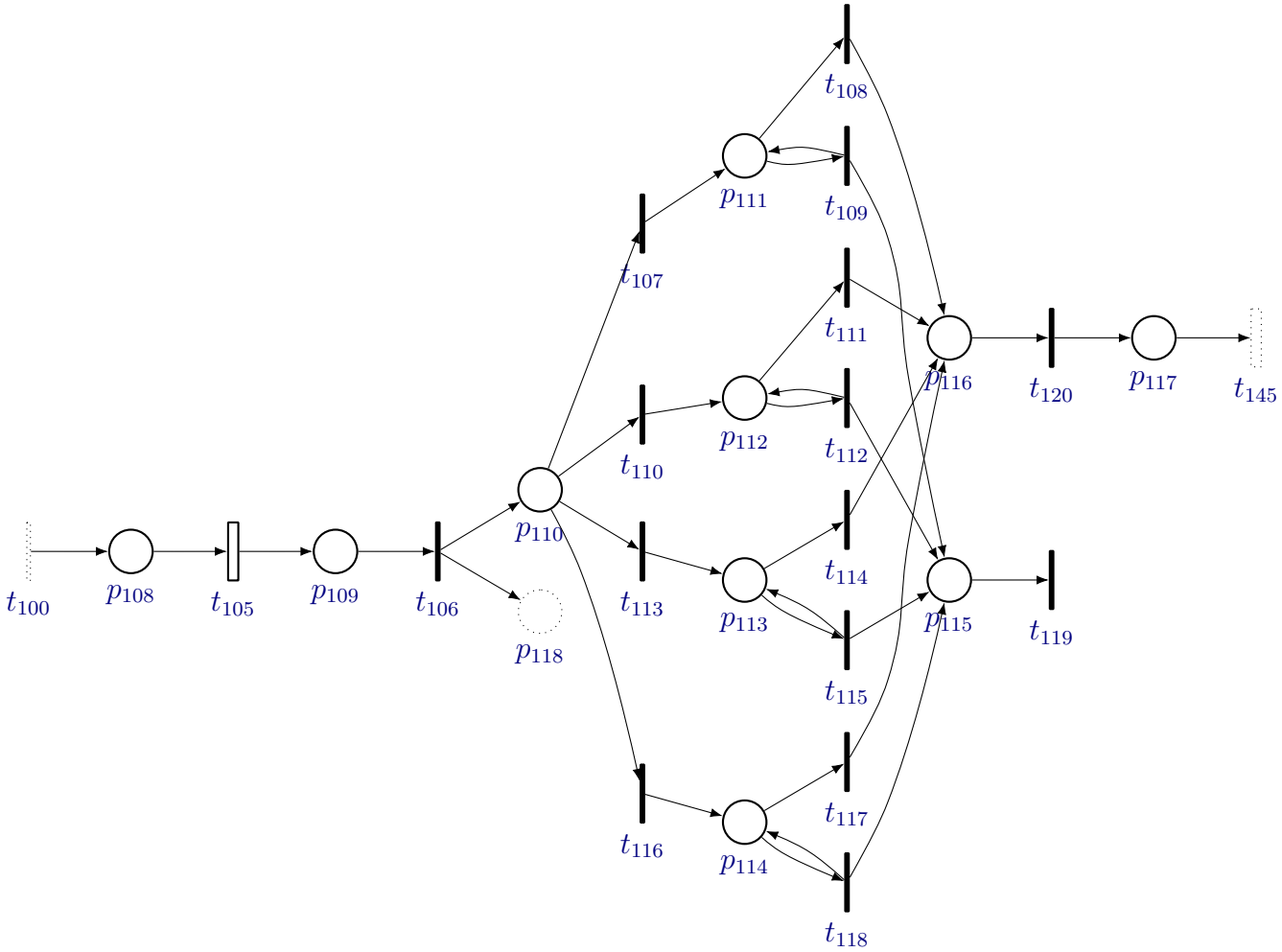


Figure 4.7: Petri net of storage unit positioning module (y-axis).

4.1.8 Storage Unit Positioning (x Axis)

This module sets the horizontal position of the Storage Device. This module and the last module occurs simultaneously. 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}	$\overline{\text{Horizontal Encoder}}$

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

Places	Meaning
p_{118}	COUNTER1:=COUNTER1+1 e COUNTER4:=COUNTER4+1
p_{119}	Move Storage Unit to the Left
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}	

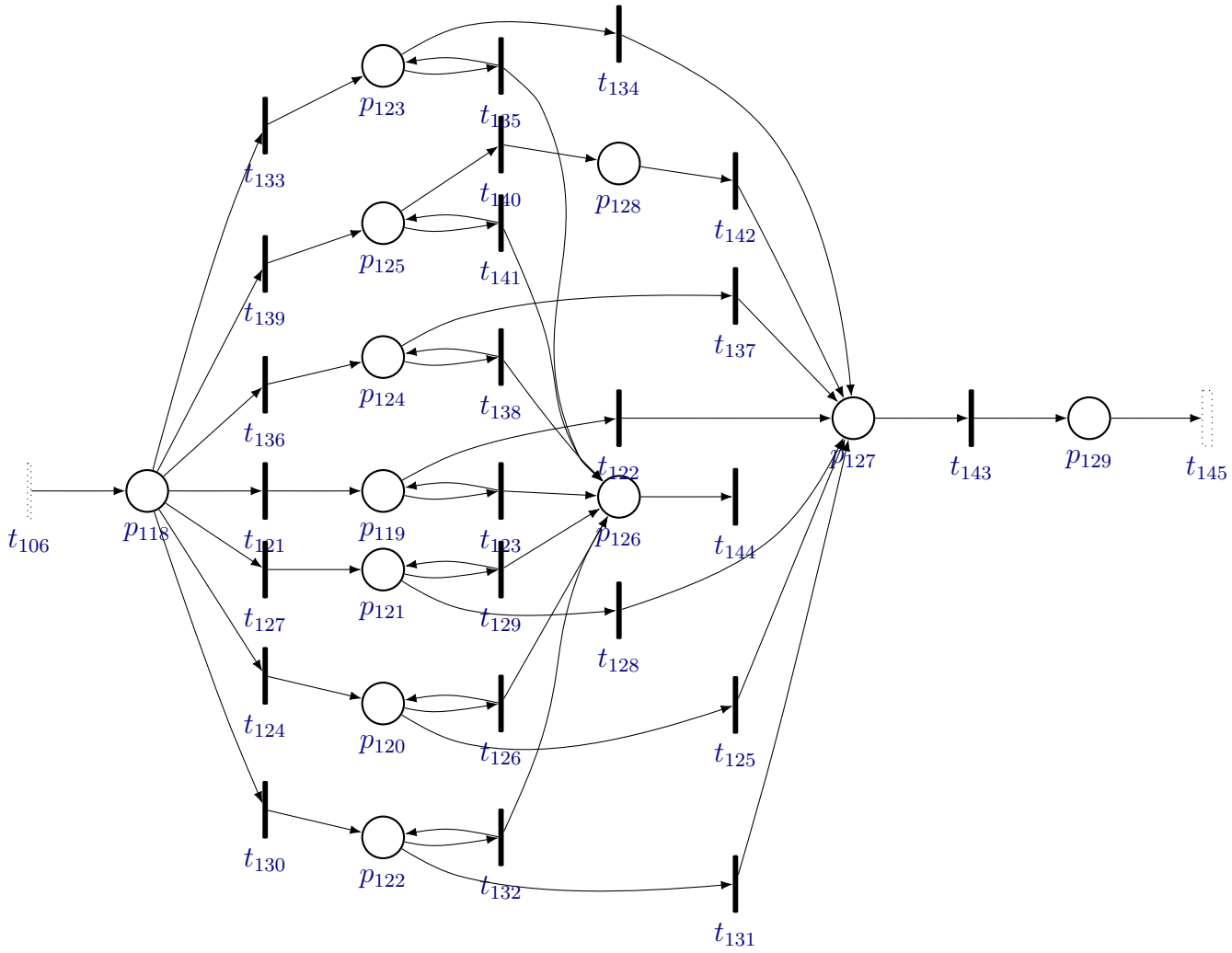


Figure 4.8: Petri net of storage unit positioning module (x-axis).

4.1.9 Cube Storage

This module has the objective of storing the cube in the correct 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*

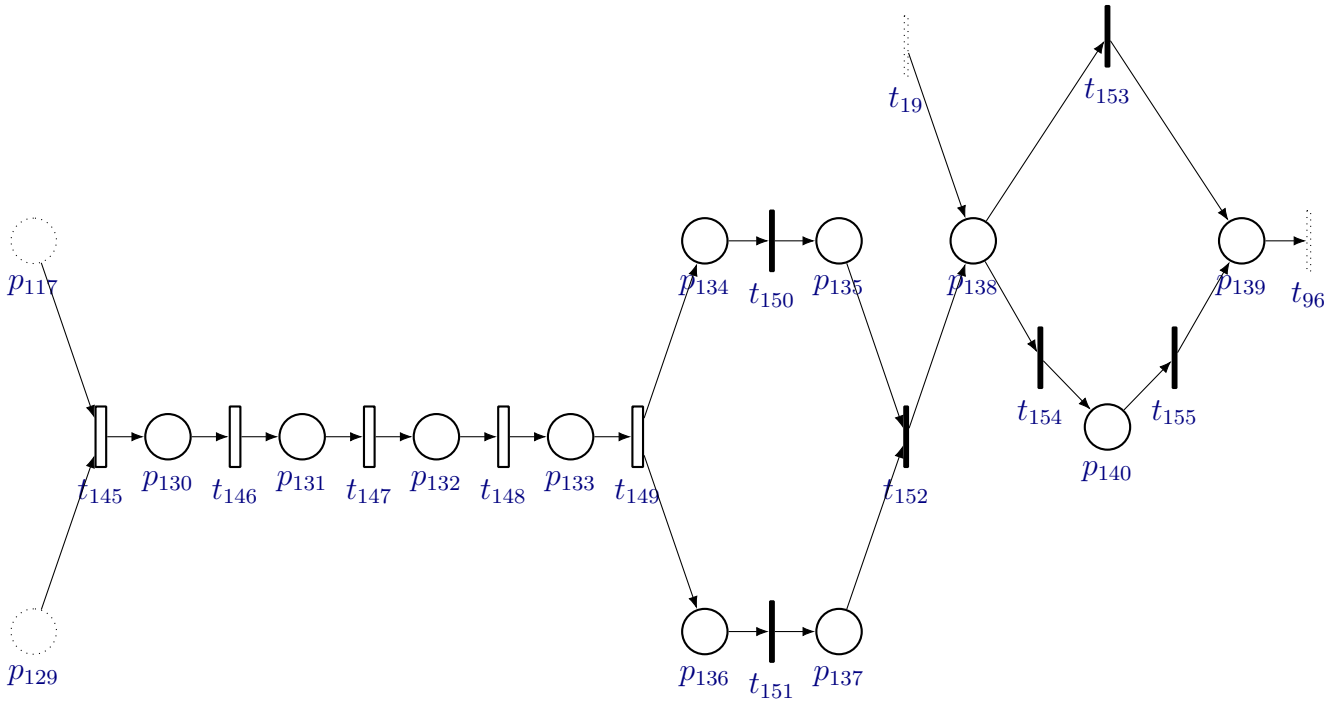


Figure 4.9: Petri net of cube storage module.

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 turned off voluntarily or not, they are going to retract, which can damage the arm. So, in order to prevent these kinds of accidents this module was created. This module creates different behaviours for the turning off of the arm depending on the angle it is. Each behaviour turns the arm to a safe position before retracting the cylinders. A transition is created from every place in all other modules to the first place in this module. This transition corresponds to the will of stopping the system and consequently the arm. For organisation's sake all those 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 angles can be seen in Figure 4.10.

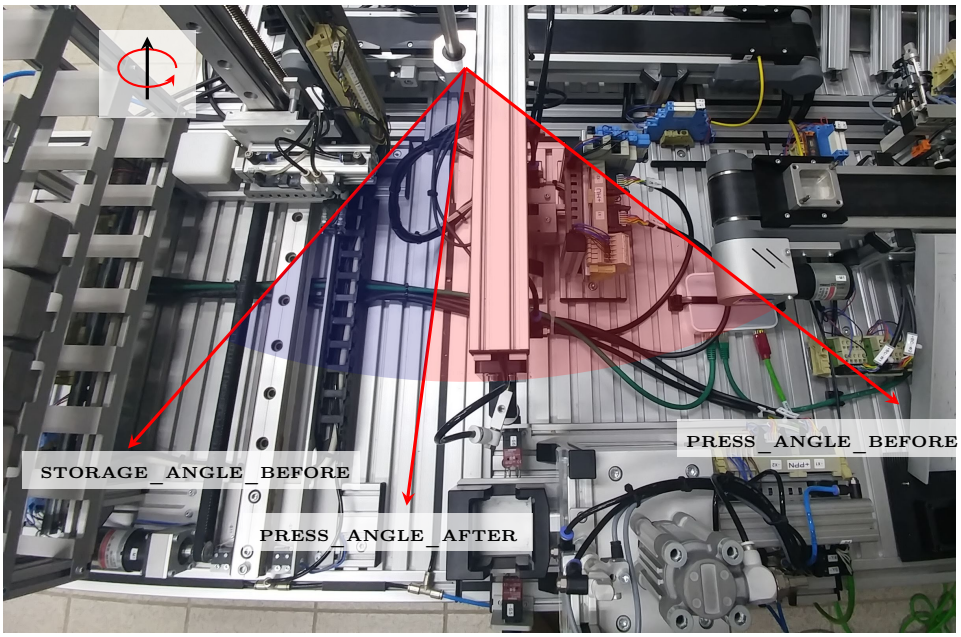


Figure 4.10: Arm Stop Logic Angles

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 >= STORAGE_ANGLE_BEFORE
t_{160}	(ARMCOUNTER >= STORAGE_ANGLE_BEFORE and ARMCOUNTER < PRESS_ANGLE_AFTER) or ARMCOUNTER >= PRESS_ANGLE_BEFORE
t_{161}	Arm Raised and Retracted
t_{162}	Inductive Sensor Arm
t_{163}	ARMCOUNTER >= PRESS_ANGLE_AFTER and ARMCOUNTER < PRESS_ANGLE_BEFORE
t_{164}	Arm Retracted
t_{165}	$\overline{\text{Arm Retracted}}$

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}	

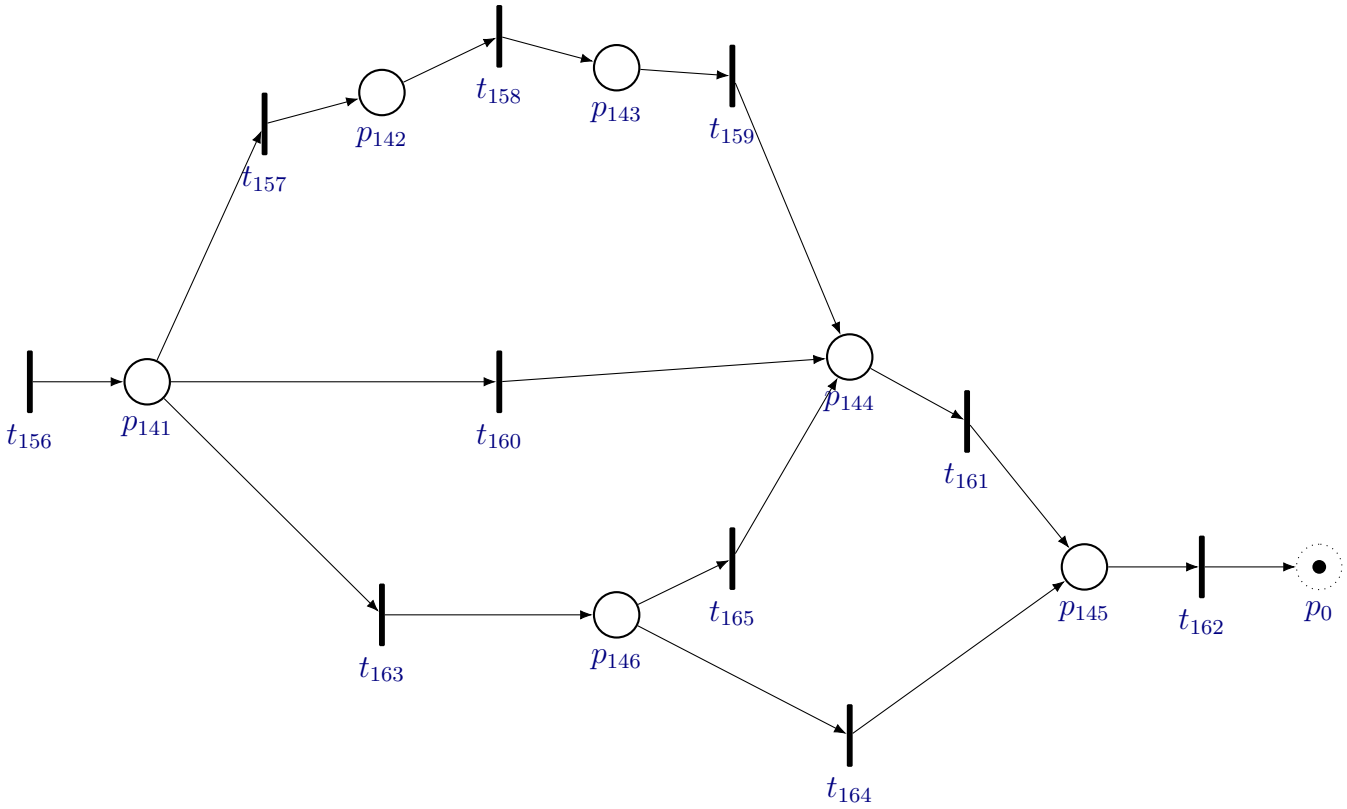


Figure 4.11: Petri net of manipulator Stop Logic module.

4.2 Implementation of the Control

The implementation of the control in this work is carried out using PLCs. The units shown in chapter 3 are divided in two groups. Each group is 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 is 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. 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

The `Distance Sensor` outputs an integer so the variables `ConcUP` and `ConcDWN` were created using the following comparisons:

$$\texttt{ConcUP} = \texttt{Distance Sensor} \geq 1000 \;\&\; \texttt{Distance Sensor} < 10000 \tag{4.1}$$

$$\texttt{ConcDWN} = \texttt{Distance Sensor} \geq 10000 \tag{4.2}$$

The other units (Handling Unit, Assembly Unit and Storage Unit) are connected to the second PLC, identified as Handling-Assembly-Storage. 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 Alignement 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_PRESSHIG
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 section 4.1 into LD the method presented in subsection 2.6.2 was used. 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 consequently the connection between the two PLCs, the tutorials shown in section 3.4 of 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

Manufacturing System Identification

In this chapter, the identification process of the controlled system is 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 are described.

5.1 Data Acquisition

There are multiple ways to obtain the values of the inputs/outputs of a system. The data acquisition methods can be divided in two categories. In the first one, the data is continuously registered, and in the second one 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 fault detection modules. On the other hand, the second one is usually used for offline processes, processes that are computationally expensive and sporadic processes. An example of such processes can be modelling a big system, what can be a resource-intensive task.

Since algorithm 1 takes as input a set of paths, all the data is acquired beforehand. The data can be acquired in batches, and once all the data is collected, the algorithm can be executed.

The most straightforward way to obtain the data from a Siemens PLC is by using datalogs (SIEMENS, a). The Siemens PLC S7-1500 includes function blocks to use inside a LD to store custom data in a Comma Separated Values (CSV) formatted file. This file is saved in a SD card. In order to download this file to a computer, the SD

card can be connected to a PC, or the file can be downloaded using a web browser, if a Web Server is configured in the PLC.

The five function blocks used to log data are called `DataLogCreate`, `DataLogOpen`, `DataLogWrite`, `DataLogClose` and `DataLogDelete`. These blocks are shown in the LD of Figures 5.1, 5.2, 5.3, 5.4 and 5.5.

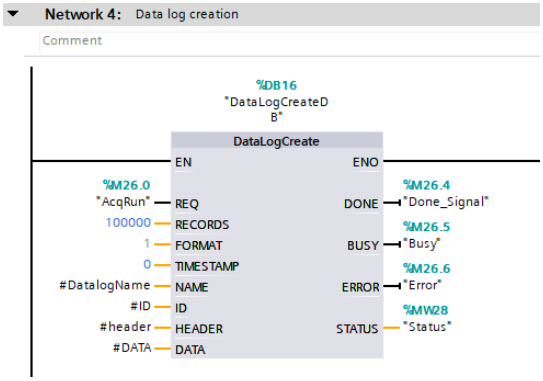


Figure 5.1: DataLogCreate block.

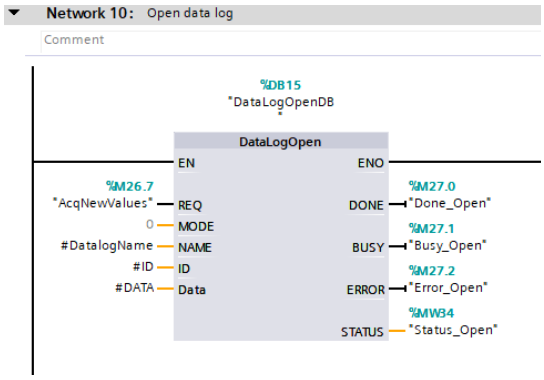


Figure 5.2: DataLogOpen block.

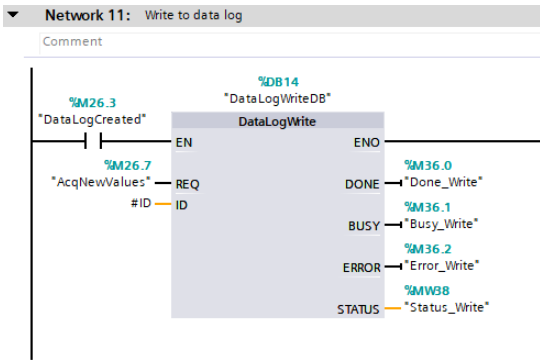


Figure 5.3: DataLogWrite block.

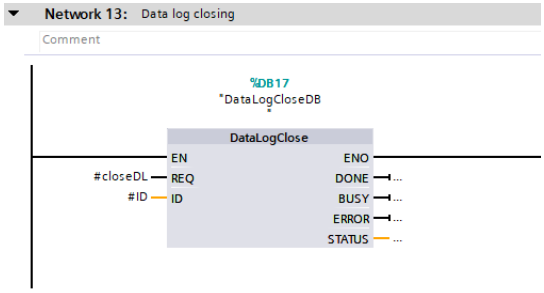


Figure 5.4: DataLogClose block.

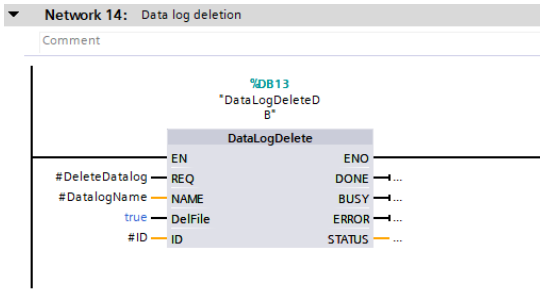


Figure 5.5: DataLogDelete block.

The DataLogCreate block creates the .csv file. The DataLogOpen block opens the file, allowing it to be written. The DataLogWrite block writes the data to the file. The DataLogClose block closes the file, forbidding it to be written. The DataLogDelete block deletes the file if it is needed.

The inputs and outputs of these blocks are used to determine the information about the .csv file and the data to be stored. So, the function of these inputs and outputs will be presented in the following paragraphs.

The REQ input triggers the action to be performed using the rising edge of its corresponding variable. In order to identify the file to be manipulated, the ID and NAME inputs are used. These inputs use a unique id number and a string, respectively. This string is used as the name of the .csv stored in the SD card.

The DATA input receives the data to be stored in the .csv file. The data is structured in a variable of type struct. The struct type can contain variables of different type and size, for instance: a struct can contain a boolean, an int, a word and a string.

The HEADER input is a string to be *prepended* to the first line of the .csv file. The first line of the file identifies the stored variables. As this string variable will be part of the .csv file, it is needed to include commas "," inside it to separate the identifiers of the variables.

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

The TIMESTAMP input is used to identify if a timestamp column will be inserted or not in the .csv. A boolean variable is used to set the column. If the variable is *true* the column is added, otherwise it is not.

The outputs DONE, BUSY, ERROR and STATUS are not used in this work, but they can be used to identify the status of the action to be performed. The status of each data log function block can be the following: the action is being performed (BUSY), the action is done (DONE) or an error occurred (ERROR). Each status has its corresponding boolean output. If an error occurred, STATUS outputs a code to identify the error. This code is used for troubleshooting and can be found in the Siemens manual, SIEMENS (a).

In order to organise the data used for all these blocks, a *DataBlock* was used. The structure of this *DataBlock* is shown in Figure 5.6. We can see in this *DataBlock* the main variables used to create and write the log data: DATA, HEADER, ID and NAME.

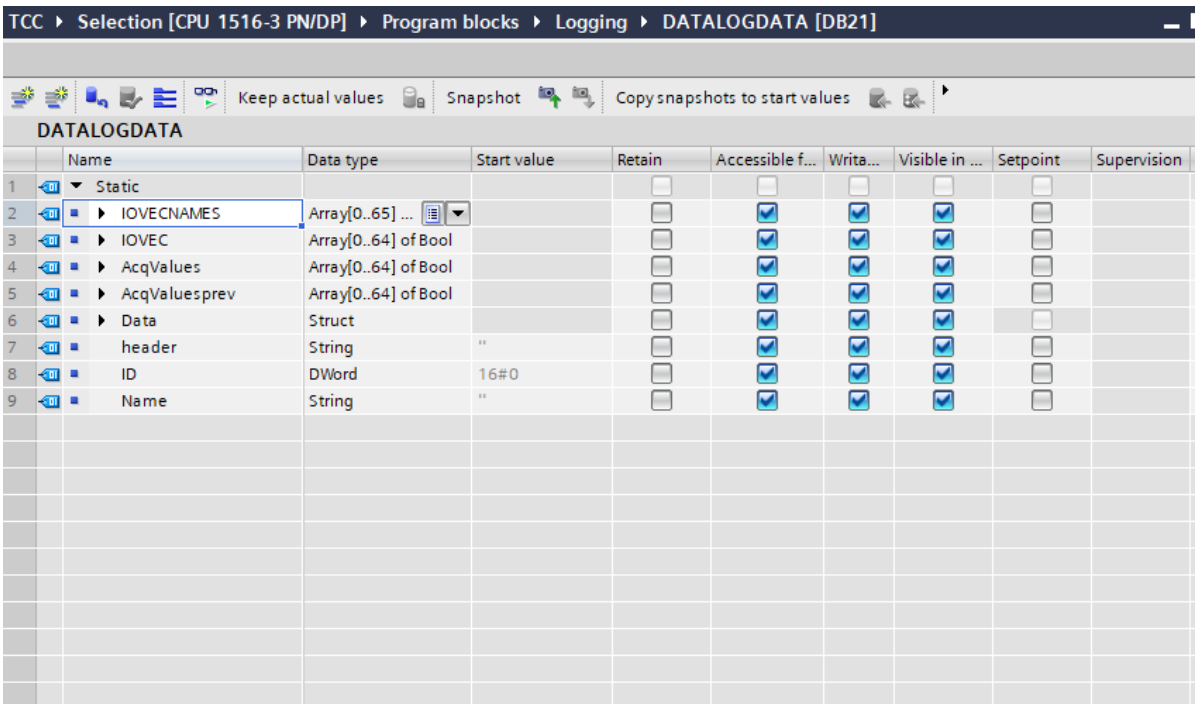


Figure 5.6: Example of DataBlock used to log data.

In this work, we need to store the input/output vectors of the controller. The *IOvectors* (input/output vectors) are composed by boolean variables. One restriction to the storage of these vectors is that there must not be two consecutive vectors with the same data.

In order to achieve the needs of the project, a function block was created to be used in the LD. This new function block uses the data log function blocks from Figures 5.1 to 5.5 and some additional logic. This function block is called *LOGDATA*, and it is depicted in Figure 5.7.

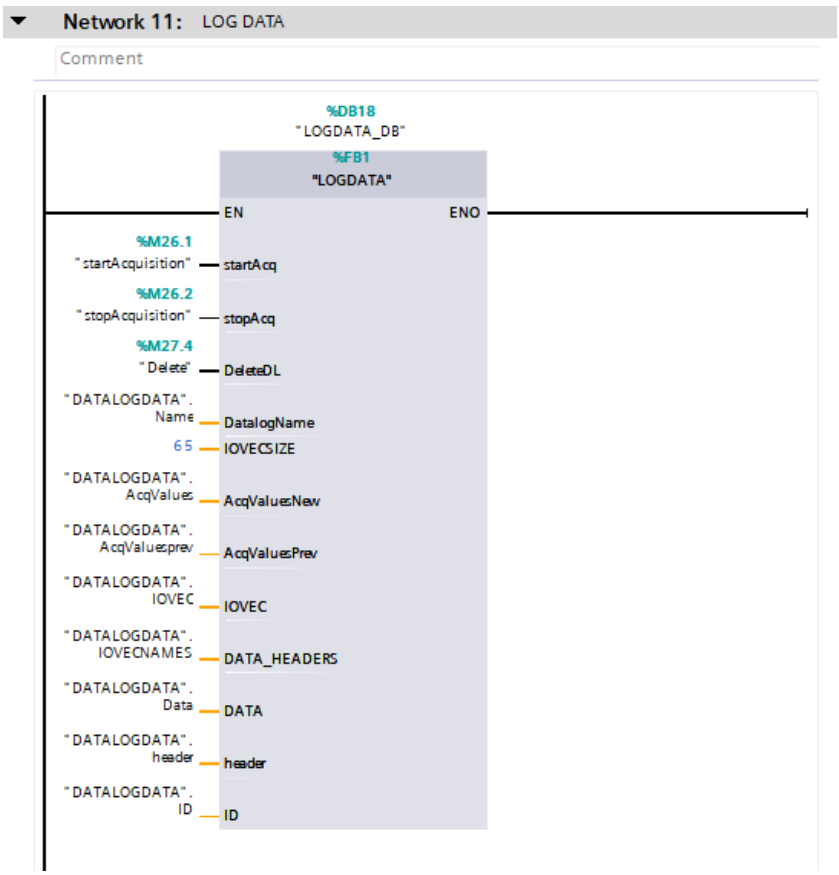


Figure 5.7: LOGDATA block.

The 12 inputs of the *LOGDATA* block will be briefly described in the following paragraphs. And after that, the logic used to implement the block will be presented.

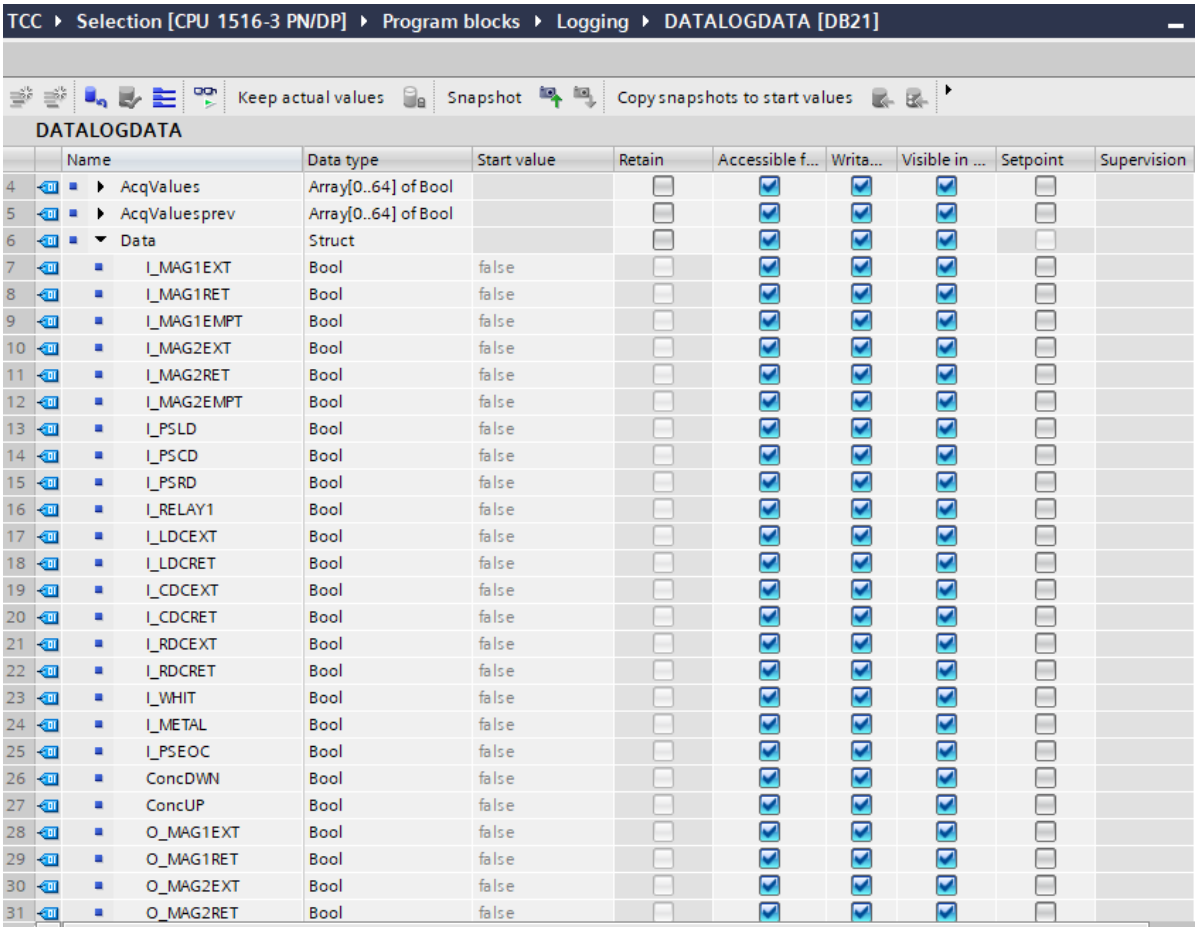
The `startAcq`, `stopAcq` and `DeleteDL` inputs are used to start and stop the acquisition and to delete the `.csv` file, respectively. The `DatalogName` input represents the name of the file. `IOVECSIZE` identifies the number of variables to be stored. In order to avoid two equal consecutive *IOvectors*, two arrays are created : `AcqValuesNew` and `AcqValuesPrev`. These two variables store the current *IOvector* and the preceding one, respectively. These arrays are compared to each other, and if they differ, `AcqValuesNew` is stored in the `.csv` file. The value of these arrays is changed from inside the block via an update block.

`IOVEC` is also an input that is changed from inside the block. It is used as a buffer for the vectors. The data of this buffer is periodically copied to the variable connected to the `DATA` input. `DATA`, in turn, is internally connected to the `DATA` input of the data log function blocks.

Instead of writing the tags of the variables in the `HEADER` input string, a `DATA_HEADERS`

input is created. An array of strings of size `IOVECSIZE` is created and connected to `DATA_HEADERS`. This array contains the tags of all variables to be stored in `DATA`. The contents of the array are concatenated with commas placed between them, and a new string is created. This new string is assigned to the variable *header*, connected to the `HEADER` of the data log function blocks.

An example of the data stored in this work can be seen in Figure 5.8. In this figure is possible to notice that the data structure is composed of variables whose tags are shown in section 4.2.



	Name	Data type	Start value	Retain	Accessible f...	Writa...	Visible in ...	Setpoint	Supervision
4	AcqValues	Array[0..64] of Bool		☐	☑	☑	☑	☐	
5	AcqValuesprev	Array[0..64] of Bool		☐	☑	☑	☑	☐	
6	Data	Struct		☐	☑	☑	☑	☐	
7	I_MAG1EXT	Bool	false	☐	☑	☑	☑	☐	
8	I_MAG1RET	Bool	false	☐	☑	☑	☑	☐	
9	I_MAG1EMPT	Bool	false	☐	☑	☑	☑	☐	
10	I_MAG2EXT	Bool	false	☐	☑	☑	☑	☐	
11	I_MAG2RET	Bool	false	☐	☑	☑	☑	☐	
12	I_MAG2EMPT	Bool	false	☐	☑	☑	☑	☐	
13	I_PSLD	Bool	false	☐	☑	☑	☑	☐	
14	I_PSCD	Bool	false	☐	☑	☑	☑	☐	
15	I_PSRD	Bool	false	☐	☑	☑	☑	☐	
16	I_RELAY1	Bool	false	☐	☑	☑	☑	☐	
17	I_LDCEXT	Bool	false	☐	☑	☑	☑	☐	
18	I_LDCRET	Bool	false	☐	☑	☑	☑	☐	
19	I_CDCEXT	Bool	false	☐	☑	☑	☑	☐	
20	I_CDCRET	Bool	false	☐	☑	☑	☑	☐	
21	I_RDCEXT	Bool	false	☐	☑	☑	☑	☐	
22	I_RDCRET	Bool	false	☐	☑	☑	☑	☐	
23	I_WHIT	Bool	false	☐	☑	☑	☑	☐	
24	I_METAL	Bool	false	☐	☑	☑	☑	☐	
25	I_PSEOC	Bool	false	☐	☑	☑	☑	☐	
26	ConcDWN	Bool	false	☐	☑	☑	☑	☐	
27	ConcUP	Bool	false	☐	☑	☑	☑	☐	
28	O_MAG1EXT	Bool	false	☐	☑	☑	☑	☐	
29	O_MAG1RET	Bool	false	☐	☑	☑	☑	☐	
30	O_MAG2EXT	Bool	false	☐	☑	☑	☑	☐	
31	O_MAG2RET	Bool	false	☐	☑	☑	☑	☐	

Figure 5.8: Example of Data struct.

The code inside the `LOGDATA` block uses the following logic. First the value of `AcqValuesNew` is copied to `AcqValuesPrev`. Then `AcqValuesNew` is updated with the values of the inputs/outputs of the controller. After that, `AcqValuesNew` and `AcqValuesPrev` are compared and if they differ, `AcqValuesNew` is prepared to be stored. Once the data is prepared, it is stored in the file. The last step, storage in the file, is made using the data log function blocks depicted in Figures 5.1 to 5.5. The other steps will be shown

in the next paragraphs.

The copy of the values from the `AcqValuesNew` array is made by using the *MOVE_BLK* function block present in the Siemens PLC. In order to update the values of `AcqValuesNew`, the custom function block called *UpdateValues* is created inside *LOGDATA*. This block can be seen in Figure 5.9. The code of this block is programmed using the Structured Control Language (SCL) language. In Figure 5.10 we can see the code of the *UpdateValues* function block.

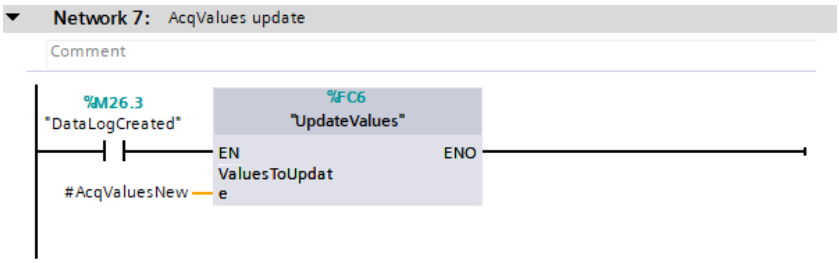


Figure 5.9: UpdateValues block.

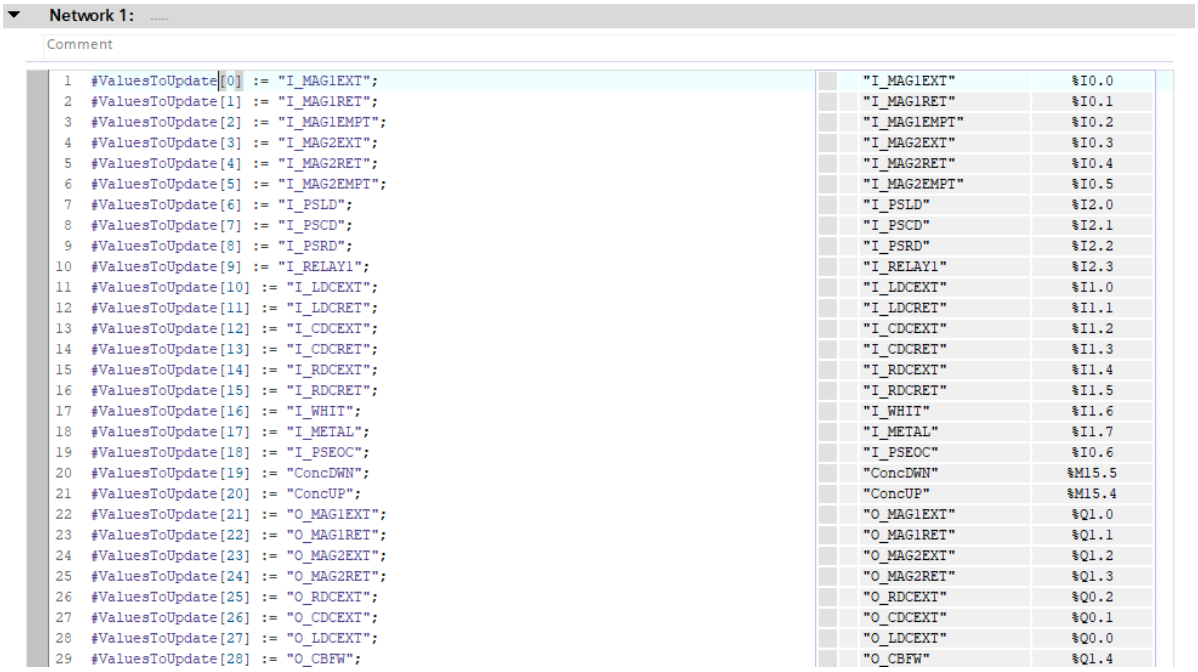


Figure 5.10: Code inside UpdateValues block.

In order to compare `AcqValuesNew` and `AcqValuesPrev`, a *CompareArrays* block is created, in which all the values of both arrays are compared bitwise. And if they are different, the `AcqValuesNew` is copied to the temporary variable input to `IOVEC`, the buffer. This logic can be seen in Figure 5.11.

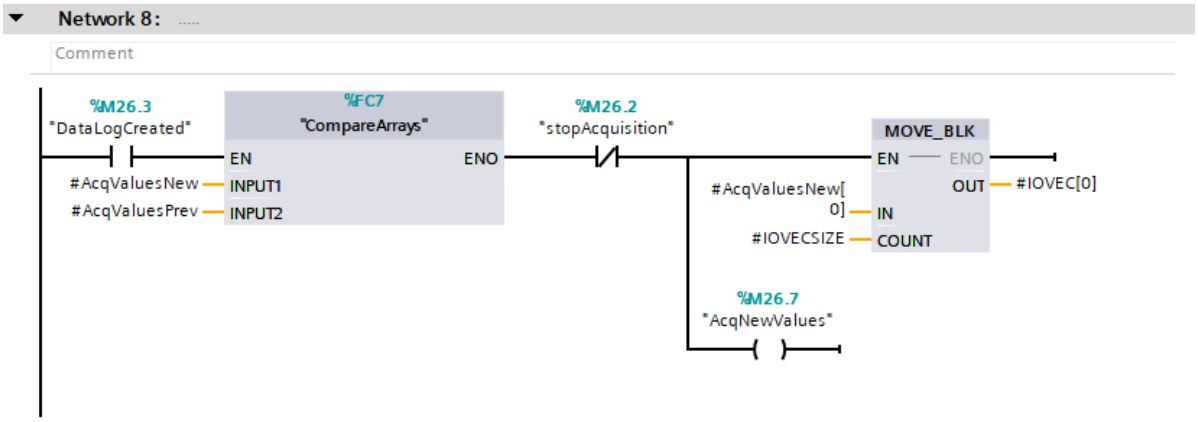


Figure 5.11: CompareArrays block.

Before using the blocks depicted in Figures 5.1 to 5.5 to create the `.csv` file and store the data, the data in the temporary variable `IOVEC` is copied to `DATA`. The copy is carried out by using the custom function block *PutInDataStruct* shown in Figure 5.12. The code of this block is also programmed using SCL and it is represented in Figure 5.13.

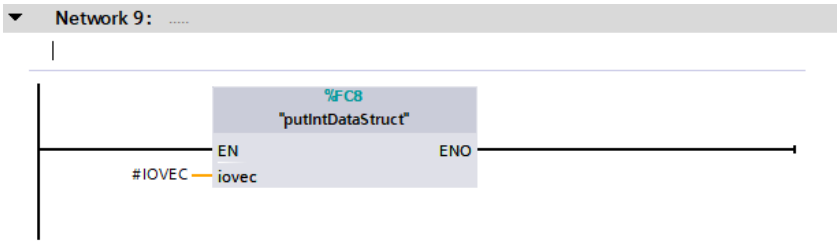


Figure 5.12: PutInDataStruct block.

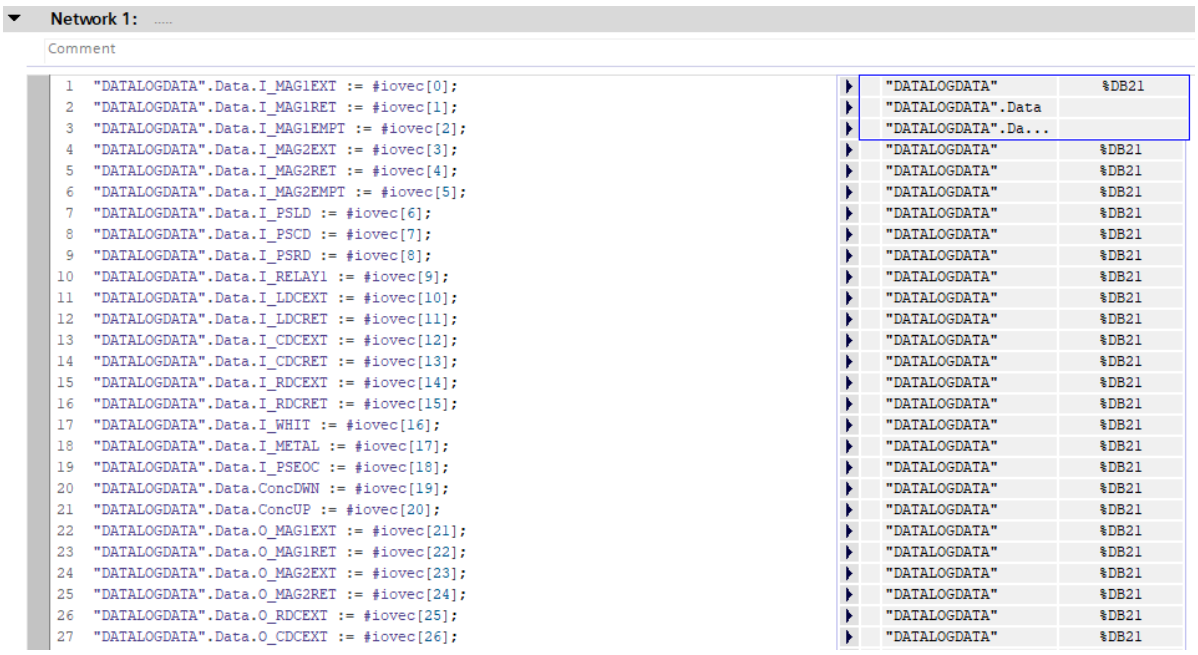
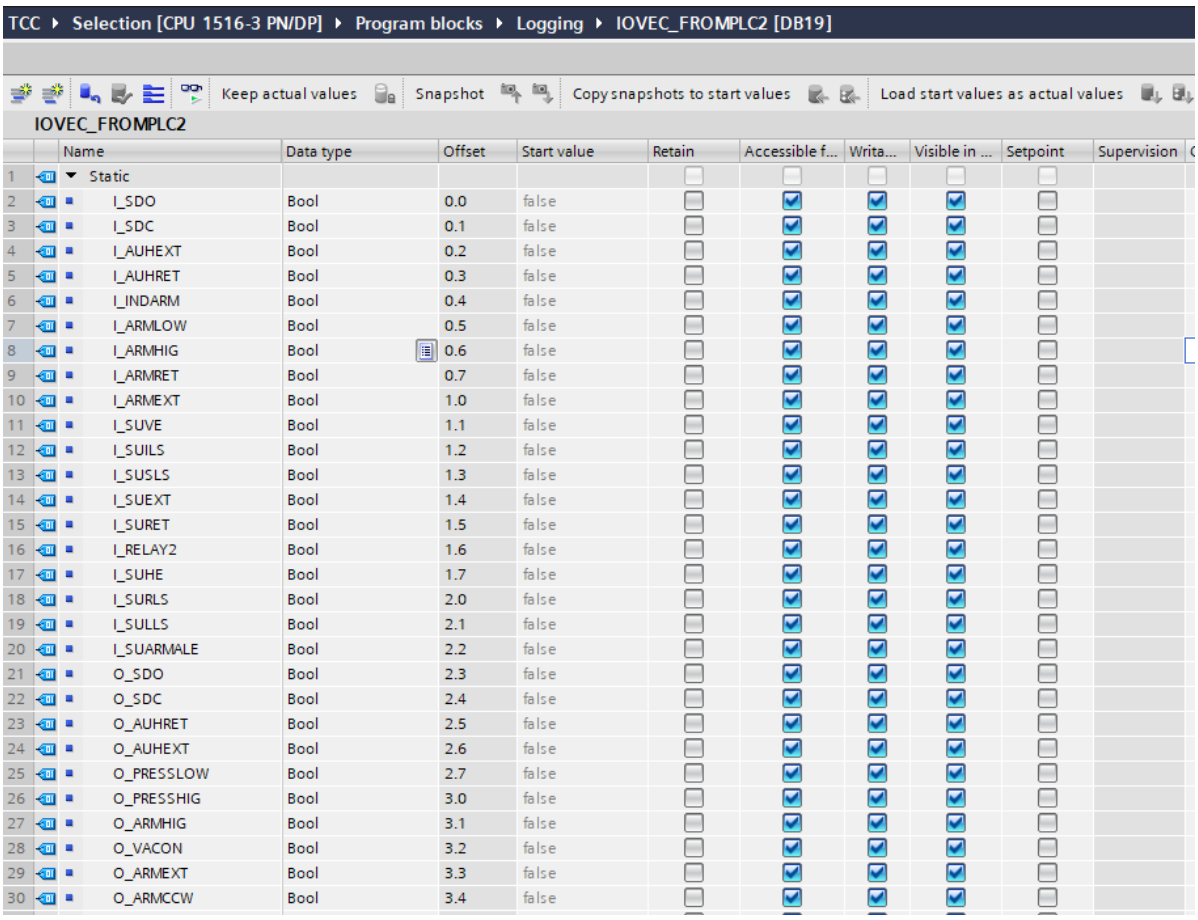


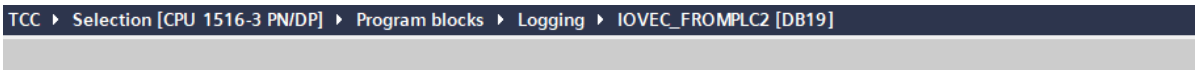
Figure 5.13: Code inside PutInDataStruct block.

Remark 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. Two Data blocks are used to store the inputs and outputs of the Handling-Assembly-Storage PLC and send/receive data using those function blocks. Each data block is located in a different PLC. The aspect of these data blocks can be seen in Figures 5.14a and 5.14b*



	Name	Data type	Offset	Start value	Retain	Accessible f...	Writa...	Visible in ...	Setpoint	Supervision
1	Static									
2	I_SDO	Bool	0.0	false						
3	I_SDC	Bool	0.1	false						
4	I_AUHEXT	Bool	0.2	false						
5	I_AUHRET	Bool	0.3	false						
6	I_INDARM	Bool	0.4	false						
7	I_ARMLOW	Bool	0.5	false						
8	I_ARMHIG	Bool	0.6	false						
9	I_ARMRET	Bool	0.7	false						
10	I_ARMEXT	Bool	1.0	false						
11	I_SUVE	Bool	1.1	false						
12	I_SUILS	Bool	1.2	false						
13	I_SUSLS	Bool	1.3	false						
14	I_SUEXT	Bool	1.4	false						
15	I_SURET	Bool	1.5	false						
16	I_RELAY2	Bool	1.6	false						
17	I_SUHE	Bool	1.7	false						
18	I_SURLS	Bool	2.0	false						
19	I_SULLS	Bool	2.1	false						
20	I_SUARMALE	Bool	2.2	false						
21	O_SDO	Bool	2.3	false						
22	O_SDC	Bool	2.4	false						
23	O_AUHRET	Bool	2.5	false						
24	O_AUHEXT	Bool	2.6	false						
25	O_PRESSLOW	Bool	2.7	false						
26	O_PRESSHIG	Bool	3.0	false						
27	O_ARMHIG	Bool	3.1	false						
28	O_VACON	Bool	3.2	false						
29	O_ARMEXT	Bool	3.3	false						
30	O_ARMCCW	Bool	3.4	false						

(a) IOVEC_FROMPLC2 DataBlock.



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

(b) IOVEC_FROMPLC2 DataBlock - Continuing.

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

5.2 Model Identification

Once the data is logged, the `.csv` file can be downloaded from a Web Server or from the SD card. Following the steps shown in the Siemens Manual, SIEMENS (b), it is possible to configure a web server in the Siemens PLC S7-1500. And once the web server is configured, the file can be downloaded in different ways, most of them are shown in section 3.13 of the same manual, SIEMENS (b). In this work, the `.csv` file was downloaded from the terminal of a computer connected to the same network of the PLC. The commands that can be used to download the file from the terminal are the following:

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

and

```
$ curl -k "https:///192.168.2.132/Filebrowser?Path=/DataLogs/
  name_of_the_file.csv&RAW" -H "Referer: https://192.168.2.132/Portal/
  Portal.mwsl?PriNav=Filebrowser&Path=/DataLogs/"
```

Where the address "192.168.2.132" should be the address of the PLC in which the web server is running and "name_of_the_file.csv" should be the name of the file.

Once the `.csv` file is downloaded to the PC, the paths can be obtained from the data and the identification algorithm can be executed. The acquisition of the paths and the identification algorithm, algorithm 1, were implemented in python.

Since the identification is performed by 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 repeated in the file another path is created. Once the paths are obtained they are modified using Equations 2.3 and 2.4.

A brief example of this method of path acquisition can be presented. Consider the observed data of the example shown in section 2.7, is in the following `.csv` file:

Listing 5.1: CSV file generated from example of section 2.7.

```
1 SeqNo,1,2,3
  1,1,0,0
3 2,1,1,0
  3,0,1,1
5 4,0,0,0
  5,0,0,1
7 6,1,0,0
  7,0,0,0
9 8,1,1,0
```

```
9 ,0 ,1 ,1
10 ,0 ,0 ,0
11 ,1 ,0 ,0
12 ,0 ,1 ,1
13 ,1 ,0 ,0
14 ,0 ,0 ,0
15 ,1 ,1 ,0
16 ,0 ,1 ,1
17 ,1 ,1 ,1
18 ,0 ,0 ,0
19 ,0 ,0 ,1
20 ,1 ,1 ,0
//END, , ,
```

Using the method proposed, the vector $[1\ 0\ 0]^T$ is considered as the initial vector and 4 paths are obtained from the file:

$$p_1 = \left(\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, a, \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}, b, \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}, c, \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, d, \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, e, \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \right)$$

$$p_2 = \left(\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, g, \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, h, \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}, b, \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}, c, \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, i, \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \right)$$

$$p_3 = \left(\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, j, \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}, l, \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \right)$$

$$p_4 = \left(\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, g, \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, h, \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}, b, \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}, i, \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, m, \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, d, \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, n, \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} \right)$$

If we compare these paths with those shown in section 2.7, we can see that using this method, 4 paths are obtained instead of the 3 paths presented in section 2.7. If we analyse the path p_2 from section 2.7, we can notice that the vector $[1\ 0\ 0]^T$ is repeated inside the path :

$$p_2 = \left(\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, g, \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, h, \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}, b, \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}, c, \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, i, \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, j, \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}, l, \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \right)$$

Since it is not possible to have a repeated vector in a path using the method proposed, this path p_2 is divided in two, resulting on the p_2 and p_3 of this section.

The change in the number of paths is reflected on the identified model. Choosing $k = 1$ and $k = 2$ for the modified paths and executing the identification algorithm, the models depicted in Figures 5.15 and 5.16 were obtained. Comparing the state transition diagram of these models with those shown in Figures 2.26 and 2.27, we can see the difference caused by the additional path in the identified model.

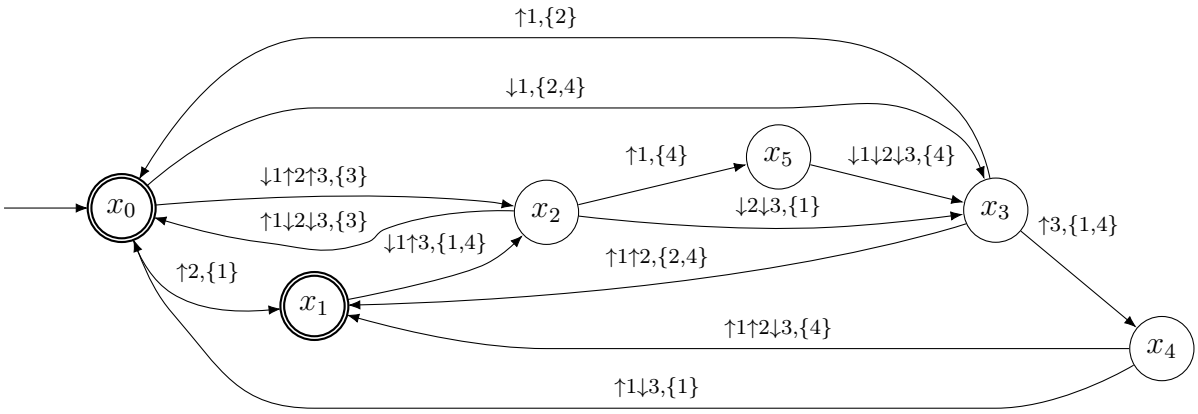


Figure 5.15: Identified model from paths extracted from `.csv` file using $k = 1$.

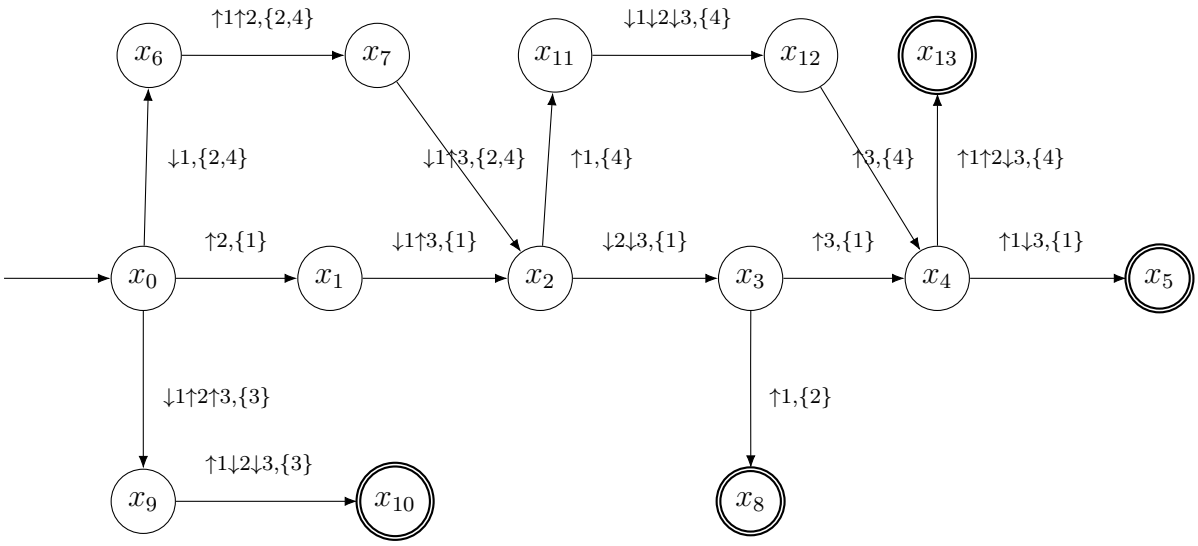


Figure 5.16: Identified model from paths extracted from `.csv` file using $k = 2$.

The path acquisition method presented in this work is used in the identification of the manufacturing system and its results are discussed in the next chapter. The tools created to implement the path acquisition method and the identification algorithm are presented in Appendix B.1.

Chapter 6

Identified Model

In this chapter the identified models generated by the algorithm 1 are discussed. The models are obtained through the execution of the identification algorithm using modified paths with different values of k.

6.1 Identified Model

In this work, as in other works KLEIN *et al.* (2005); MOREIRA and LESAGE (2018), we make the assumption 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 these sequences of events, the observation of the system must be made for a sufficiently long time. Thus, an experiment was made in order to observe the fault-free behaviour of the system. Normally, the system is observed until there is no considerable change in the observed language, but unfortunately, the experiment was interrupted by errors on the communication between the 2 PLCs, errors that dead-locked the system. The origin of these communication faults was not detected in this work and troubleshooting the communication can be proposed as a future work.

So, due to these errors, the experiment lasted for 2 hours, time in which it was possible to assemble and store over 100 cubes. A time-lapse of part of the experiment can be seen in `https://www.youtube.com/watch?v=ZtCCKJtA9pI`.

The acquisition of the *IOvectors* started once the system was initialised, that means, when the system was ready to begin the process. This corresponds to place p_{27} in Figure 4.1. The collected data[1] of this experiment has 19751 entries using 65 variables, the

[1] Available at: `https://raw.githubusercontent.com/Accacio/docsTCC/master/data/2019-05-10/2019-05-10_1524.csv`

inputs/outputs of the system and the auxiliary variables `ConcUP` and `ConcDWN`, presented in section 4.2.

Once this data was collected, the paths were obtained from the `.csv` using the method shown in section 5.2. The total number of paths obtained was equal to 2. This result can be easily explained by the behaviour of the system and the duration of the observation. The system is formed of different modules, and it presents a considerable concurrent behaviour. Since these modules are not necessarily synchronised, it is very unlikely that all the modules will return to their initial state at the same time. As presented in the path acquisition method, in order to generate a new path it is needed that the *IOvector* be the same as the initial vector. Thus, the concurrent behaviour of the system results in a few long paths. Using such systems with strong concurrent behaviour, an observation during 2 hours is not enough to observe the system in its completeness. If there were no communication errors between the PLCs the observation should have lasted for days or weeks, in order to observe better the system. However, as it was the longest duration possible of observation, the analysis of the system and identification was made using this dataset.

As expected, with a greater value of k, more states are identified. Figure 6.1 shows the variation of the number of states with respect to the value of k.

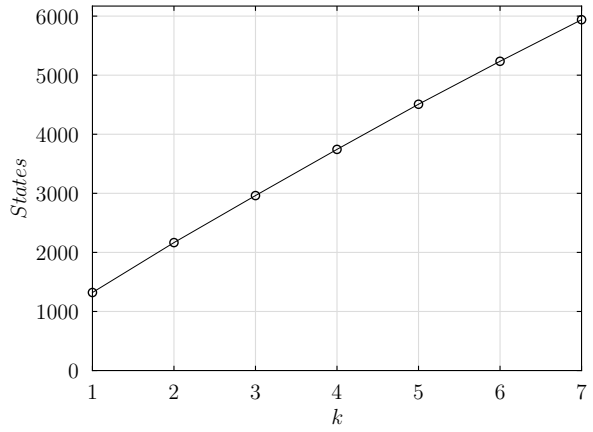


Figure 6.1: Number of states of identified model for different values of k.

The number of identified states for all values of k is greater than 1000. As state transition diagrams representing the identified model would be almost incomprehensible, a tool was developed to generate the f function of the identified model. This tool is presented in Appendix B.1.

The f function 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.

In order to analyse the exceeding language generated by the identified model using the DAOCT model, it was compared with another model, the Non-Deterministic Autonomous Automaton with Output (NDAAO) model, proposed by KLEIN *et al.* (2005). The NDAAO model is very similar to the DAOCT model, but the difference resides in the fact that NDAAO do not use path indices, what can increase considerably its exceeding language when there are multiple paths. This comparison shows that even for a considerable large system with more than 60 inputs/outputs, the DAOCT is more tailored for fault detection. The cardinality of the exceeding language of the DAOCT model is inferior to that of a NDAAO model of the same size (with a similar f function).

Figure 6.2 shows the comparison between both models using 2 values of k. In this case, considering $k = 1$ and the sequences of length smaller or equal to $n = 12$ the exceeding language of NDAAO is 1018 and for DAOCT it is 923. For $k = 2$, 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. As there are only a few paths, the difference between the exceeding language of both models is not very expressive.

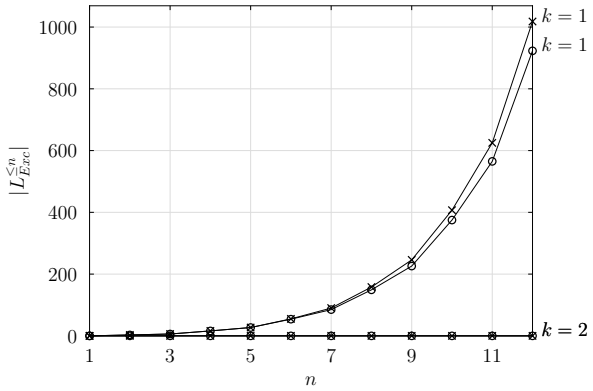


Figure 6.2: Comparison between the cardinality of the exceeding language generated by the DAOCT (o) and NDAAO (×) models, for $k = 1$ and $k = 2$.

Since using the path acquisition method on the original `.csv` file could obtain only 2 paths, an experiment was made in order to increase the number of paths and see how increasing the number of paths can reflect on the difference between the exceeding language of both models.

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, consequently the initial state of the new model.

A new `.csv` file was created from the original one. The file was created by discarding all vectors from the beginning of the file up to the first appearance of the new initial vector. Then, this new initial vector is used for the path acquisition method.

Instead of the original 19751 entries, the new file had 19427 entries, since, some entries were discarded from the original. The difference in number of entries was reflected in the number of generated states.

Using the path acquisition method in the modified file resulted in 80 paths, 40 times the number of paths of the original.

Figure 6.3 shows the variation of number of states with respect of the values of k.

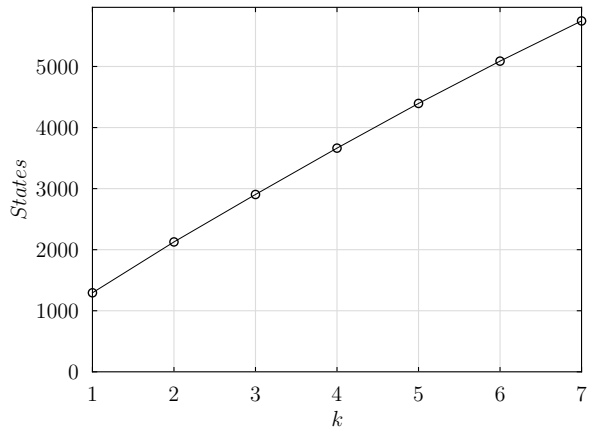


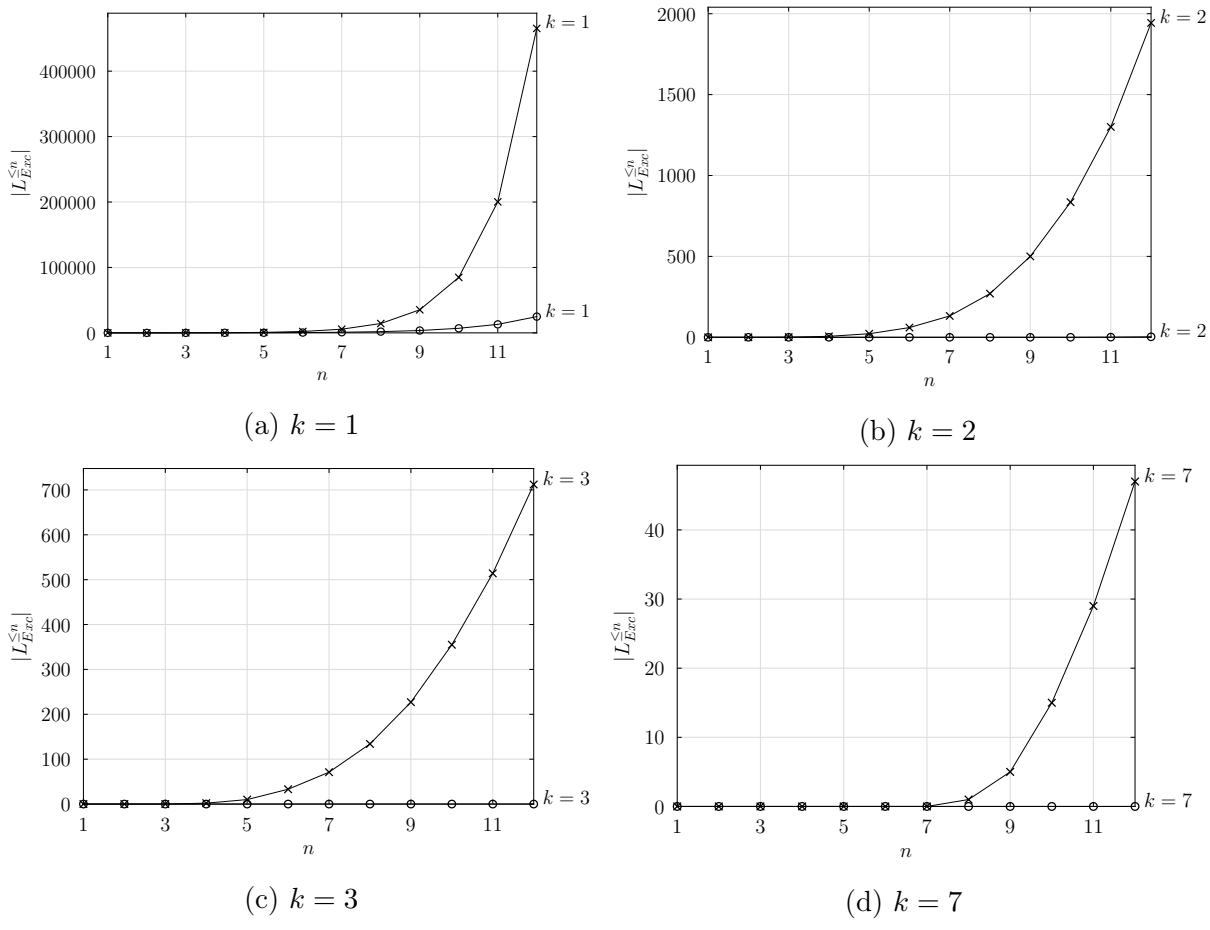
Figure 6.3: Number of states of identified model for different values of k.

Although both Figures 6.1 and 6.3 have the same order of magnitude for each k, the number of states diverges. Putting the values in a vector can be useful to 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]. The difference in the number of identified states can be caused by the difference in the number of entries in the `.csv` files.

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 models from Figure 6.2 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 for the DAOCT. With smaller and more numerous paths, we can see more clearly the difference between the exceeding language of the models. It is proved in MOREIRA and LESAGE (2018) that for acyclic paths the exceeding language of DAOCT is equal to 0, what can sustain the idea that DAOCT is better suited for fault-detection. For instance if we want to detect correctly the faults of the system for sequences of length equal to or smaller than 12, using the DAOCT model it is needed only to use a $k = 3$, that makes the paths acyclic and the exceeding language equal to 0 consequently. Meanwhile for the NDAAO, it is needed to use a k greater than 7, since for $k = 7$ and $n = 12$ the exceeding language of NDAAO is still equal to 47.



(a) $k = 1$

(b) $k = 2$

(c) $k = 3$

(d) $k = 7$

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

Although the modified `.csv` generates more paths and shows a more considerable difference in the exceeding language generated by both models, it does not mean that

the model identified from the modified `.csv` represents better the system than the model identified from the original `.csv`.

The choice of the initial vector affect directly the paths used as input for the identification algorithm, and consequently the identified model and how it represents the system. The effects on the modelling caused by the paths are discussed in the next section.

6.2 Discussion about Paths

As discussed in section 2.7, paths are used to model the system. Normally, these paths represent well the system behaviour. However, in the implementation phase, when the paths are not given but obtained from the observation of a sequence of *IOvectors* it is difficult to tell what paths represent well the system's behaviour. In this section, an example is presented in order to discuss how the choice of the initial vector for the path acquisition method presented in section 5.2 can modify the obtained paths.

Let us consider the following system as an example :

Example 6.1 (Conveyor Belt with 3 sensors)
This simple system consists of 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. The conveyor belt 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. 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 and the box falls from the conveyor belt, another box is placed over the belt restarting the cycle. Since only a box is placed over the belt, it is impossible for 2 sensors to be activated at the same time. As the belt is always turned on, this system only has outputs (inputs to the controller). The outputs of the system are the signals of the three sensors S_1, S_2 and S_3.

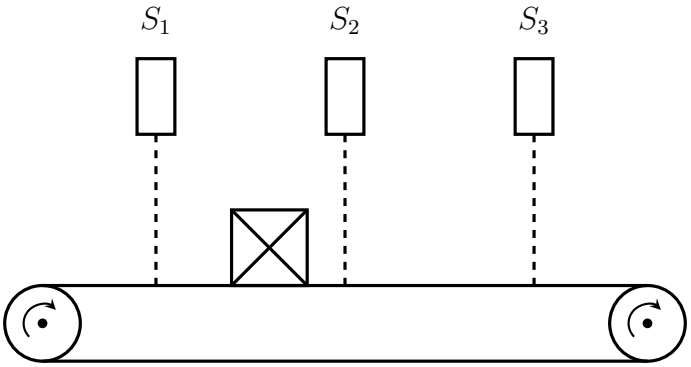


Figure 6.5: Scheme of the example 6.1.

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 *IOvectors*:

$$\begin{bmatrix}0\\0\\0\end{bmatrix}\begin{bmatrix}1\\0\\0\end{bmatrix}\begin{bmatrix}0\\0\\0\end{bmatrix}\begin{bmatrix}0\\1\\0\end{bmatrix}\begin{bmatrix}0\\0\\0\end{bmatrix}\begin{bmatrix}0\\0\\1\end{bmatrix} \tag{6.1}$$

This pattern will be repeated multiple times on the `.csv` file forming cycles, and since it forms cycles the pattern 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 of the analysis we will only discuss 2 ways of writing it, the first one shown in Equation 6.1 and the second shown in Equation 6.2. So, we can define two datasets of acquisition, one beginning with $[0\ 0\ 0]^T$ and other with $[1\ 0\ 0]^T$.

$$\begin{bmatrix}1\\0\\0\end{bmatrix}\begin{bmatrix}0\\0\\0\end{bmatrix}\begin{bmatrix}0\\1\\0\end{bmatrix}\begin{bmatrix}0\\0\\0\end{bmatrix}\begin{bmatrix}0\\0\\1\end{bmatrix}\begin{bmatrix}0\\0\\0\end{bmatrix} \tag{6.2}$$

If we take the first dataset, the one beginning with $[0\ 0\ 0]^T$, use the path acquisition method and then execute the identification algorithm, the identified model for $k = 1$ would be equal to the model depicted in Figure 6.6.

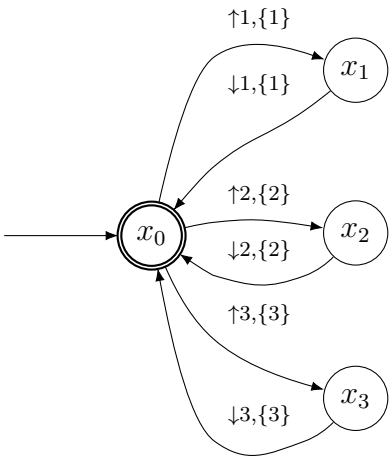


Figure 6.6: Identified model using $[0\ 0\ 0]^T$ as initial state, $k = 1$.

From the arcs of the state transition diagram, we can distinguish three paths. Since $[0\ 0\ 0]^T$ 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, the one beginning with $[1\ 0\ 0]^T$ the identified model for $k = 1$ can be seen in Figure 6.7.

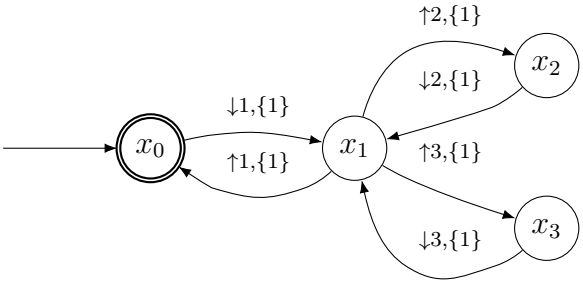


Figure 6.7: Identified model using $[1\ 0\ 0]^T$ as initial state, $k = 1$.

Differently, only one path is created this time. In this figure we can see the vector $[0\ 0\ 0]^T$ represented as the state x_1 in this state transition diagram. All other states have arcs coming from or going to it. Using a greater value of k, $k = 2$, for instance, we can have a better vision of this unique path, see Figure 6.8.

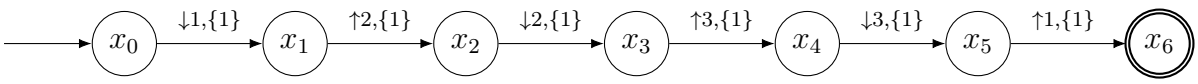


Figure 6.8: Identified model using $[1\ 0\ 0]^T$ as initial state, $k = 2$.

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 reliability of the identified model, but actually, it does not. If we consider 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 $\uparrow 3$ can only happen after the $\downarrow 1 \uparrow 2 \downarrow 2$ sequence, as described in example 6.1. It is important to notice that this representation is not perfect, but represents better the system than the model using $[0\ 0\ 0]^T$ as initial state.

From this example we can verify that the choice of the first vector plays a very important part on the path acquisition method and consequently on the identified model.

Remark 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 system ceases to be a black box, and it becomes a grey box.*

Remark 6.1 shows that the accuracy of the DAOCT model strongly depends on a good choice of paths. And as in the implementation the initial vector is used to determine the paths and the initial state of the system, this initial vector also plays a part in the identification phase. A question remains to be answered: how can we be sure if the initial state was well-chosen if we do not have any information about the system's original behaviour? Maybe the root of this problem resides on the fact that input/output vectors are used to extract the paths and to create the events. An approach that could fix the problem would be to use an identification model that uses the observation of the events, instead of observing the inputs and outputs of the system. And through the observation of the events the states and paths would be obtained, consequently, identifying the system.

Chapter 7

Conclusion

7.1 Concluding Remarks

In this work a method for the control, observation and identification of a DES was presented. First, the control logic was created using a CIPN, and then implemented in LD to be used in a Siemens PLC (chapter 4). After that, the observation of the inputs and outputs of the controller was made using data log function blocks that saved the data in `.csv` files, and finally, these `.csv` files were used as the input of 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 concurrent behaviour.

7.2 Further Work

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 instance (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, IL or ST. Since IL is less used, ST would be a better 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, increasing the maintainability of the code.

Another issue was about the observation. Although the acquisition of inputs/outputs using data logs 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.

As shown in chapter 6, the didactic manufacturing system used for the experiments have a considerable concurrent behaviour, affecting the identified model, on the number of states and extracted paths. As future work would be 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 behaviour is previously known, by a simple description of the function of the system and another considering a black box approach.

Another proposition for a future work is made in chapter 6. Instead of using an identification model based on the observation of inputs/outputs of the system, an alternative would be to create and use a model that uses the observation of the events of the system.

Bibliography

CABRAL, F. G., MOREIRA, M. V. "Synchronous Codiagnosability of Modular Discrete-Event Systems", *IFAC-PapersOnLine*, v. 50, n. 1, pp. 6831–6836, 2017.

CARVALHO, L. K., MOREIRA, M. V., BASILIO, J. C. "Diagnosability of intermittent sensor faults in discrete event systems", *Automatica*, v. 79, pp. 315 – 325, 2017. ISSN: 0005-1098. doi: https://doi.org/10.1016/j.automatica.2017.01.017. Available at: <http://www.sciencedirect.com/science/article/pii/S0005109817300274>.

CASSANDRAS, C. G., LAFORTUNE, S. *Introduction to discrete event systems*. Springer Science & Business Media, 2009.

DAVID, R., ALLA, H. L. *Du Grafcet 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.

FLORIANO, L. A. *Sincronização de Sistemas a Eventos Discretos Modelados por Redes de Petri Usando Lugares Comuns*. Undergraduate Project, Universidade Federal do Rio de Janeiro, 2019.

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.

PEREIRA, A. P. R. *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.

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.

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.

SAMPATH, M., SENGUPTA, R., LAFORTUNE, S., et al. "Diagnosability of discrete-event systems", *IEEE Transactions on Automatic Control*, v. 40, n. 9, pp. 1555–1575, Sep. 1995. ISSN: 0018-9286. doi: 10.1109/9.412626.

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>.

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>.

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.

VIANA, G. S., BASILIO, J. C. "Codiagnosability of discrete event systems revisited: A new necessary and sufficient condition and its applications", *Automatica*, v. 101, pp. 354 – 364, 2019. ISSN: 0005-1098. doi: https://doi.org/10.1016/j.automatica.2018.12.013. Available at: <http://www.sciencedirect.com/science/article/pii/S0005109818306198>.

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} p_{34}, p_{42}, p_{57}, p_{69}, p_{110}, p_{117}, p_{129}, p_{138}, p_{141}, p_{146}	Turn Conveyor Belt On
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
p_{77}, p_{80}, p_{97}, p_{100}	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_{82}, p_{102}	Extend Arm
p_{83}, p_{95}, p_{103}, p_{142}	Raise and Extend Arm
p_{85}	Raise Arm and Rotate Arm CCW
p_{86}	Raise Arm and HALFPIECE-COUNTER:=HALFPIECECOUNTER+1
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_{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_{104}	Turn Arm CCW
p_{106}	Turn Arm CW
p_{108}	Cube on Storage Unit
p_{111}, p_{112}, p_{113}, p_{114}	Move Storage Unit Upwards
p_{115}	COUNTER3:=COUNTER3+1
p_{116}	RESET COUNTER3*
p_{118}	COUNTER1:=COUNTER1+1 e COUNTER4:=COUNTER4+1
p_{119}, p_{120}, p_{121}, p_{122}, p_{123}, p_{124}, p_{125}	Move Storage Unit to the Left
p_{126}	COUNTER5:=COUNTER5+1
p_{127}	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 <= BELT_ANGLE_CW
t_{19}	Start Button
t_{20}	$\overline{\text{MAG1 Empty}}$
t_{22}	↑ MAG1's Cylinder Extended
t_{23}	↑ 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}	↑ Proximity Sensor Left Discharge Cylinder
t_{29}, t_{59}	Right Discharge Cylinder Extended
t_{31}	White Color Sensor
t_{32}, t_{63}	↑ 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}	↑ Proximity Sensor End Of Conveyor Belt
t_{44}, t_{70}	↓ Proximity Sensor End Of Conveyor Belt
t_{46}	$\overline{\text{MAG2 Empty}}$
t_{48}	↑ MAG2's Cylinder Extended
t_{49}	↑ MAG2's Cylinder Retracted
t_{58}	↑ Proximity Sensor Right Discharge Cylinder
t_{72}	Arm Raised
t_{77}	ARMCOUNTER <= PRESS_ANGLE
t_{82}	HALFPIECECOUNTER=1, Assembly Unit Holder Extended and Safety Door Opened
t_{84}	ARMCOUNTER >= BELT_ANGLE_CCW
t_{95}, t_{101}	Arm Raised, Storage Unit Right and Inferior Limit Switches
t_{96}	Storage Unit Arm Alignement Encoder
t_{97}	ARMCOUNTER <= 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}	$\overline{\text{Vertical Encoder}}$
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

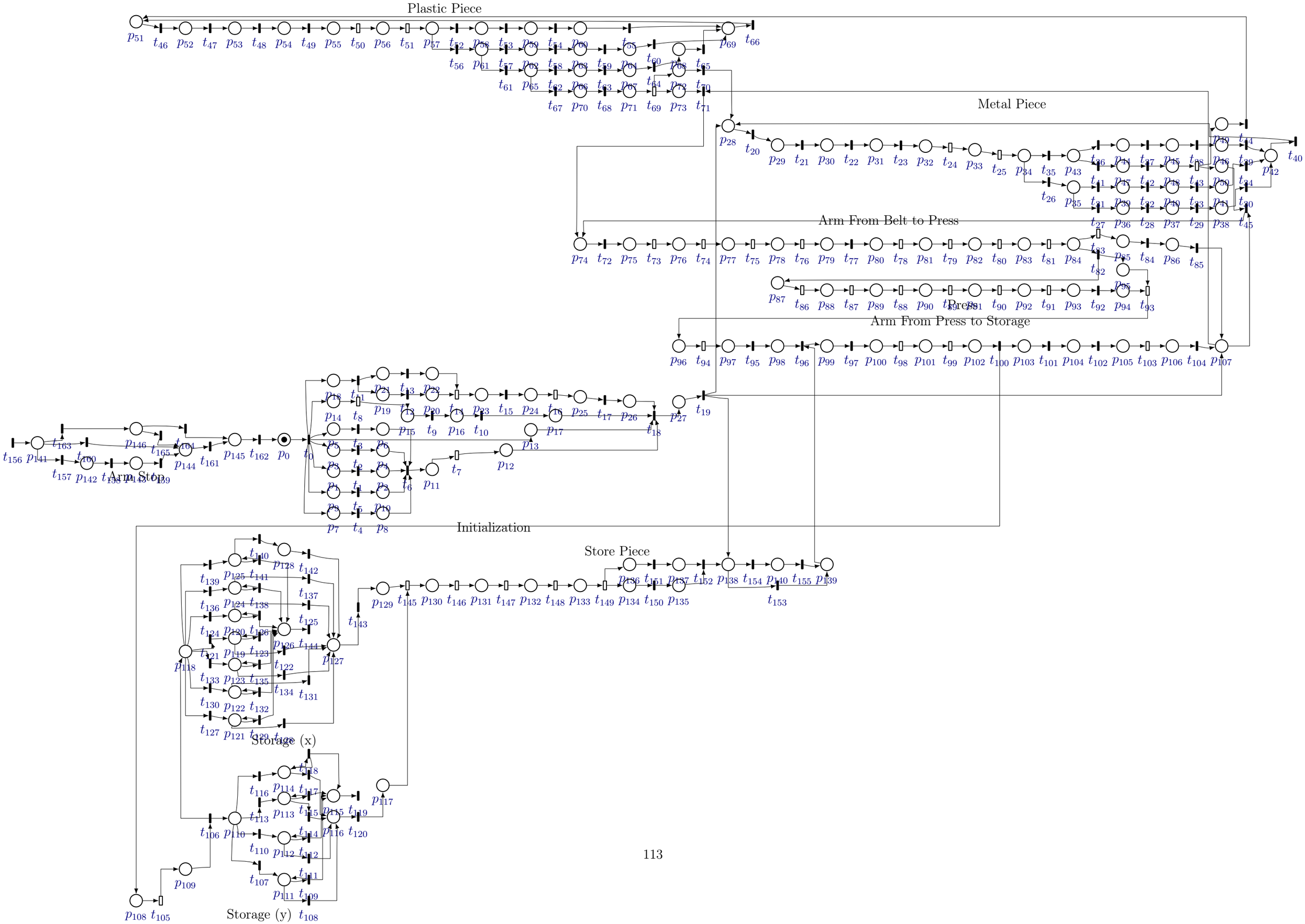
Continued from previous page

Transitions	Meaning
t_{140}	COUNTER5=7 and Horizontal Encoder
t_{141}	COUNTER5<=7 and Horizontal Encoder
t_{144}	$\overline{\text{Horizontal Encoder}}$
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
t_{160}	(ARMCOUNTER >= STORAGE_ANGLE_BEFORE and ARMCOUNTER < PRESS_ANGLE_AFTER) or 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}	$\overline{\text{Arm Retracted}}$
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


Plastic Piece
Metal Piece
Arm From Belt to Press
Press
Arm From Press to Storage
Arm Stop
Initialization
Store Piece
Storage (x)
Storage (y)

Appendix B

Tools

The development of these tools used in this work 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. The tools developed for this work are available at `https://github.com/Accacio/docsTCC/tree/master/tools`, and the most used ones will be presented in the following sections.

B.1 daoct

To implement the algorithm 1, presented in MOREIRA and LESAGE (2018), a script was created by Ryan Pitanga as part of his undergraduate project, PITANGA CLETO DE SOUZA (2019). His code was partially reimplemented, so it could be used as a command line tool based in common Unix tools (that uses stdin and stdout[1] to pipe[2] processes). Some extra features were added, in order to represent the identified model in two forms: as a graph or as a list. The graph is represented using the dot language[3] and the list represents the f function of the identified automaton. The output in `.dot` file format can be input in another script to draw the state transition diagram of the identified models.

Figure B.1 depicts the help menu of the daoct program. An example of the most common usage of the daoct program can be presented. Considering the `.csv` file presented in Figure B.2, it can be input to the daoct program in order to obtain the paths from the file and identify the DAOCT model. The two outputs can be obtained by using the fol-

[1] `http://man7.org/linux/man-pages/man3/stdin.3.html`
[2] `http://man7.org/linux/man-pages/man2/pipe.2.html`
[3] language used by the program graphviz (`https://graphviz.org/`) to draw graphs

lowing commands: `daoct -i filename.csv -g` and `daoct -i filename.csv -f`. The first one generates the graph using dot language (shown in Figure B.3) and the second one results in the f (shown in Figure B.4).

```
[ 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
  -r , --report           output automaton Full Report
  -e , --exceeding        output graphic of the exceeding language
  -o , --output           chooses graphic output folder
  -m                      use multiple k to plot graphic

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

Figure B.1: daoct help dialog.

```
[ 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/ tools/daoct ]
```

Figure B.2: daoct csv input file.

```
[ accacio/ tools/daoct ] daoct -i sampleData.csv -g
digraph a {
rankdir=LR;
# splines = ortho
ratio=fill
graph [pad="0.01", nodesep="0.1", ranksep="0.01"];
node [shape=circle];
margin=0;
{rank =same;}
# size="11.7,8.3!";
init [style=invis]
init -> x0
x4 [shape=doublecircle];

x0 -> x1 [texlbl="\scriptsize ⬇Output1,\{1\}"]
x1 -> x2 [texlbl="\scriptsize ⬆Input1,\{1\}"]
x2 -> x3 [texlbl="\scriptsize ⬇Input1,\{1\}"]
x3 -> x4 [texlbl="\scriptsize ⬆Output1,\{1\}"]

}
[ accacio/ tools/daoct ]
```

Figure B.3: daoct graphviz output.

```
[ accacio/ tools/daoct ] daoct -i sampleData.csv -f
f(x0,⬇Output1) = x1 {1} \\
f(x1,⬆Input1) = x2 {1} \\
f(x2,⬇Input1) = x3 {1} \\
f(x3,⬆Output1) = x4 {1} \\
[ accacio/ tools/daoct ]
```

Figure B.4: daoct f function output.

B.2 dot2automata

In order to visualize the output of the daoct program, the script `dot2automata` was created. This program is basically a wrapper of the `dot2tex` program, that is capable of transforming a `.dot` file into a `.tex` file using tikz syntax. The program `dot2automata` pre-process the `.dot` file so the tikz output can be drawn using an automaton style with states, marked states and labelled arcs, similar to the style presented in MOREIRA and LESAGE (2018). Figure B.5 shows the help menu of the `dot2automata` program, describing how to operate it.

```
[ 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.

The output of the program `daoct` can be piped to the `dot2automata` program in the following manner: `daoct -i filename.csv -g | dot2automata - -o outputFilename`

This program outputs a `.tikz` file, but it can also output a `.pdf` file. The `.pdf` file is used as a preview of the image that is generated by the tikz figure. The tikz figure can be included in a LaTeX document and resized using the tikzscale package. So, including a `.tikz` file in a LaTeX document, as in Listing B.1, it can result in the diagram depicted in 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}
```

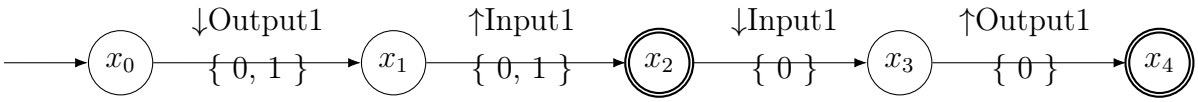


Figure B.6: dot2automata output.

B.3 dot2petri

The `dot2petri` program is a similar to `dot2automata`.The working principle is the same but the objective is different. For `dot2petri` program, the objective is to visualize Petri nets. Its help dialogue is presented in Figure B.7:

```
[ 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 and vice-versa). An inhibitor arc can be created changing the style of the arc. A label can be used to represent the *Pre* and *Post* functions of the Petri Net. A tikz style for external places and transitions is created in order to be represented by dotted lines. External places and transitions use the same tags of normal places and transitions, but with a letter 'e' prepended.

Such dot files can be created in two ways: manually writing them or using another program called `petriml2dot` present in the same repository. The `petriml2dot` program converts a file in *petriml* format, created using the Platform Independent Petri net Editor 2 (PIPE2)[4] into 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 in its current version, it lacks a tool to export the graph as a `.tikz` file. So, `petriml2dot` and `dot2petri` are used to fill the gap.

The code shown in Listing B.2 used as input for the `dot2petri` script outputs a `.tikz` file. Including the `.tikz` in a similar fashion to the one shown in Listing B.1, can result in the following figure:

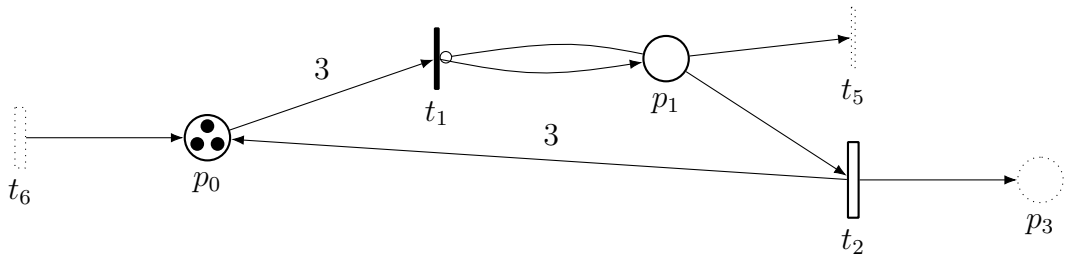


Figure B.8: dot2petri output.

[4] http://pipe2.sourceforge.net/