An Experimental Testbed and Methodology for Security Analysis of SCADA Systems

Bernardo Lamas, Ayoub Soury, Bilel Saadallah, Abdelkader Lahmadi, Olivier Festor

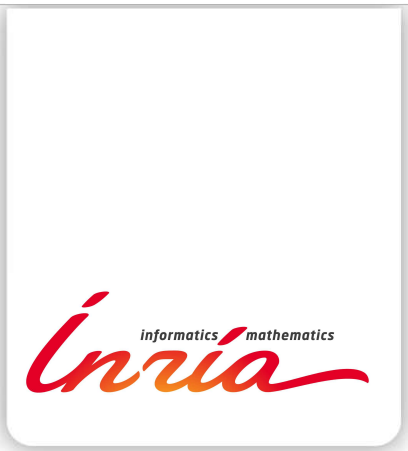
TECHNICAL REPORT

N° 443

December 2013

Project-Team MADYNES

ISSN 0249-0803 ISRN INRIA/RT--443--FR+ENG



An Experimental Testbed and Methodology for Security Analysis of SCADA Systems

Bernardo Lamas, Ayoub Soury, Bilel Saadallah,
Abdelkader Lahmadi, Olivier Festor

Project-Team MADYNES

Technical Report n° 443 — December 2013 — z pages

Abstract: In this report, we detail the development of an experimental testbed dedicated to the security analysis of process control networks (SCADA) employed in industrial systems. The testbed is built on real hardware controllers and simulated physical processes which makes it suitable for laboratory environments. Its process-level modularity makes it easy to configure and develop multiple networked control scenarios. We designed and implemented a variety of networked process control systems using the PROFINET protocol at their communication layer. Using the developed experimental processes, we elaborated a methodology to infer a discrete model of the running controlled system through network traffic observation and process mining techniques. Secondly, we carried data manipulations on PROFINET messages to identify their impact on the controlled process behaviour. We evaluated our approach by comparing the inferred models with the designed baseline models.

Key-words: Process control networks, Profinet, model inference, process mining

RESEARCH CENTRE
NANCY – GRAND EST

615 rue du Jardin Botanique
CS20101
54603 Villers-lès-Nancy Cedex

Développement d'un banc d'essai expérimental et d'une méthodologie pour l'analyse de la sécurité des systèmes SCADA

Résumé : Dans ce rapport, nous détaillons la mise en oeuvre d'une plate-forme expérimentale pour analyser le protocole PROFINET employé dans les réseaux de contrôle industriels (SCADA). La plate-forme s'appuie sur des équipements de contrôle matériels (automates et blocs d'E/S) et des outils logiciels (Matlab, Simulink et xPCtarget) pour le développement des différents scénarios de systèmes contrôlés. En utilisant cette plate-forme, nous avons élaboré une méthodologie de reconstitution d'un modèle discret d'un processus contrôlé en s'appuyant sur l'observation du trafic réseau et les techniques de fouille des processus. Ensuite, nous avons développé des scénarios d'attaques en manipulant les données protocolaires afin d'identifier leur impact sur les systèmes contrôlés. Nous avons évalué notre approche en comparant les modèles inférés avec les modèles de base conçus manuellement.

Mots-clés : Réseaux de contrôle industriels, Protocole Profinet, Inférence de modèles

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Modern industrial automated systems consist of multiple devices exchanging information trough communication networks. Traditionally these local networks are confined to an industrial plant, but recently there is a tendency to interconnect them remotely trough Internet to collect available supervisory and control data to apply maintenance and diagnose operations on the running devices. This interconnection brings security problems to industrial networks, as they are becoming exposed to traditional attacks (flooding, denial of service, etc.) and intrusions through external communication links to the Internet. In addition, it brings novel attacks where process control components and algorithms are attacked, aiming to perturb the process dynamics.

1.1 Background

In [2] authors explain that Industrial Control System (ICS) is a general term that refers to a set of interconnected systems that include Programmable Logic Controllers (PLC), Distributed Control Systems (DCS) and Supervisory Control and Data Acquisition (SCADA) as described in Figure 1.1. They allow automation and control of industrial processes.They include industrial processes for Power Generation, Gas Transportation, Aero-Space Industry, Food Industry, Automotive, among others.

A PLC (Programmable Logic Controller) is an electronic device widely used in industrial automation. As the name suggests, it was designed to be programmed and control sequential processes in real-time. It is essentially a digital computer that can store and run programs. The major difference it has with other computers is that a PLC is able to work in severe environment conditions, with varying temperatures, dust, dirt, etc. It also allows the connection of modules such as digital inputs and outputs, analogue or special modules.

SCADA is a computer-based system that supports monitoring and controlling process variables at long distance, providing communication with the field devices (standalone controllers) and controlling the process automatically by specialized software. It also sends all the information generated in the production process to different users in the same level or to users with supervising capabilities. The exchanged information can cover supervision tasks, quality control, production control, storage of data, etc.

A DCS (Distributed Control System) is a system of control in which the controller elements are not centralized in location but distributed throughout the system. The controllers (PLC) are connected through communication and monitoring networks.

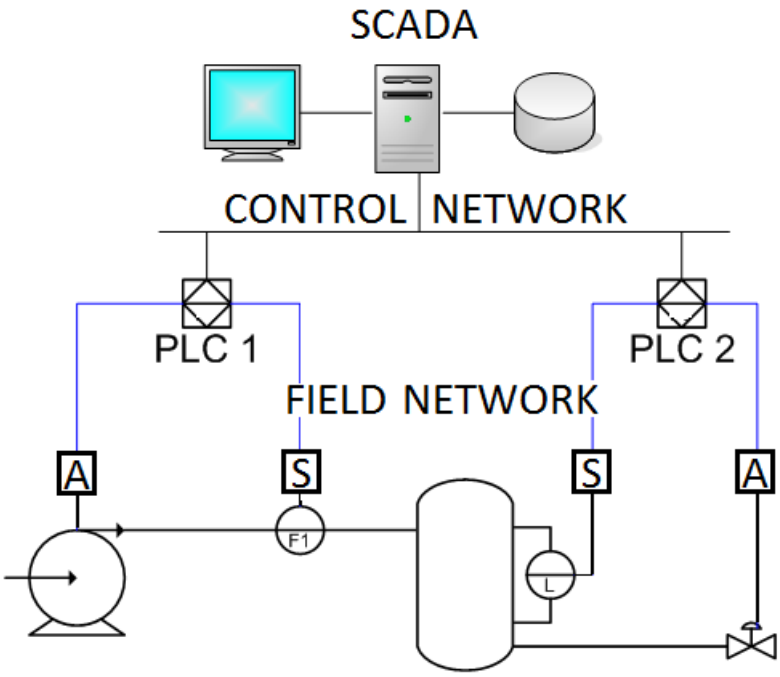


Figure 1.1: An example of a SCADA control network.

1.2 The security challenge

The security of a process control system can be defined as the amount of information about the system dynamics and model parameters contained in local measurements made by an adversary [3]. It differs from traditional IT security since the ultimate goal of the attacker targeting a process control network is often to perturb the physical system dynamics and structure rather than stealing or modifying available data.

While several previous works have proposed security mechanisms comparable to those used to protect traditional Information Technology (IT) systems, such as using firewalls or packets filters, few have explored new and fundamentally different paradigms for securing industrial control systems. In this work, we define a model of security enforcement, focusing on the goal of the attack and not on the mechanisms of how particular IT vulnerabilities are exploited, and how the attack is hidden. It will be possible to provide a security level that allows the automatic response to avoid that the system will be taken into an insecure state. Our proposed approach is close to the work presented in [4] with the exception that their work focuses on systems and controls in the continuous time domain, while our covers a portion of the Hybrid systems, i.e. continuous systems with discrete-time control. Our proposal is based mainly on the incorporation of knowledge about the running controlled physical system using identification techniques and its modelling using Petri Nets. Through this identification mechanisms, it will be possible to detect several attacks that modify in any way the behaviour of the system.

In this report we present a testbed and a method to monitor and collect network traffic traces of a networked process control system relying on the PROFINET protocol in their communication layer. This type of systems relies on a continuous process with digital inputs and outputs, controlled through a dedicated controller such as a PLC (Programmable Logic Controller). From

the collected traces, we inferred a discrete model of the running process to characterize the attributes of both the system and the controller. We developed then several attack scenarios against two designed process control systems. The attack scenarios consist in discarding and injecting data packets into the network that connects the PLC and the I/O module. Arbitrary data modifications have been used on sensors, actuators of the system to observe the consequences caused by these changes in the inferred control model.

1.3 Report organization

The rest of the report is organised as follows. Chapter 2 describes the technical environment of the testbed for physical processes simulation and their control through PLCs with enabled PROFINET protocol. Chapter 3 describes the designed process control systems. We detail for each process its design model and its control part. Chapter 4 describes the reverse engineering method that we have used to infer a control model from observed PROFINET messages between a PLC and the I/O blocks. Chapter 5 details the obtained inferred models for two designed systems : a sliding door system and a mixing paint system. We also describe several attack scenarios that we made to identify their impact on the inferred reference models.

Chapter 2

Testbed description

In this chapter, we detail the different components of the testbed. It contains a combination of interconnected hardware and software simulating in real time a physical system and applying a control program running on a PLC. The testbed allows us to capture and inject traffic that will help us to infer the model of a running controlled process.

2.1 Hardware

This section describes all the hardware used in the testbed. We explain their conditioning, wiring and configuration. We also present some technical characteristics to reproduce experiments with other control hardware or define different physical processes.

2.1.1 Host machines

In the testbed, we used 3 hosts with the following characteristics:

Host 1 Used for real time simulation.

- PC Model: Dell Precision Workstation 390
- Operating System: LiveCD Matlab
- Memory: 8000MB Ram
- Processor: X6800 a 2.930Ghz (2 CPU)

Host 2 It is used for the design of physical processes using Matlab.

- PC Model: HP Z400 Workstation
- Operating System: Windows 7 Professional 64-bit
- Memory: 12288MB Ram
- Processor: W3670 a 3.20Ghz (6 CPU)

Host 3 It is used as the programming station for the PLC, IO block and SCADA interfaces.

- PC Model: Dell Precision Workstation 390
- Operating System: Windows XP Professional
- Memory: 3568MB Ram
- Processor: X6800 a 2.930Ghz (2 CPU)

2.1.2 Siemens PLC

The control of the system will be performed by a Simatic S7-300 PLC. The device is equipped with a 315F-2PN/DP CPU and a communication module that supports the standard Ethernet. Through this port we connect the PLC with the Digital I/O module and the HMI station. We used the TIA Portal (Totally Integrated Siemens Automation Portal) to program the PLC using the Ladder logic language.

Figure 2.1: PLC Simatic S7-300

Ladder is a graphical language which represents a program in a diagram based on circuits with contacts and relays. The name is based on the observation that programs written in this language resemble ladders whose steps are logical functions that run in parallel.

2.1.3 Digital I/O block

Inputs and outputs are mounted in a module called **ET200S** which is a decentralized peripheral module. It has a communication part that provides a network connectivity using either PROFIBUS or PROFINET. It embeds a SD card reader to save the configurations and several general purpose slots in the module. In our case, on the ET200S (Figure 2.2) are mounted four digital ports with labels in yellow and green, allowing to know the nomenclature used internally in the TIA PORTAL, for their programming:

- 2 ports of 8 digital outputs, 24 V DC/0,5 A
- 2 ports of 8 digital inputs, 24 V DC Standard.

The figure shows several blue and red wires for power use, white for the digital inputs and blue for the outputs. Together they connect all the sensors and actuators of the system. Table 2.1 shows the mapping of the input and output addresses between the physical connections and labels used for programming in LADDER.

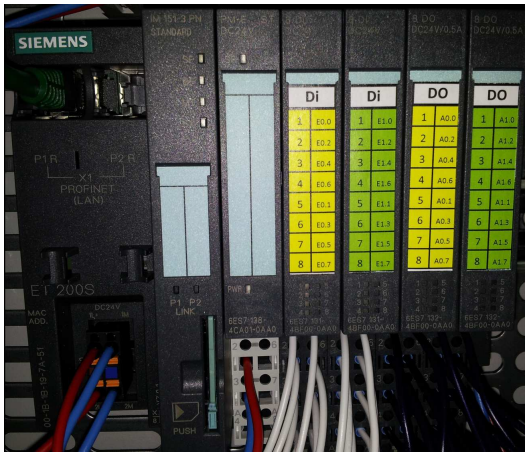


Figure 2.2: Digital module ET200S Siemens

Table 2.1: Addresses mapping of the digital I/O block.

Physical I/O Order	Ladder I/O Order			
Nro/DIO	DI1	DI2	DO1	DO2
1	I0.0	I1.0	Q0.0	Q1.0
2	I0.2	I1.2	Q0.2	Q1.2
3	I0.4	I1.4	Q0.4	Q1.4
4	I0.6	I1.6	Q0.6	Q1.6
5	I0.1	I1.1	Q0.1	Q1.1
6	I0.3	I1.3	Q0.3	Q1.3
7	I0.5	I1.5	Q0.5	Q1.5
8	I0.7	I1.7	Q0.7	Q1.7

2.1.4 HMI

We created a human-machine interface (HMI) to visualize the behaviour of the controlled system. This device is responsible for presenting data, allowing the operator to enter values such as operation modes, setting points, alarms, etc. The used HMI is based on the SIMATIC TP700 Comfort series device illustrated in Figure 2.3. This computer is provided by Siemens and has Windows CE as operating system. It is connected to the control system through a network wire and uses the PROFINET protocol.

2.1.5 Contec PIO interface

All sent and received data from the I/O block ET200S to the Target PC goes through a digital adapter connected to a PCI Slot of the machine. The card is manufactured by the company CONTEC and the model is PIO-32/32RL (PCI).

The adapter operates at input as a high impedance opto-coupled port to prevent electrical over-voltage damage and can provide at output 32 current source opto-isolated digital signals 12 to 24 VDC and a maximum electric current of 100 mA per channel. The Contec kit has another

Figure 2.3: Simatic TP700 Comfort

card that is located inside the testbed. It connects the digital inputs and outputs to the ET200S module. Table 2.2 shows some technical specifications regarding the 32/32RL PIO Card.

Table 2.2: Technical specifications of PIO 32/32RL

Type	Item	Specification
Input	Input type	Opto-Isolated / Positive Logic *1
	Number of Channels	32
	Input Resistence	4.7kΩ
	Input Current in ON	2.0mA
	input Current in OFF	0.16mA
	Time Response	200μS
Output	Output Type	Opto-Isolated / Positive Logic *1
	Number of Channels	32
	Output Voltage	35 VDC (max)
	output Current	100mA (max)
	Time Response	200μS

2.1.6 Ethernet switch

The interconnection of PLC, SCADA, I/O block, configuration PC and data capture PC is done via a Cisco 16 ports switch (SG 300-20) as shown in Figure 2.4. All wires are UTP Category 5e.

The switch offers the possibility of port mirroring. Port mirroring is used to send a copy of network packets seen on one switch port, multiple switch ports, or an entire VLAN to a network monitoring connection on another switch port. This is commonly used for network appliances

that require monitoring of network traffic. Up to eight sources can be mirrored which allows to set any combination of eight individual ports and/or VLANs. But only one instance of mirroring is supported system-wide; the target port is the same for all the mirrored VLANs or mirrored ports.

The switch in our possession was particularly configured so that the port number 10, which is connected to the PC performing packet capture, works as a target port for all captured mirrored packets. The configuration of the switch is done using the web-based configuration utility. Below are the different switch configuration steps:

1. Open a web browser and enter the IP address of the switch to be configured in the address bar. If the switch is subject to its first configuration, use the factory default IP address 192.168.1.254. To log in to the configuration utility, enter the username/password. The default username is cisco and the default password is cisco. If you log in for the first time using the default username and password, you are required to enter a new password.
2. Click Administration > Diagnostics > Port and VLAN Mirroring. The Port and VLAN Mirroring Page opens.
3. Click Add to add a port or VLAN to be mirrored. The Add Port/VLAN Mirroring Page opens.
4. Enter the parameters:
   - Destination Port: Select the target port to where packets are copied. Here we assigned port 10 in our configuration.
   - Source Interface: Select Port or VLAN as the source port or source VLAN from where traffic is to be mirrored. We selected all used ports for mirroring.
   - Type: Select whether incoming, outgoing, or both types of traffic are mirrored to the target port. If Port is selected, the options are Rx Only for port mirroring on incoming packets, Tx Only for port mirroring on outgoing packets and Tx and Rx for port mirroring on both incoming and outgoing packets. We chose the last type for all mirrored ports.
5. Click Apply. Port mirroring is added, and the switch is updated.

Figure 2.4: Cisco SG300-20 Switch

2.2 Real Time simulation of a physical process

2.2.1 Matlab, Simulink and xPCTarget

We used Matlab[1] a numerical computing environment and programming language to perform the Real-Time Simulation of the physical process. It allows matrix manipulations, plotting of

[1] http://www.matlab.com

functions and data, implementation of algorithms, etc... This Software provides a collection of tools, called Toolboxes that simplifies the necessary work to simulate almost any kind of systems.

The first step was to install Matlab on a PC running Windows of 32 or 64 Bits. It was decided to use the 2012b version in order to take the advantages of the last features. The Toolbox that allows Real-Time simulation is xPC Target[2]. It is a real-time software environment from MathWorks. Together with a x86-based real-time system, it is able to simulate and test Simulink and Stateflow models in an early-as-possible stage in real-time on the physical hardware under test. It uses a target PC (another machine, can be a PC, without Operating System) connected using Ethernet to the Matlab Machine.

The installation process also adds the necessary Toolboxes (xPC Target, Simulink, Simscape, etc). We also installed the C compiler with Microsoft Visual Studio 2010.
The complete list of required software is as follows:

- 32-bit or 64-bit Windows operating system
- MATLAB Version 7.14
- Simulink Version 7.9
- Simulink Coder Version 8.2
- MATLAB Coder 2.2
- Microsoft .NET Framework 4.0
- C Language compiler
- xPC Target Version 5.2

Matlab Configuration : At Matlab prompt the `xpsetCC('setup')` command was executed. It allows the selection of the C compiler used by Matlab. After that, the command `xpcexplr` was run to open the Target PC Explorer. Options for the target PC can be chosen, i.e IP Address, Boot method, and others. In our case, the CD Boot method was chosen. After putting on the target computer it has to appear a blue screen as shown in Figure 2.5.

To verify that Matlab recognises the PCI card installed into the Target PC, the command `getxpcpci` can be executed. The output should be a list of installed devices. In our case the CONTEC PIO-32/32RL(PCI)H. **Unfortunately, this card was not supported by Matlab, so we have adapted the driver to make it working with the card recognized by Simulink.**

2.2.2 Contec PIO driver modifications

As mentioned before, the PCI card was not supported by the xPC Target toolbox, but another similar card the PIO-32/32L(PCI)H from the same manufacture was supported. The difference between them is the electrical outputs. We modified the available driver to support the new card using the guide provided by Matlab about how to create your own drivers[3].

- First, we prepared a **.mdl** file with Simulink in which the supported block PIO-32/32L(PCI)H was used.

[2]http://www.mathworks.fr/products/xpctarget/
[3]http://www.mathworks.fr/help/toolbox/xpc/driver/f1-5937.html

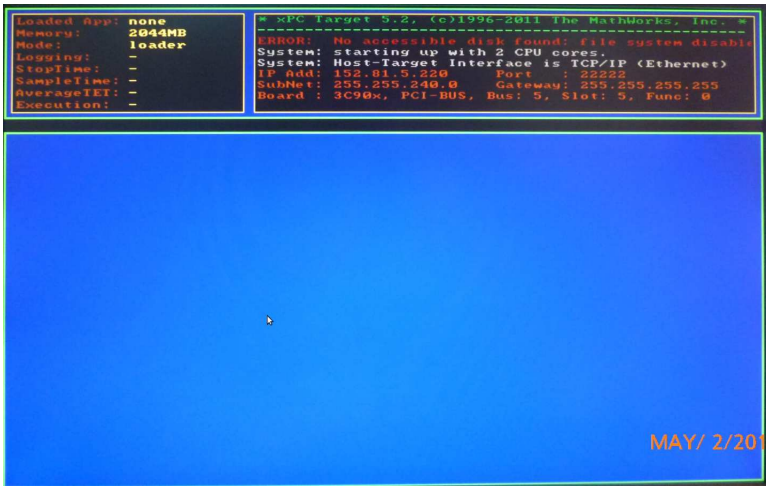


Figure 2.5: Target PC

- After that, it was compiled and uploaded to the Target PC. An error message appeared, saying that the card with address `0x9152` was not found.
- Matlab was trying to find a card with that address, so a search in all driver files was done to find out which one contains that address. Only 3 files appear `diocontecpio64pcih.c`, `xpcconteclib.mdl` and `getsuppcidev.m`.
- With the command `getxpcpci('all')`, we obtained the address of the PIO-32/32RL(PCI)H card and files were modified with the new address.
- The driver is rebuilt using the `mex` command as follows `mex diocontecpio64lpcih.c`.
- The boot CD-ROM must be prepared again in order to add the new driver. **The driver is added automatically**.
- When the boot process ended, it was possible to verify that the card was detected as the PIO-32/32L(PCI)H.

2.2.3 Installation and configuration of the Siemens TIA Portal

The TIA Portal from Siemens is a new engineering framework that gives to the user a complete control of the configuration, diagnose and maintenance of the automation system in only one screen. It is a collection of tools that allow the configuration of PLCs, I/O Blocks, HMIs and lots of different devices related to automation applications. The installation of the software requires Microsoft Windows OS and some special licenses that allow the user to execute the program modules.

In our case, we used 4 different modules or features from the framework. We first created a new project containing all the available devices. The software provides a list of all supported hardware, so the user only needs to pick them from there. Next, we configured all the equipments by providing to each device an IP address and a name. Finally, we created a PROFINET network and all the devices were connected to it.

One of the modules available in TIA portal allows the programming of PLCs in LADDER logic language. So the next step was to design and write the control program of the system. We

chose LADDER because it is very simple and easy to understand and debug. The debug work is highly simplified by the use of real time visualization of the PLC inputs and outputs. This feature is very useful if you want to see the behaviour of the program continuously.

The SCADA part was also programmed entirely from the Portal. The screens were designed and uploaded to the HMI device. From there, the operator can observe and introduce variables values in the system.

Chapter 3

Experimental design

In this chapter, we detail the design of two control processes to be used on our testbed as experimental systems.

3.1 Overview

First, we designed a simple system which is a sliding door with two digital inputs and a single output. The two inputs represent two sensors located at the edges of the door which are set at 1 when the door reaches each end. The output variable is associated to a motor where the 0 value denotes the closing operation and 1 denotes the opening operation. The control has two modes, one manual and one automatic. When manual mode is selected, the desired position of the door can be set by a switch on the digital display of the HMI. In automatic mode, the reference is modified automatically according to the position of the door. The idea is that the door can keep opening and closing continuously for an indefinite period of time or until the operator changes the mode to manual. This system is very simple and we have already made the associated manually-designed Petri net. Thus, the obtained results from this experimental system will allow us to adjust all the necessary parameters required for the proper functioning of the testbed.

In a second step, we designed a mixing paint system with 3 tanks and a mixer. Each paint tank has 3 level sensors and 2 valves. The mixer has 1 valve and 1 sensor. The design also includes two working modes, one manual and one automatic. In the first one, the operator is responsible for opening and closing paint valves, controlling the mixing time and then emptying the mixer. In the automatic mode, the operator provides, using the HMI screen, the paint code which is the combination of paint colors to be mixed. Then the system starts the process by opening and closing valves. When the paint is mixed, it calculates the mixing time and finally empties the mixer before executing the process over again. The Petri net associated with this process is more complex. As each tank provides four possible levels of paint (from 0 to 3 with the value 0 for an empty tank), the three tanks system offers 64 (4x4x4) different color possibilities, which implies 64 different mixing processes. Each tank's color has a different density, so the required time to fill a tank varies depending on the chosen color and its quantity.

3.2 Design and control of a hybrid system

The formal definition of a system may be found in Appendix A, so here we focus on a particular subclass of systems which are hybrid-type systems. As stated in [5], a hybrid system is a dynamic system with both continuous and discrete components. For example, a car is a hybrid system where the engine injection is continuous and its is controlled by a microprocessor which is discrete. Such systems generally have a continuous behaviour, but when we want to implement their control we find that for cost reduction or design simplicity purposes, the data is collected using sensors that are either discrete or continuous but their values are turned discrete by A/D converters to allow digital processing.

Real systems: Based on the description of a hybrid system that was given above, we face here the first design problem. We need an industrial system with a controller that manages a particular process. We also should be allowed to recollect data from the network that links the system with the controller. But it is often impossible to use a real industrial system, first because they are usually very expensive and difficult to build, and secondly because over it will be performed experimental tests with often uncertain results and therefore under certain circumstances the control of the system could be lost causing damage or accidents. The solution to this problem is to model and simulate in real time the physical system on a computer. A model never represents the whole complexity of a real system, but in some cases, this approach has a range of validity and the models may be lawful in a range of works.

Real time simulation: The main difference between a real time simulations compared to other simulations is that they must meet certain time constraints regarding the behaviour of the system under study. In general, a rule widely applied in these simulations is that they must comply with a response time **Tr**, defined as the period of time between a data input and the output response. One of the conditions to be satisfied in a real-time simulation is to limit this parameter to a maximum value. Sometimes, it is even required that this response time is constant along the simulation.

Control: The Control is performed using a PLC, the digital I/O module and a PCI interface that connects the devices to the computer that performs the real time simulation of the system. The PLC is connected to a HMI computer that allows the entry of parameters and the visualization of the system behaviour.

3.3 The Sliding Door System

A sliding door is a special type of doors that slides horizontally using a bearing system. It can be operated manually or by an electromechanical mechanism. It has certain characteristics such as mass, friction, etc. that will be modelled below.

3.3.1 Model

We consider a sliding door operated by a motor and a PID position controller. We assume that it is a 1st order system with a pure delay as described in Figure 3.2). The dynamics of the system was designed so that the transition between the input 0 and 1 takes about 5 seconds, of which 2 seconds belong to the delay introduced by the control and 3 seconds to the dynamic shifting of the door. The response of the system to a step input is depicted in Figure 3.3. To generate this

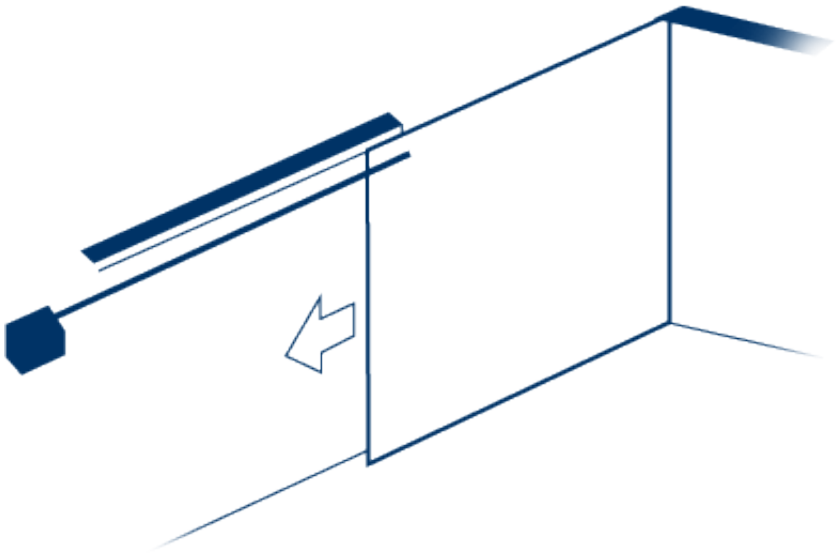


Figure 3.1: Graphical representation of a sliding door.

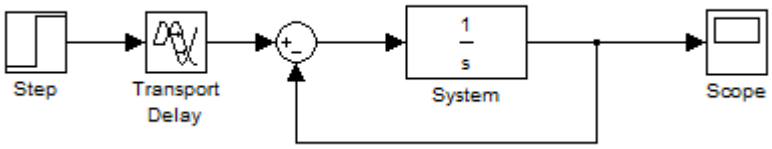


Figure 3.2: Model of the controlled sliding door system

dynamic behaviour we used the Laplace transform. We proceeded as follows:

Let the generic 1^{st} order transfer function as:

$$G(s) = \frac{b_1 s + b_0}{s + a_0}$$

Where $a_0 > 0$ (Stable System) This system has a unique pole of value $-a_0$, the unitary step response is:

$$h(t) = \pounds^{-1}\{\frac{G(s)}{s}\} = h_f + (h_i - h_f)e^{a_0 t}$$

Where h_i is the initial value of the response, and h_f is the final value. The system reaches the 95% of the final value in $3\mathcal{T}$ (Time constant). If we want that the system takes about 3 seconds to change its state, we need thus:

$$3\mathcal{T} = 3s$$

$$\mathcal{T} = 1s = 1/a_0$$

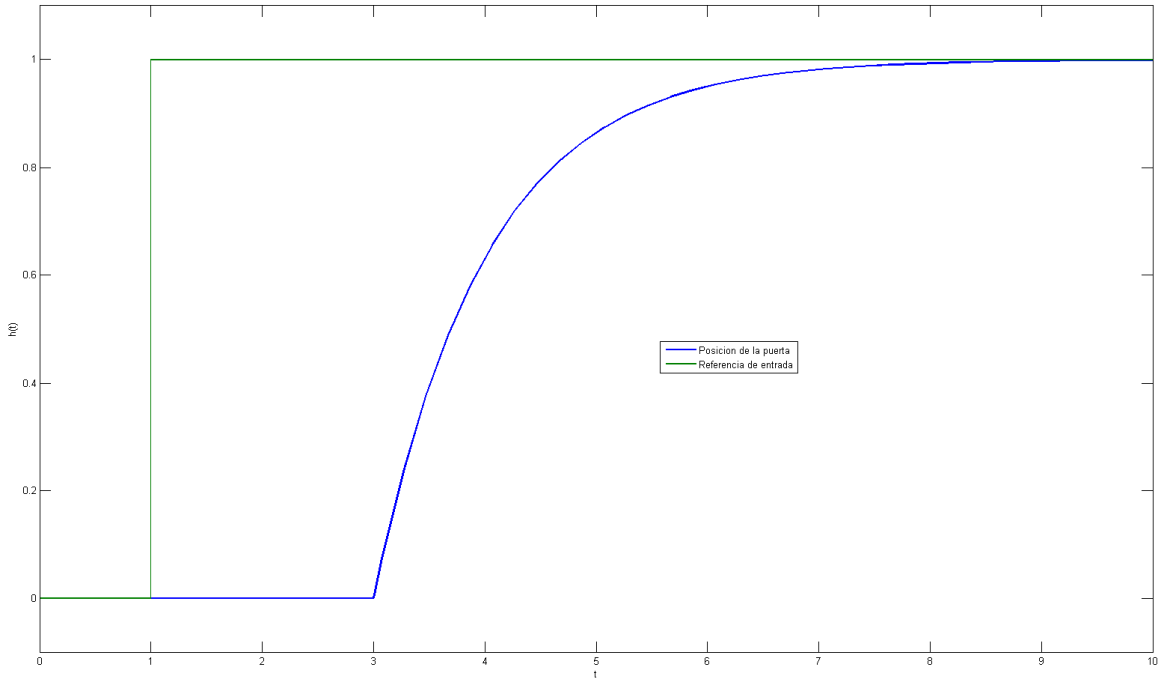


Figure 3.3: Step response of the sliding door system

$$a_0 = 1$$

To find the constant values b_0 y b_1 we use the final and initial value theorems:

$$h_f = \lim_{t\to\infty} h(t) = \lim_{s\to 0} sH(s) = \lim_{s\to 0} G(s) = \frac{b_0}{a_0}$$

$$\Rightarrow h_f = \frac{a_0}{b_0}$$

$$h_i = h(0^+) = \lim_{s\to\infty} sH(s) = \lim_{s\to\infty} G(s) = b_1$$

$$\Rightarrow h_i = b_1 \Rightarrow b_1 = 0$$

According to these values, we obtain the following transfer function (TF):

$$G(s) = \frac{1}{s+1}$$

We represent the TF using the block diagrams available in Simulink. There we used adder blocks, integrators and one unitary feedback as depicted in Figure 3.4).

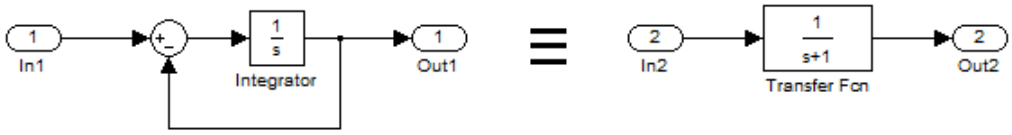


Figure 3.4: 1st Step model of the sliding door system

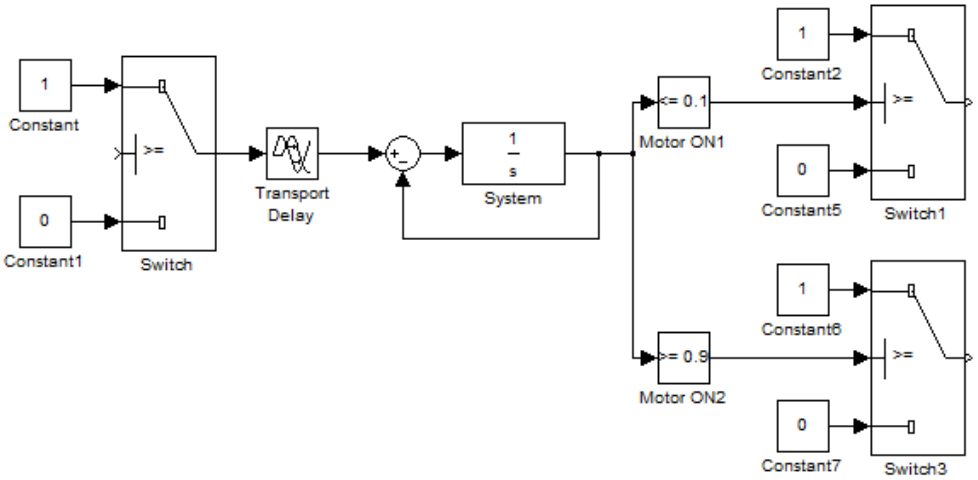


Figure 3.5: 2nd step model of the sliding door system

Further, we added the 2 seconds delay with a special block provided by Simulink. We then make discrete the continuous time variables. We used a selection switch, with binary selection values (0 or 1). The final model is shown in Figure 3.5

In a next step, we added the blocks that connect the system model with the I/O using the Contec PIO card. As we observe in Figure 3.6, there is one block that acts as an input interface and another acting as an output interface. When we added those I/O blocks, several internal parameters had to be configured. These parameters are as follows:

```
Format: 32 1bit-Channels
Channel Vector: [17 18 19 20 211 22 23 24]
Sample time: 0.01
PCI slot (-1: autosearch): [5,4]
```

The most important parameter is the Sample Time that we will detail next.

Sample Time: Why is this parameter important? At the beginning, we did not pay much attention to this value. We followed all the steps to completely finish the model. Then, we selected the integration routine ode45 (Dormand-Prince), one of the most commonly used to balance efficiency vs. accuracy, and we tried to make the compilation of the model to simulate

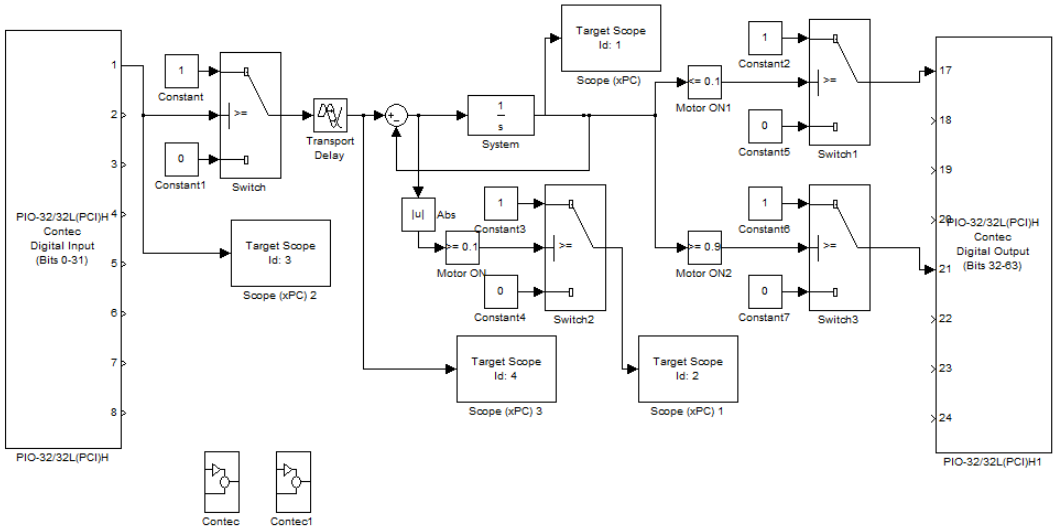


Figure 3.6: 3rd step model of the sliding door system

it on the Target PC. There appeared the first inconvenience with the chosen routine, which has been overlooked at first.

Matlab gives an error when we want to simulate a real-time system that uses an integration routine with a variable step. It is required that the integration routine has a fixed step because the Simulink encoder does not support a variable step. After selecting the correct integration routine (it was decided to use ode2-Heun), we selected the value of the step. We realized that, depending on the complexity of the system to be simulated, it is very easy to overload the Target machine CPU. In fact, when we made the first attempt, the simulation stopped due to an overload of the CPU. The relationship between the integration step and the frequency sampling is that these values must be the same. Matlab does not allow these values to be different, which otherwise is very logical. It would not be useful to sample in a higher frequency than the results are updated by the integration routine, since the samples would be always the same over a period.

After several tests, the sample time was set at $T = 0.01s$. It is enough because the system dynamics is 100 times slower.

3.3.2 Control

Previously, we explained that the door control have two modes of operation, the manual mode and the automatic mode. In the manual mode, the operator is responsible for selecting whether the door is opened or closed while in automatic mode, the door will be opened and closed in a continuous loop managed by the controller. This behaviour was programmed into the PLC using the Ladder language. The visualization and the setting of modes are programmed in the HMI (SCADA) using the TIA Portal. Figure 3.7 shows the HMI touch screen where inside there are a mode and a position switches, four text boxes with binary values and a building with a sliding glass door, which responds to the signals generated by the PLC showing the current status. The door will open or close according to the state of the simulated system in Target PC. We also observe other buttons as *System screens* related to the HMI configuration which are very useful if we want to debug a program, because they allow the operator to manually introduce values of

digital inputs and outputs while the system is running.

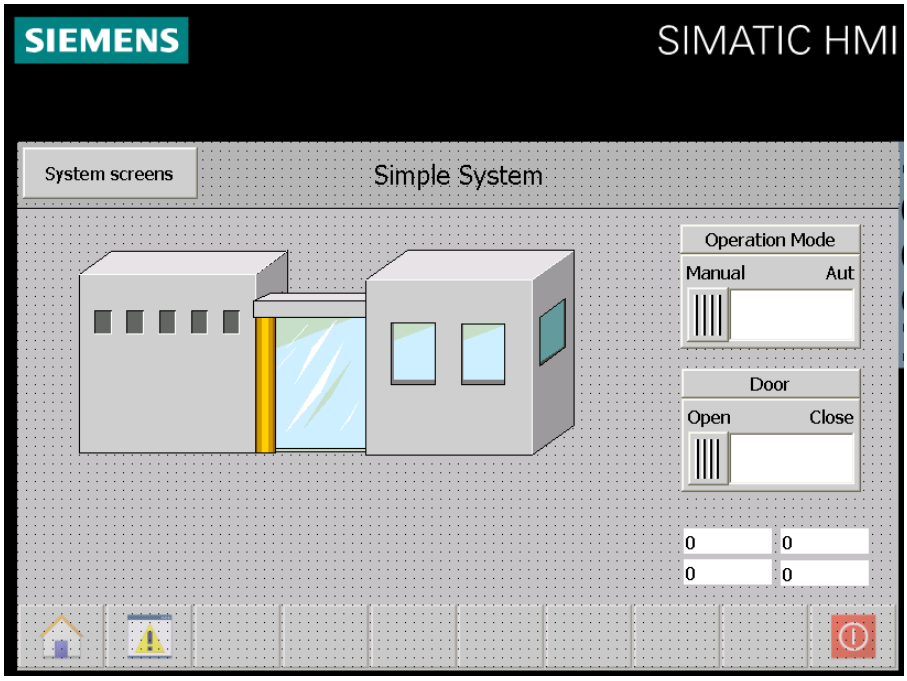


Figure 3.7: The SCADA interface for the sliding door

3.3.3 Petri Net model

We observe that the modelled system is really simple and therefore, without much effort, we should be able to manually identify its Petri Net. In Figure 3.8 we show the manually constructed Petri Net modelling the system places and transitions. If we analyse more closely the system, we see that there are only four states. The door may be open (state 1), closing (state 2), closed (state 3) or opening (state 4) which are the 4 places that are in the Petri net. In turn we find the four transitions on these states and two transitions that appear in two particular cases. The first case is when the door is opening and without becoming completely open, we decide to close it, which makes the system going directly to the state closing. The second case is simply the opposite, where the door is closing and we decide to open it before it is completely closed.

We have to note that the Petri Net in Figure 3.8 has M0 and M1, which does not mean that there are 2 outputs. It means that the output M (referring to the Motor) has value 1 or 0 depending on the sliding direction. We used this notation for a better understanding of the behaviour of the system.

This manually constructed Petri Net presents interesting proprieties that we will discuss in detail in the next paragraphs.

Unreachable States: The manually constructed Petri Net, does not contain all the possible states of Inputs and Outputs. The system has 1 digital output (M) and 2 digital inputs (S1, S2), so totally it has 3 binary variables, with a possible combination of 2^3. It means there are 8 possible states while, in the Petri Net, we only observe 4 states. The table 3.1 shows all the possible combinations and only states identified in the Petri Net are labelled.

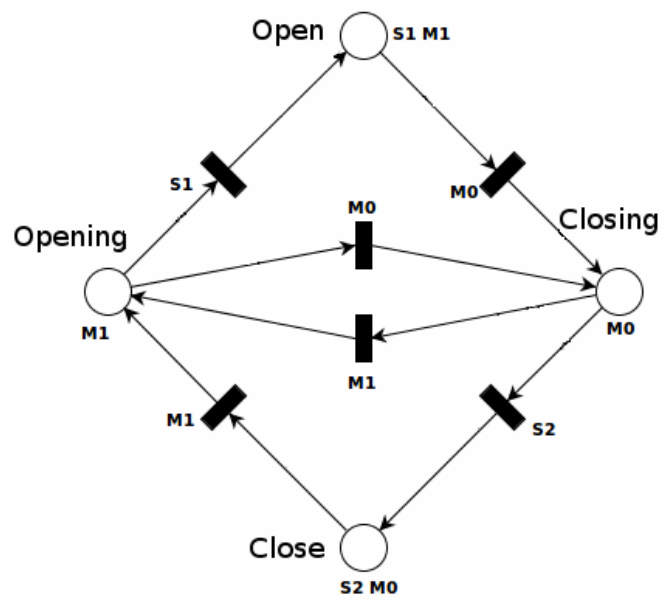


Figure 3.8: The manually made Petri Net of the sliding door system

A question someone might ask here: what would happen if an attacker modifies the data frame to make believe the controller that the value of Input and Outputs is one of the possibilities that does not appear in the Petri net? For example, the state vector [0 1 1] implies that the door is closed and open at the same time. The designed controller does not support this state, so it is a vulnerability that an attacker can exploit to bring the system to an unsafe state. Here we exposed one of the security vulnerabilities to be avoided in this system.

Timeless states: Another property of this Petri Net is that there is no information related to the residence times inside states or transitions. This parameter is usually omitted when designing Petri networks, but it is very important for securing the designed system, because an attacker could use the absence of this parameter to provide a wrong view to the system operator (HMI). For example, the door is closed and the operator gives the order to the controller to open it. How much time does the *Opening* state should be active? As we saw previously, this system was designed to have a 2 seconds delay and a dynamic of approximately 3 seconds. So we expect that the system remains at least 5 seconds in the *Opening* state. In the same way, if the door is open and the operator wants to close it is logical to guess that the system requires the same delay for the *Closing* state. But this information is not available in the Petri Net. An attacker who wants to keep the door open to get into the the room detects the door-closing order form the controller and immediately injects a faked packet informing that the door is *Closed*. In this way, the controller detects that the door reaches its end position sensor but in reality, it is still open.

This type of attacks can not be reflected in the Petri network, because the system does not deviate from its normal behaviour. What is happening is that the *Closing* state is timeless meaning that the system does not remain for a specific well-known time in that state to guarantee the correct way of working.

State number	M	S1	S2	Label
1	0	0	0	Closing
2	0	0	1	Close
3	0	1	0	-
4	0	1	1	-
5	1	0	0	Opening
6	1	0	1	-
7	1	1	0	Open
8	1	1	1	-

Table 3.1: All states of the sliding door system

3.4 The Paint Mixing System

3.4.1 Model

The system has three paint tanks: one for red paint, one for green paint and another for blue paint. Each tank has three level sensors and two valves, one for filling and the other for emptying. The drain pipes go to a fourth tank, to mix the paints. This tank also has one level sensor and one valve. Filling the tanks starts when the valves 1, 2 and 3 are open respectively. Each color has a different density, so the required time to fill a tank is different for each color. Therefore, we can find 10 digital inputs and 7 digital outputs. In total, we have 17 binary variables and thus, $2^{17} = 131072$ possible states of the paint mixing system.

The physical laws that dominate the system are:

$$Q = A.v$$

Where Q is the flow, A is the area and v is the speed of the fluid.

$$Q = A.dx \longrightarrow \int Q = \int A.dx$$

If the area is constant,

$$\longrightarrow \int Q = A. \int dx$$

Finally, we obtain:

$$\int Q = A.x \longrightarrow x = \frac{\int Q}{A}$$

Paint Tanks: To simplify the model, we assume that the area of tanks is $A = 1$. As we see in Figure 3.9, the block has 2 ports `Pipe1T1 and Pipe2T1`, both get into the adder block with opposite signs. `Pipe1T1` is for filling and `Pipe2T1` is for draining. The output of the integrator is the paint level of the tank (continuous variable) and we observe that is connected to a block that has as outputs the 3 digital levels `Level 0, Level 1 and Level 2` (discrete variables).

Level sensors: We used switches as depicted in Figure 3.10 to convert the continuous time variable of the paint level to discrete values.

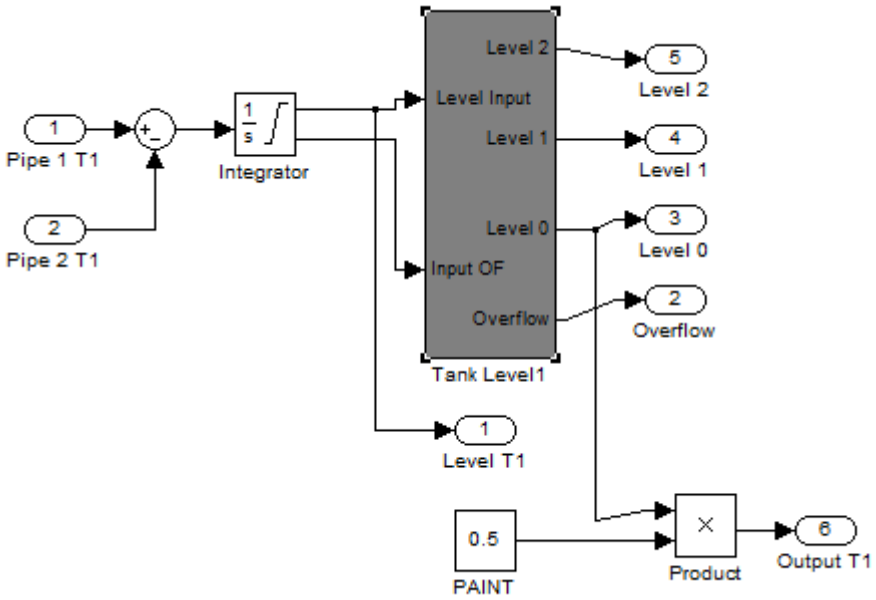


Figure 3.9: Model of the paint tanks

Valves: The flow control valves (continuous variables) are modelled as multipliers with 2 inputs. One of them is the paint flow and the other is the control parameter (digital variable) that can be 1 or 0 depending on whether the valve is open or close as depicted in Figure 3.11.

Mixer: Another fundamental part of the system is the paint mixer as depicted in Figure 3.12) where inside it, the paint is mixed for a fixed time. As we explained previously, the tank has one level sensor with a value of 1 when the tank is empty. It also has one valve to drain the paint from the tank after it is prepared.

The complete system: The complete system with all the different blocks and connections is shown in Figure 3.13.

The interface with the outside: In the same way as the sliding door system, we connected the paint system to the I/O block with the Contec PIO card. All the valves and sensors are wired to the different blocks as depicted in Figure 3.14.

3.4.2 Control

The paint system control, like the sliding door system, has two working modes. The first is the manual mode where the operator opens the valves using buttons in the HMI screen to define the paint level in the tanks, then the mixer is filled, next the mix is performed and finally the mixer is emptied. The second mode is the automatic mode which is totally different. It works with 3 input text boxes where the operator introduces the desired paint code and starts the system. The system follows the working logic written in a VB Script (See Appendix D) together with the LADDER control program uploaded in the PLC. The program is in charge of opening and

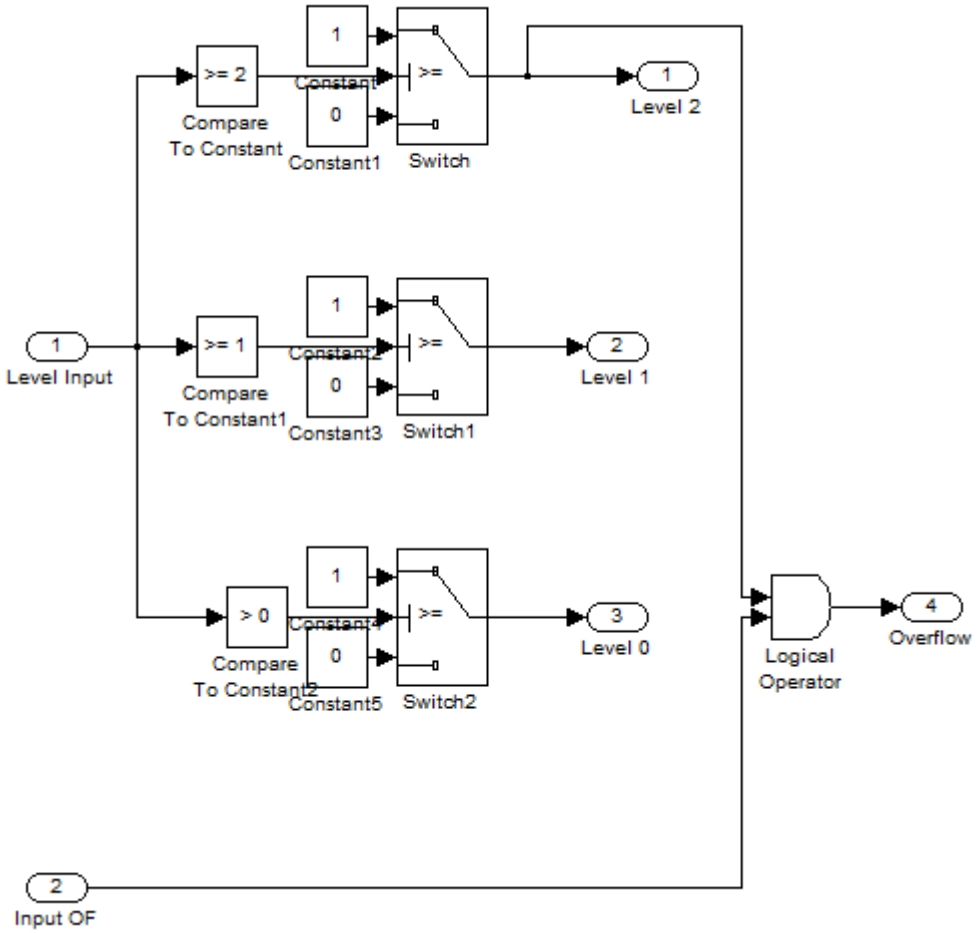


Figure 3.10: Paint level conversion block

closing the valves. It chronometers the mixing time and finally empties the mixer to start the process again.

In the automatic mode, the operator must introduce the identification code of the paint color to be prepared. The codification used here is very simple: we have 3 boxes (one for each tank) where in each one, can be introduced one of 4 different levels of paint. The operator selects between [0 1 2 3] (level 0 to leave the tank empty). This implies $4 * 4 * 4 = 64$ possibilities. The process starts using a *START* button available on the HMI screen and the evolution can be followed trough the Target PC screen where the real time values of the variables are displayed. It is very important to mention that this value is the real one because it is generated by the system and not by the controller, so it could be the same or different from the value displayed on the HMI.

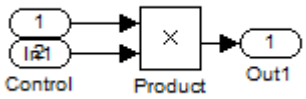


Figure 3.11: Model of the control valves

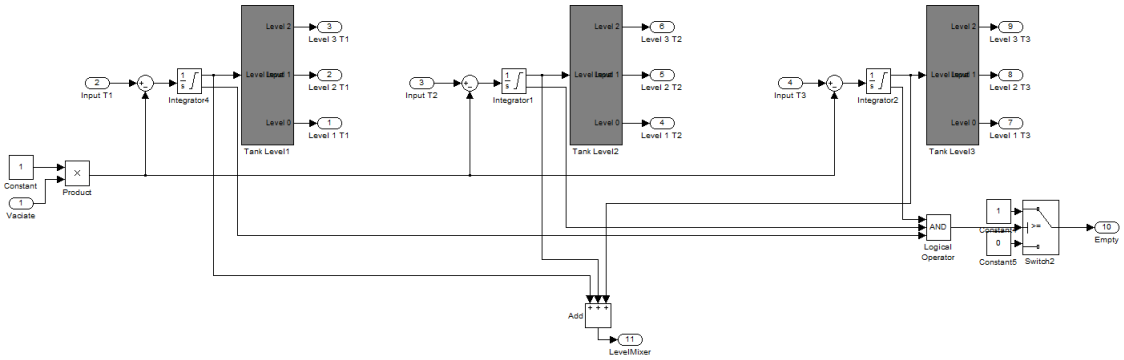


Figure 3.12: The paint mixer

3.4.3 Petri Net

The Petri Net modelling the paint system is not as simple as the sliding door system's one. First, imagine that the operator selects the automatic working mode and introduces one of the 64 different paint codes to the HMI. The controller starts opening the adequate valves until the paint level in the tanks reaches the adequate values. Next, the paint is drained into the mixer and mixed taking the necessary period of time. Finally, the mixer is emptied. The associated Petri Net describes only the process for this specific paint code. It means that if the operator changes the code, the Petri Net changes too, and this happens with the 64 different paint combinations. Therefore, it is not easy to make the Petri Net manually since the number of states is important.

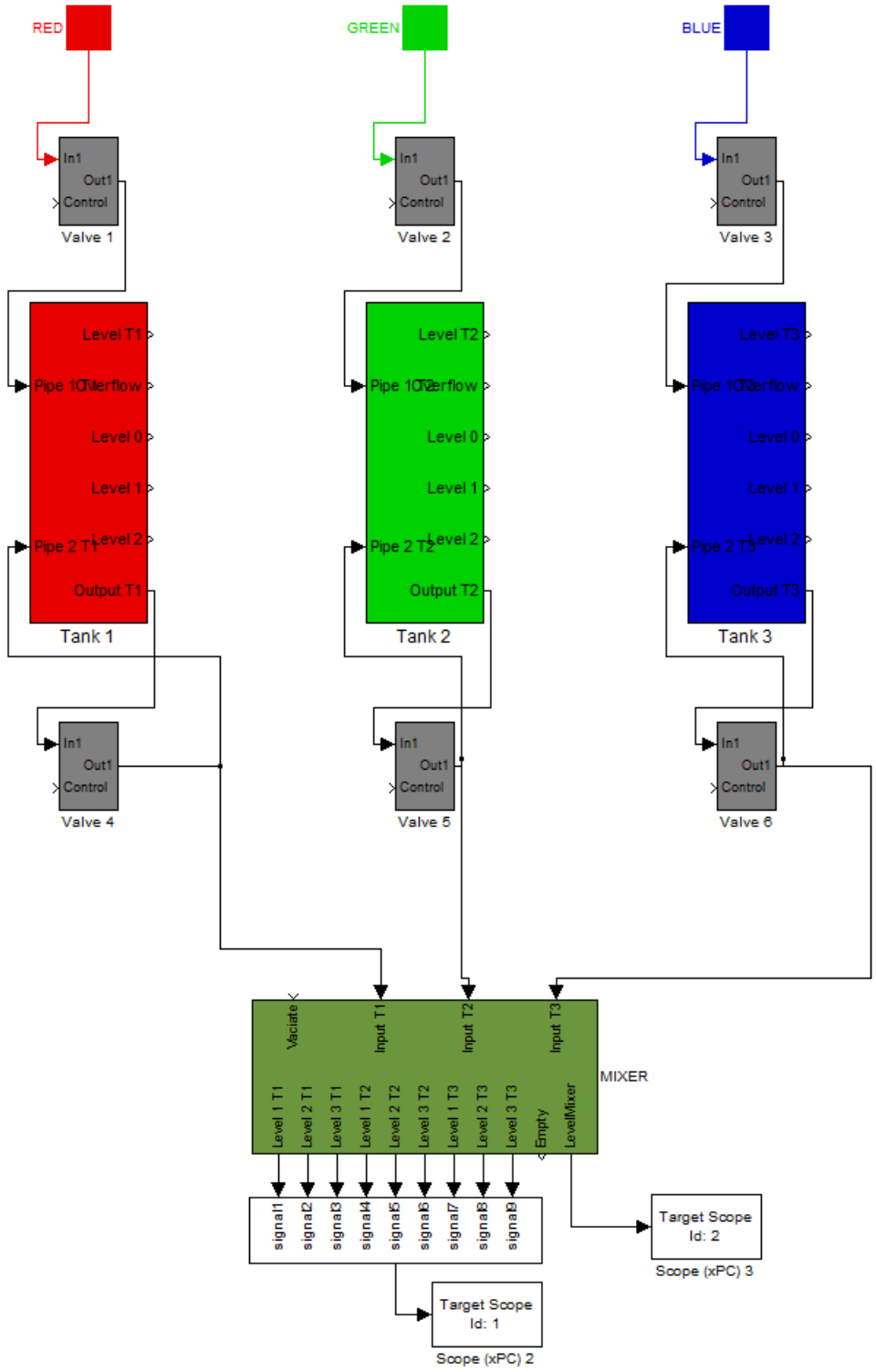


Figure 3.13: Overall design of the mixing paint system

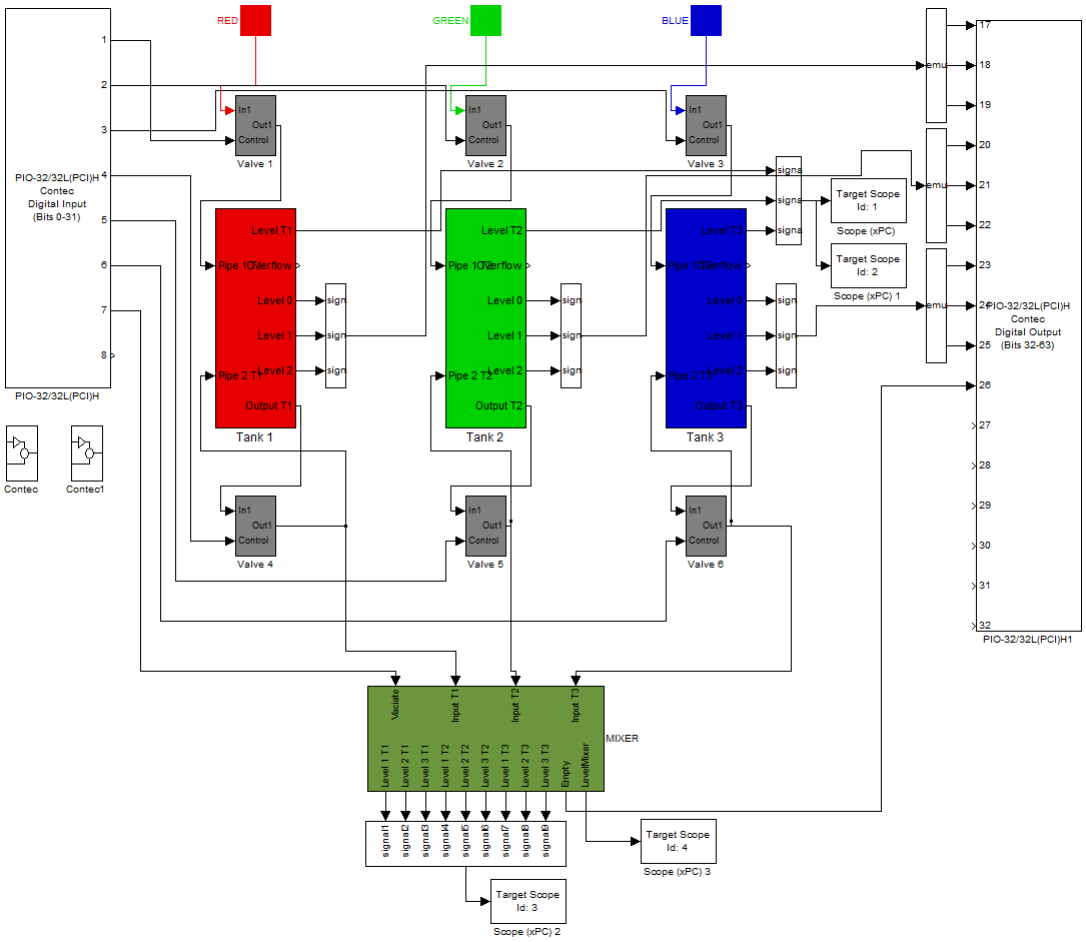


Figure 3.14: Connection of the Simulink model of the paint system to the IO card

Chapter 4

Analysis of the PROFINET protocol

We have already described in chapter 3 the design of two hybrid systems (one simple and another with a greater degree of complexity) and their respective controls. We then implemented them on our testbed to be able to capture the PROFINET frames transmitted in the network between the PLC and the digital I/O module. In this chapter we will present our methodology for the analysis of the data carried by the PROFINET protocol interconnecting the control devices as shown in Figure 4.1.

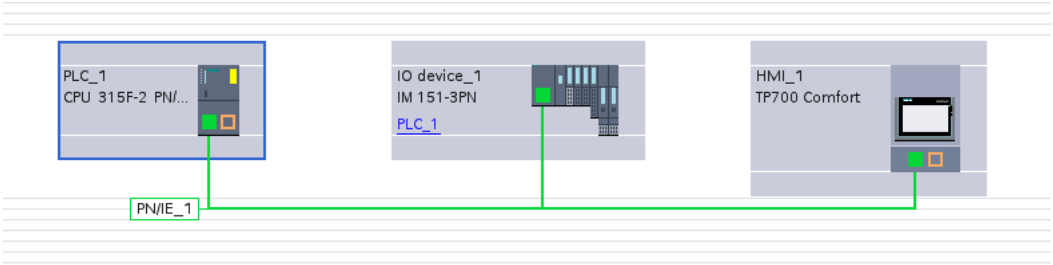


Figure 4.1: Schematic interconnection of the PLC, HMI and Digital I/O module using the PROFINET protocol

4.1 Overview

The PLC and its digital I/O module are connected through an Ethernet network and the data frames are encapsulated in the PROFINET protocol. PROFINET is the standard industry interconnection protocol specified by PROFIBUS and PROFINET International (PI) for automation processes. It is essentially an Ethernet datagram that must meet strict specifications of time. It is also commonly known as Ethernet in Real Time. PROFINET enabled networks [6] are adapted either for modular machines or any input-output distributed device. Therefore they can be described as a type of network that is responsible for the communication between control devices in field. Among the stringent requirements of real-time communication, are deterministic performance, low device resource requirements and a response time in the range of 5-10ms to factory automation and 1ms for motion control applications. These temporal requirements must be met by control devices or input-output modules, and the switches that interconnect them. The switch must allow connections at 100 Mbps Full Duplex and priority handling of packets.

One of the tasks of this project was to discover how the PROFINET protocol works. There is not too much information about its implementation in Siemens devices. Thus, we applied a reverse engineering technique to acquire information about the decoding of IO values inside PROFINET frames. We made a series of tests to acquire detailed parameters. After that, we automated this task using some scripts to perform extraction of IO values.

The PLC communication with peripherals uses the PROFINET IO protocol. It is fully defined by the data exchange between controllers (Master Devices) and the distributed devices (Slaves devices). PROFINET IO is designed for fast data transmission between field devices based on Ethernet and complies with requirements of bus cycles of a few milliseconds. Any system based on PROFINET IO consists of the following devices:

- An I/O controller, responsible for managing automation tasks. It exchanges data with I/O devices.
- An I/O device, which is a distributed sensor/actuator device monitored by an I/O controller.
- An I/O supervisor, which is normally a privative software running on a PC or a HMI device. Its task consists in handling the system parameters, showing process states and diagnosing individual I/O devices.

A PROFINET IO system requires at least one I/O controller and one I/O device. Different system configurations can be set: multiple I/O controllers for a single I/O device, single I/O controllers for multiple I/O devices, and multiple I/O controllers with multiple I/O devices.

An I/O device consists of a device with slots, sub-slots, and channels. Sub-slots are subcomponents of a slot. Each sub-slot is assigned a number of I/O points or channels. A channel is the PROFINET IO term which refers to one physical discrete input, discrete output, analogue input, or analogue output. Except for slot zero, every other slot and sub-slot contains status, diagnostic, and alarm data for the channels of that slot. For I/O points transferred to an I/O controller, provider status and diagnostic information is automatically transferred on every IO scan cycle. For I/O points transferred to an I/O device, consumer status is returned to the I/O controller [7]. Note that there actually is no real slot zero. Slot zero refers to the device itself and carries no IO data. In place of IO data, slot zero manages all the generic device data like the vendor name, product catalogue number, and software and hardware version information, as well as other similar information.

We mentioned above that the components of a PROFINET network communicate over Ethernet. Each PROFINET IO frame is encapsulated inside an Ethernet frame. It has a reduced length message telegram because the overhead of the protocol has to be low. Table 4.1 shows an Ethernet frame that transports a PROFINET datagram.

Standard Ethernet						PROFINET Specific			
PreAmb	Sync	MACDes	MACSrc	Prior	EthTyp	ID	Data	State	FCS
7 Bytes	1 Byte	6 Bytes	6 Bytes	4 Bytes	2 Bytes	2 Bytes	40-1440 Bytes	4 Byte	4 Bytes

Table 4.1: Ethernet-PROFINET Frame

4.2 Type and format of frames

Data capture is done using Wireshark[1]. This software performs the capture and analysis of packets. It is an Ethereal's successor. It provides a set of functionality similar to TCPDUMP, incorporating only a graphical interface that simplifies the task of adding filters and organization of captured information. Using this software we captured packets from the network connecting the PLC with the I/O module. Wireshark captures all the traffic flowing through the network, then we can add filters to select the data of interest. The captured packets can be stored in `.pcap` files.

We captured several packets to verify that when connecting an I/O device to a PROFINET network, this is detected and configured automatically by the MASTER device. This is done by the exchange of a PROFINET Configuration Management (CM) frame, trough it the master defines the offsets inside the data field of the frame PROFINET IO. The offsets specify the data location of outputs of the I/O modules and also where the Master device expects to receive Inputs values. Figure 4.2 shows a data capture of the sliding door system; there we see both PNIO-CM and PNIO data frames.

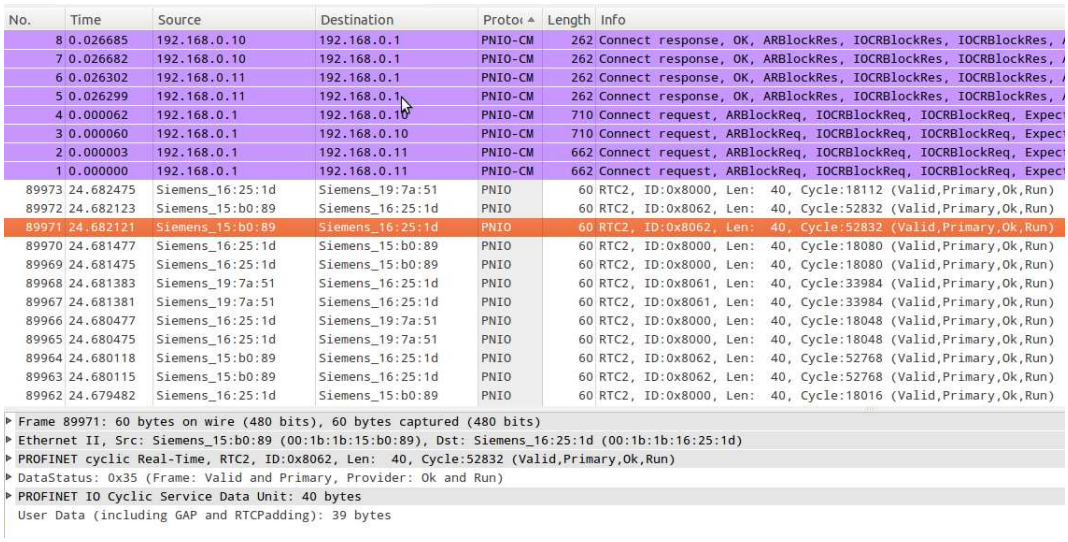


Figure 4.2: Data capture with the Wireshark tool

We verified the settings made by the Master trough PROFINET CM by selecting arbitrary values in the I/O device and verifying them through capturing PROFINET IO frames and decoding data. We found that each module, composed of 8 I/O channels, receives and sends information using 1 byte. This implies that each channel is coded on 1 bit.

Table 4.2 shows an example of a frame received by the master (PLC). This frame carries information related to inputs on the I/O device. We can observe the values in hexadecimal and in binary, and also the order of the digital inputs (from 8 to 1) according to both the LADDER program and the positions of the physical connections. The first line shows the data in the same way as displayed by the TIA portal. The second line is the binary conversion. The third line is the order of the digital inputs as provided by the LADDER program. The fourth line is the physical positions of inputs connections. In the example, we observe that the PROFINET IO

[1]http://www.wireshark.org/

frame contains 80 and 41 in Hex separated by 00. The value 80 means that in the first block, input 8 is at high. The value 41 means that, in the second block, inputs 7 and 1 are at high (positions according to the order defined in the Ladder program).

Data (Hex)	80								00								41								00							
Data (Bin)	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0
Ladder Order	8	7	6	5	4	3	2	1	0	0	0	0	0	0	0	0	8	7	6	5	4	3	2	1	0	0	0	0	0	0	0	0
Physical Order	8	4	7	3	6	2	5	1	0	0	0	0	0	0	0	0	8	4	7	3	6	2	5	1	0	0	0	0	0	0	0	0

Table 4.2: Example of digital input data

4.3 PROFINET reverse engineering

Through Wireshark captures, we noticed that the PNIO-CM frames contain certain information relevant to the location of the data inside PN-IO frames. This information is presented as follows: inside the frame PNIO-CM, there are two fields, one named `IOCRBlockReq: Input CR` and other named `IOCRBlockReq: Output CR`.

Within `IOCRBlockReq: Input CR` there is an API field that has information about the number of I/O objects configured in the PLC and the offset from baseline for each SLOT. With this packet, the PLC indicates to the I/O block, in which part of the PNIO frame it expects the module to put the data of the digital inputs.

`IOCRBlockReq: Output CR` is used by the PLC to inform the module where it will send the value of the digital outputs.

To clarify a little more how the protocol works, we attach two figures showing the data frames. In Figure 4.3 we observe that the PLC is informing the module that it recognizes 7 input objects.

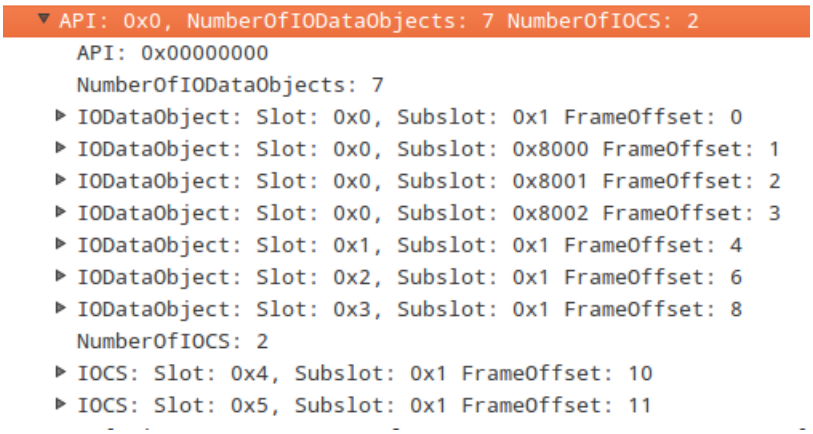


Figure 4.3: Wireshark capture of the PNIO-CM Input CR

- The Slot 0x0 manages all the generic device data like the vendor name, product catalogue number, and software and hardware version information, etc. We see that data fields with offsets `0,1,2,3` will carry information related to them.
- The Slot 0x1 is the Power module; the PLC gives him the offset `4`.
- The Slots 0x2 and 0x3 correspond each one of them to a module of 8 Digital Inputs, and the PLC reserves the offsets `6,8` respectively.

In figure 4.4, we see that the PLC informs the module that it recognizes 2 Output objects.

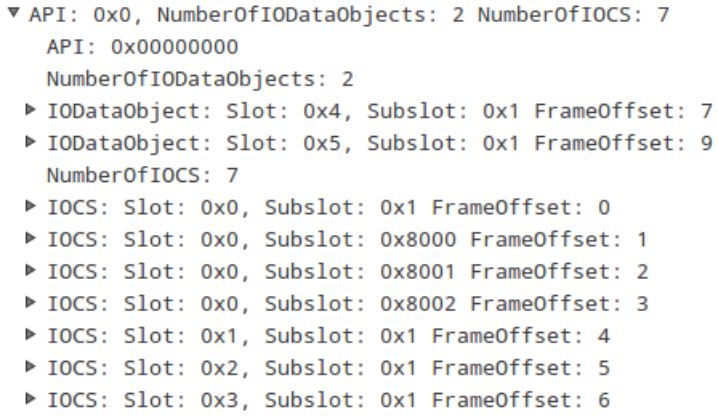


Figure 4.4: Wireshark capture of the PNIO-CM Output CR

- The Slots 0x4 and 0x5 correspond each one to a module of 8 Digital Outputs and the PLC reserves the offsets `7,9` respectively.

It is interesting to note that this disposition was not made automatically by the PLC. When the PLC was configured in Siemens TIA Portal, we manually added all this blocks in the right positions and connected them. The PLC only sends this information to the blocks and waits for the confirmation. If the confirmation does not arrive, a red light starts blinking in the PLC and the I/O blocks, notifying the problem to the operator.

Based on all these acquired information, we were able to identify and localize each I/O value in a PNIO data frame. Figure 4.5 shows the offsets of the I/O values in a Profinet data frame in the case of the Sliding Door system.

Figure 4.6 lists the inputs and outputs used for the Sliding Door system and the Paint Mixing System. For each used I/O channel, the label and the position in the Profinet data payload are given.

Verification : We made 2 tests to verify the input/output modes.

In the first test, we put all the Digital Inputs in High. This was made using a Matlab design with 16 digital outputs connected to VCC (24V). After that, we captured the PNIO frames that went from the I/O block to the PLC. The obtained data field had this values:

```
xx xx xx xx xx xx FF xx FF xx ...
```

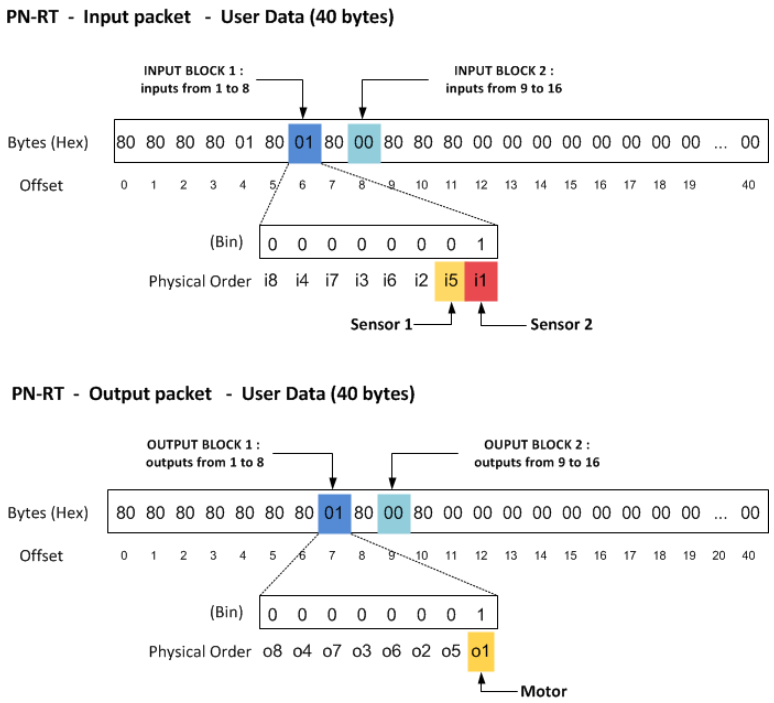


Figure 4.5: The sliding door I/O offsets in a data frame.

Remember that the information is in Hexadecimal format, so [F F] is equivalent to [1111 1111] in binary. The result shows well that all 16 digital Inputs, held in slots 0x2 and 0x3 with offsets 6 and 8 respectively, are set at the value 1.

In the second test, we assigned High values to all digital Outputs using a LADDER logic Program loaded in the PLC. The data field in PNIO frame that went from the PLC to the I/O block has the values at offsets 7 and 9 set at high which corresponds to slots 0x4 and 0x5 reserved for the 16 Digital Outputs:

```
xx xx xx xx xx xx xx FF xx FF xx ...
```

4.4 Data storage and conversion

Using the packets captured by Wireshark, we are able to retrieve all the information related to the system inputs and outputs, which allows us to reconstruct the different system events. An event is created when the value of an input or an output changes. To each created event, we associate the time during which the system remains in the corresponding state. The extracted I/O data is converted into another format of files with the `.XES` extension where the system events logs are stored. This `.XES` file id then processed by a process mining tool to identify the system process and infer the model of both control and system.

Input positions and labels

Inputs	I1	I2	I3	I4	I5	I6	I7	I8	I9	I10	I11	I12	I13	I14	I15	I16
Byte Offset	6								8							
Bit Offsets (on two bytes)	0	2	4	6	1	3	5	7	0	2	4	6	1	3	5	7
Sliding Door System Inputs	Sensor2	-	-	-	Sensor1	-	-	-	-	-	-	-	-	-	-	-
Paint Mixing System Inputs	L0T1	L1T1	L2T1	L0T2	L1T2	L2T2	L0T3	L1T3	L2T3	Empty Mixt	-	-	-	-	-	-

Output positions and labels

Outputs	O1	O2	O3	O4	O5	O6	O7	O8	O9	O10	O11	O12	O13	O14	O15	O16
Byte Offset	7								9							
Bit Offsets (on two bytes)	0	2	4	6	1	3	5	7	0	2	4	6	1	3	5	7
Sliding Door System Outputs	Motor	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Paint Mixing System Outputs	V1	V2	V3	V4	V5	V6	Vaciate	-	-	-	-	-	-	-	-	-

Figure 4.6: I/O positions and labels.

XES is an XML-based standard for event logs[2]. Its purpose is to provide a generally-acknowledged format for the interchange of event log data between tools and application domains. Its primary purpose is for process mining, i.e. the analysis of operational processes based on their event logs. However, XES has been designed to also be suitable for general data mining, text mining, and statistical analysis.

In order to filter and store the data captured by Wireshark, a script in Lua[3] language was written (see Appendix C). Lua is a lightweight embeddable scripting language designed to be used as a general purpose extension language[8]. It is an extension language because it helps to extend applications through configuration, macros, and other end-user customizations.

In Wireshark, Lua can be used to write dissectors and taps. We use it here through our Lua script to filter the PROFINET frames exchanged between the PLC and the I/O block. The script intercepts both PNIO-CM and PNIO frames and uses the Syslog logging system to store the data of the intercepted frames in a Mysql database. Below are some details about how the script functions. The PLC has the IP address 192.168.0.1 and the MAC address 00:1b:1b:16:25:1d (Siemens_16:25:1d) while the I/O block has the IP address 192.168.0.10 and the MAC address 00:1b:1b:19:7a:51 (Siemens_19:7a:51).

- First of all, the script waits for the reception of PNIO-CM (PROFINET Configuration Management) frames. These frames are generated when the driver is uploaded to the PLC via the TIA Portal. The script needs to receive these frames at the first place because they hold the offsets specifying the data location of inputs and outputs in the PROFINET IO frames. The script uses the following filter to detect the PNIO-CM frames *_io.opnum == 0 and eth.dst == 00:1b:1b:19:7a:51 and eth.src == 00:1b:1b:16:25:1d"]*. It extracts the fields values related to the input and output frameIds, slots numbers, data objects and their offsets, and provider/consumer status and their offsets. The script can then start the analysis of captured PNIO frames.

- The script looks for PNIO data frames using the filter *[pn_rt]* and distinguishes between input frames *[des == "Siemens_16:25:1d" and src == "Siemens_19:7a:51"]* and output frames *[des == "Siemens_19:7a:51" and src == "Siemens_16:25:1d"]*. For each frame

[2]http://www.xes-standard.org/
[3]http://www.lua.org/

matching the filters, a new log entry is inserted into the database. The log contains a set of information including the type of the frame (I/O), the source, the destination and the data.

To use a Lua script untitled *file.lua*, run the following command from the command prompt:

```
$ tshark -X lua_script:file.lua
```

The Lua script content will be executed on Wireshark real-time captured packets. It is also possible to apply the script on a .pcap file using the command line option -r <infile> :

```
$ tshark -r file.pcap -X lua_script:file.lua
```

To analyse and convert the data that has been stored in the database, we have made two scripts in Perl. The first one parses the log entries and stores data using a more detailed representation. The second script acts on data to detect the system events and generate the .XES file.

- The second script maintains a state list where values of all inputs and outputs are saved. For each entry corresponding to a PNIO data frame captured between the PLC and the I/O block, the script checks if there are changes in the values of inputs or outputs. If a change occurs, the state list is updated with the new values of inputs or outputs. Note that the script checks only the values of input channels if the entry corresponds to an input frame and only the values of output channels in the cas of an output frame entry.
- For each update of the state list, the script creates a new entry in the `.XES` file where an event log is inserted. The event log entry holds the state of the system (Inputs or Outputs that are active), the parameter causing the transition and the residence time in that state.

After applying the different scripts on the captured packets from the Sliding Door system network, we got the following .XES file. It holds the different system events of one execution cycle of the system in the automatic mode. Starting from the state *Closed*, the door is opened and then closed. Each event entry has 5 fields:

- *concept:input*: the transition input : inputs or outputs that are active before the transition
- *concept:output*: the transition output : the parameters triggering the system transition
- *org:group*: the transition type (Change or Command)
- *org:resource*: the system module responsible for the transition (Plant or PLC)
- *time:timestamp*: the residence time of the system in the corresponding state

The XES file of the Sliding Door System

```
<?xml version="1.0" encoding="UTF-8" ?>
<!-- XES standard version: 1.0 -->
<!-- OpenXES library version: 1.0RC7 -->
<!-- OpenXES is available from http://www.openxes.org/ -->
<log xes.version="1.0" xes.features="nested-attributes" openxes.version="1.0RC7" xmlns="http://
    www.xes-standard.org/">
  <extension name="Lifecycle" prefix="lifecycle" uri="http://www.xes-standard.org/lifecycle.
      xesext"/>
  <extension name="Organizational" prefix="org" uri="http://www.xes-standard.org/org.xesext"/>
  <extension name="Time" prefix="time" uri="http://www.xes-standard.org/time.xesext"/>
  <extension name="Concept" prefix="concept" uri="http://www.xes-standard.org/concept.xesext"/>
```

```
<extension name="Semantic" prefix="semantic" uri="http://www.xes-standard.org/semantic.xesext"
    />
<global scope="trace">
  <string key="concept:name" value="__INVALID__"/>
</global>
<global scope="event">
  <string key="concept:input" value="__INVALID__"/>
  <string key="concept:output" value="__INVALID__"/>
  <string key="org:group" value="__INVALID__"/>
  <string key="org:resource" value="__INVALID__"/>
  <long key="time:timestamp" value="0000000000"/>
</global>
<classifier name="MXML Legacy Classifier" keys="concept:input concept:output"/>
<classifier name="Input" keys="concept:input"/>
<classifier name="Output" keys="concept:output"/>
<classifier name="Resource" keys="org:resource"/>
<string key="concept:name" value="scada.mxml"/>
<string key="lifecycle:model" value="standard"/>
<trace>
  <string key="concept:name" value="Case1"/>
  <event>
     <string key="concept:output" value="Sensor2=1"/>
     <string key="concept:input" value="Motor=0"/>
     <string key="org:group" value="CHANGE"/>
     <string key="org:resource" value="PLANT"/>
     <long key="time:timestamp" value="506000"/>
  </event>
  <event>
     <string key="concept:output" value="Motor=1"/>
     <string key="concept:input" value="Sensor2=1"/>
     <string key="org:group" value="COMMAND"/>
     <string key="org:resource" value="PLC1"/>
     <long key="time:timestamp" value="2107429000"/>
  </event>
  <event>
     <string key="concept:output" value="Sensor2=0"/>
     <string key="concept:input" value="Motor=1"/>
     <string key="org:group" value="CHANGE"/>
     <string key="org:resource" value="PLANT"/>
     <long key="time:timestamp" value="2207992000"/>
  </event>
  <event>
     <string key="concept:output" value="Sensor1=1"/>
     <string key="concept:input" value="Motor=1"/>
     <string key="org:group" value="CHANGE"/>
     <string key="org:resource" value="PLANT"/>
     <long key="time:timestamp" value="509000"/>
  </event>
  <event>
     <string key="concept:output" value="Motor=0"/>
     <string key="concept:input" value="Sensor1=1"/>
     <string key="org:group" value="COMMAND"/>
     <string key="org:resource" value="PLC1"/>
     <long key="time:timestamp" value="2107427000"/>
  </event>
  <event>
     <string key="concept:output" value="Sensor1=0"/>
     <string key="concept:input" value="Motor=0"/>
     <string key="org:group" value="CHANGE"/>
     <string key="org:resource" value="PLANT"/>
     <long key="time:timestamp" value="2207911000"/>
  </event>
  <event>
     <string key="concept:output" value="Sensor2=1"/>
     <string key="concept:input" value="Motor=0"/>
     <string key="org:group" value="CHANGE"/>
     <string key="org:resource" value="PLANT"/>
     <long key="time:timestamp" value="510000"/>
  </event>
 </trace>
</log>
```

The .XES file of the Paint Mixing system below shows the different events of one paint mixture operation with the paint code [3 2 1] :

The XES file of the Paint Mixing System

```
<?xml version="1.0" encoding="UTF-8" ?>
<!-- XES standard version: 1.0 -->
<!-- OpenXES library version: 1.0RC7 -->
<!-- OpenXES is available from http://www.openxes.org/ -->
<log xes.version="1.0" xes.features="nested-attributes" openxes.version="1.0RC7" xmlns="http://
    www.xes-standard.org/">
 <extension name="Lifecycle" prefix="lifecycle" uri="http://www.xes-standard.org/lifecycle.
     xesext"/>
 <extension name="Organizational" prefix="org" uri="http://www.xes-standard.org/org.xesext"/>
 <extension name="Time" prefix="time" uri="http://www.xes-standard.org/time.xesext"/>
 <extension name="Concept" prefix="concept" uri="http://www.xes-standard.org/concept.xesext"/>
 <extension name="Semantic" prefix="semantic" uri="http://www.xes-standard.org/semantic.xesext"
     />
 <global scope="trace">
  <string key="concept:name" value="__INVALID__"/>
 </global>
 <global scope="event">
  <string key="concept:input" value="__INVALID__"/>
  <string key="concept:output" value="__INVALID__"/>
  <string key="org:group" value="__INVALID__"/>
  <string key="org:resource" value="__INVALID__"/>
  <long key="time:timestamp" value="0000000000"/>
 </global>
 <classifier name="MXML Legacy Classifier" keys="concept:input concept:output"/>
 <classifier name="Input" keys="concept:input"/>
 <classifier name="Output" keys="concept:output"/>
 <classifier name="Resource" keys="org:resource"/>
<string key="concept:name" value="scada.mxml"/>
 <string key="lifecycle:model" value="standard"/>
 <trace>
  <string key="concept:name" value="Case1"/>
  <event>
     <string key="concept:output" value="EmptyMixt=1"/>
     <string key="concept:input" value=""/>
     <string key="org:group" value="CHANGE"/>
     <string key="org:resource" value="PLANT"/>
     <long key="time:timestamp" value="11363503000"/>
  </event>
  <event>
     <string key="concept:output" value="V1=1+V2=1+V3=1"/>
     <string key="concept:input" value="EmptyMixt=1"/>
     <string key="org:group" value="COMMAND"/>
     <string key="org:resource" value="PLC1"/>
     <long key="time:timestamp" value="22106000"/>
  </event>
  <event>
     <string key="concept:output" value="L0T1=1+L0T2=1+L0T3=1"/>
     <string key="concept:input" value="V1=1+V2=1+V3=1"/>
     <string key="org:group" value="CHANGE"/>
     <string key="org:resource" value="PLANT"/>
     <long key="time:timestamp" value="1896000"/>
  </event>
  <event>
     <string key="concept:output" value="V3=0"/>
     <string key="concept:input" value="L0T1=1+L0T2=1+L0T3=1+EmptyMixt=1"/>
     <string key="org:group" value="COMMAND"/>
     <string key="org:resource" value="PLC1"/>
     <long key="time:timestamp" value="1996033000"/>
  </event>
  <event>
     <string key="concept:output" value="L1T1=1"/>
     <string key="concept:input" value="V1=1+V2=1"/>
     <string key="org:group" value="CHANGE"/>
     <string key="org:resource" value="PLANT"/>
     <long key="time:timestamp" value="1343959000"/>
  </event>
  <event>
```

```
      <string key="concept:output" value="L1T2=1"/>
      <string key="concept:input" value="V1=1+V2=1"/>
      <string key="org:group" value="CHANGE"/>
      <string key="org:resource" value="PLANT"/>
      <long key="time:timestamp" value="1895000"/>
   </event>
   <event>
      <string key="concept:output" value="V2=0"/>
      <string key="concept:input" value="L0T1=1+L1T1=1+L0T2=1+L1T2=1+L0T3=1+EmptyMixt=1"/>
      <string key="org:group" value="COMMAND"/>
      <string key="org:resource" value="PLC1"/>
      <long key="time:timestamp" value="670080000"/>
   </event>
   <event>
      <string key="concept:output" value="L2T1=1"/>
      <string key="concept:input" value="V1=1"/>
      <string key="org:group" value="CHANGE"/>
      <string key="org:resource" value="PLANT"/>
      <long key="time:timestamp" value="1896000"/>
   </event>
   <event>
      <string key="concept:output" value="V1=0"/>
      <string key="concept:input" value="L0T1=1+L1T1=1+L2T1=1+L0T2=1+L1T2=1+L0T3=1+EmptyMixt=1"
          />
      <string key="org:group" value="COMMAND"/>
      <string key="org:resource" value="PLC1"/>
      <long key="time:timestamp" value="985966000"/>
   </event>
   <event>
      <string key="concept:output" value="V4=1+V5=1+V6=1"/>
      <string key="concept:input" value="L0T1=1+L1T1=1+L2T1=1+L0T2=1+L1T2=1+L0T3=1+EmptyMixt=1"
          />
      <string key="org:group" value="COMMAND"/>
      <string key="org:resource" value="PLC1"/>
      <long key="time:timestamp" value="18106000"/>
   </event>
   <event>
      <string key="concept:output" value="EmptyMixt=0"/>
      <string key="concept:input" value="V4=1+V5=1+V6=1"/>
      <string key="org:group" value="CHANGE"/>
      <string key="org:resource" value="PLANT"/>
      <long key="time:timestamp" value="9998000"/>
   </event>
   <event>
      <string key="concept:output" value="L2T1=0+L1T2=0"/>
      <string key="concept:input" value="V4=1+V5=1+V6=1"/>
      <string key="org:group" value="CHANGE"/>
      <string key="org:resource" value="PLANT"/>
      <long key="time:timestamp" value="9997000"/>
   </event>
   <event>
      <string key="concept:output" value="L0T3=0"/>
      <string key="concept:input" value="V4=1+V5=1+V6=1"/>
      <string key="org:group" value="CHANGE"/>
      <string key="org:resource" value="PLANT"/>
      <long key="time:timestamp" value="2005935000"/>
   </event>
   <event>
      <string key="concept:output" value="L1T1=0"/>
      <string key="concept:input" value="V4=1+V5=1+V6=1"/>
      <string key="org:group" value="CHANGE"/>
      <string key="org:resource" value="PLANT"/>
      <long key="time:timestamp" value="1342950000"/>
   </event>
   <event>
      <string key="concept:output" value="L0T2=0"/>
      <string key="concept:input" value="V4=1+V5=1+V6=1"/>
      <string key="org:group" value="CHANGE"/>
      <string key="org:resource" value="PLANT"/>
      <long key="time:timestamp" value="664974000"/>
   </event>
   <event>
      <string key="concept:output" value="L0T1=0"/>
```

```
      <string key="concept:input" value="V4=1+V5=1+V6=1"/>
      <string key="org:group" value="CHANGE"/>
      <string key="org:resource" value="PLANT"/>
      <long key="time:timestamp" value="53902000"/>
    </event>
    <event>
      <string key="concept:output" value="V4=0+V5=0+V6=0"/>
      <string key="concept:input" value=""/>
      <string key="org:group" value="COMMAND"/>
      <string key="org:resource" value="PLC1"/>
      <long key="time:timestamp" value="4999825000"/>
    </event>
    <event>
      <string key="concept:output" value="Vaciate=1"/>
      <string key="concept:input" value=""/>
      <string key="org:group" value="COMMAND"/>
      <string key="org:resource" value="PLC1"/>
      <long key="time:timestamp" value="2042028000"/>
    </event>
    <event>
      <string key="concept:output" value="EmptyMixt=1"/>
      <string key="concept:input" value="Vaciate=1"/>
      <string key="org:group" value="CHANGE"/>
      <string key="org:resource" value="PLANT"/>
      <long key="time:timestamp" value="1906000"/>
    </event>
    <event>
      <string key="concept:output" value="Vaciate=0"/>
      <string key="concept:input" value="EmptyMixt=1"/>
      <string key="org:group" value="COMMAND"/>
      <string key="org:resource" value="PLC1"/>
      <long key="time:timestamp" value="972783000"/>
    </event>
  </trace>
</log>
```

Chapter 5

Inference methodology and results

The security approach proposed here is based primarily on the incorporation of knowledge of the physical system and its control through techniques of identification of discrete event systems (DESs) [9]. These methods are close to those used in the identification of dynamical systems and involve the determination of a mathematical model to describe the behaviour of a given DES only with the observation of the temporal evolution of its digital inputs and outputs. In this chapter, we use the methodology described in Chapter 4 to extract automatically the model of both the controller and the system using the captured packets. For each event, we inferred the value of all the I/O and the residence time. The residence time is defined as the time in which the system remains at the same state (time between two consecutive system transitions). Each system will be attacked in several ways and packet capture will be performed again to reconstruct the system model. By comparing the new model to the primarily inferred models, it is possible to check whether these attacks were successfully detected or not.

5.1 Process mining techniques

Process mining techniques allow extracting information from event logs. It is used to discover models that capture the behaviour of the process as seen in the events logs. Moreover, such event logs can also be used to check if the reality as recorded in the log, conforms to an existing model. Several process modelling tools exist such as MiMo, EMiT, Little Thumb and Process Miner. Such tools are developed as prototypes to experiment with process mining algorithms. In our work, we used ProM[1] created and maintained by the process-mining group at Eindhoven University of Technology. It is a generic open-source framework for implementing process mining tools in a standard environment. The ProM framework receives, as input, logs in the XES or MXML format. Currently, this framework has plugins for process mining, analysis, monitoring and conversion. It has many different options to make the identification. It provides several mining algorithms such as ILS and Alpha. The software has a friendly interface that is divided into 3 parts. In the first one, the user selects the file or files holding the input data. Then, in a second step, the software displays a menu of options where the user can choose the algorithm by which the input file data will be processed. Finally, it shows the result of the process and generates the output file, which can be modified and saved for later use.

By performing several tests, we verified an assumption we made prior to the use of various algorithms. Processing time varies greatly, depending on the selected algorithm. Here we see

[1] http://www.processmining.org

that one of the fastest is the Alpha algorithm since it is very simple.

5.1.1 Mining algorithms

Discovering a process model from event logs requires a mining algorithm to process event logs, to extract and merge data. Several algorithms are available in the ProM tool to be used for process model discovery. We have performed several tests on the generated event logs from our networked control processes. We found that the $\alpha-$algorithm nicely infers the process model with an acceptable processing time.

The $\alpha-$algorithm takes as input an event log and produces a Petri Net. The algorithm tracks some particular precedence patterns in the event log. For example, in the case where an event a is followed by an event b, but b is never followed by a, then a causal dependency is detected between a and b. This dependency is reflected using a place connecting a to b. The algorithm relies on four ordering operations aiming to capture precedence patterns in the log :

- a $>$ b if and only if there is an event trace $\sigma = (e_1, e_2, .., e_n)$ and $i \in \{1, ..., n-1\}$ such that $e_i = a$ and $e_{i+1} = b$;
- a $\rightarrow$ b if and only if $a > b$ and $b \not> a$;
- a $\#$ b if and only if $a \not> b$ and $b \not> a$;
- a $\|$ b if and only if $a > b$ and $b > a$.

$a > b$ refers to a "directly follows" relation and $a \rightarrow b$ refers to a "causality relation" between the pair of events a and b. a $\|$ b is used when each event can follow the other (sometimes a follows b and sometimes the opposite happens). a $\#$ b states that there is no immediate precedence relation between the pair of events.

These ordering relations are then used to discover the patterns that form the process model. The typical different process behaviours and their respective event log patterns are depicted in Figure 5.1. If $a \rightarrow b$, then a sequence is created in the model. If $a \rightarrow b$ and $a \rightarrow c$ and $b\#c$ then a choice between b and c is created after a and is represented by XOR-split pattern. If $a \rightarrow c$ and $b \rightarrow c$ and $a\#b$ then a and b are considered as two concurrent events that precede c (XOR-join pattern). If $a > b$, $a > c$ and b $\|$ c then b and c can be executed in parallel just after a (AND-split pattern). If $a > c$, $b > c$ and a $\|$ b then c needs to synchronize a and b (AND-join pattern).

More details about the α-algorithm are available in [1]. Below is the definition of the algorithm [10].

The α-algorithm definition : Let L be an event log over T, the set of activities. $\alpha(L)$ is defined as follows:

1. $T_L = \{t \in T | \exists_{\sigma \in L} t \in \sigma\}$
2. $T_I = \{t \in T | \exists_{\sigma \in L} t = first(\sigma)\}$
3. $T_O = \{t \in T | \exists_{\sigma \in L} t = last(\sigma)\}$
4. $X_L = \{(A, B) | A \subseteq T_L \wedge A \neq \emptyset \wedge B \subseteq T_L \wedge B \neq \emptyset \wedge \forall_{a \in A} \forall_{b \in B} a \rightarrow_L b \wedge \forall_{a_1, a_2 \in A} a_1 \#_L a_2 \wedge \forall_{b_1, b_2 \in B} b_1 \#_L b_2\}$
5. $Y_L = \{(A, B) \in X_L | \forall_{(A', B') \in X_L} A \subseteq A' \wedge B \subseteq B' \Rightarrow (A, B) = (A', B')\}$

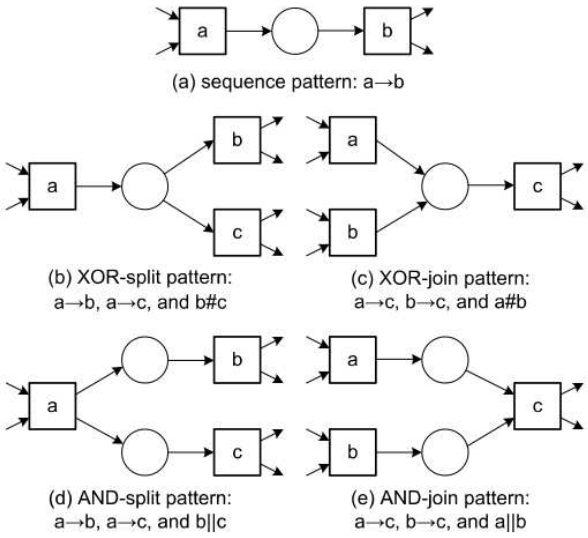


Figure 5.1: Different process behaviours and their respective event logs ordering relations [1].

6. $P_L = \{p_{(A,B)} | (A,B) \in Y_L\} \cup \{i_L, o_L\}$
7. $F_L = \{(a, p_{(A,B)}) | (A,B) \in Y_L \wedge a \in A\} \cup \{(p_{(A,B)}, b) | (A,B) \in Y_L \wedge b \in B\} \cup \{(i_L, t) | t \in T_I\} \cup \{(t, o_L) | t \in T_O\}$
8. $\alpha(L) = (P_L, T_L, F_L)$

To explain how the algorithm works, we will detail an example. Let's consider the following event log:

$$L = \{(a,b,c,e), (a,c,b,e), (a,d,e)\}$$

The following sets represent the different ordering relations found in the event log L. The footprint of L is shown in Table 5.1.

$_L = \{(a,b), (a,c), (a,d), (b,c), (b,e), (c,b), (c,e), (d,e)\}$
$\rightarrow_L = \{(a,b), (a,c), (a,d), (b,e), (c,e), (d,e)\}$
$\#_L = \{(a,a), (a,e), (b,b), (b,d), (c,c), (c,d), (d,d), (d,b), (d,c), (e,e), (e,a)\}$
$\|_L = \{(b,c), (c,b)\}$

In Step 1, we check the different events appearing in the event log (T_L). In Step 2 and Step 3, we look for the set of all events occurring initially in some trace (T_I) and the set of all events occurring at the end of some trace (T_O):

$$T_L = \{a,b,c,d,e\}$$
$$T_I = \{a\}$$
$$T_O = \{e\}$$

Then, in Step 4, we build X_L, the set of pairs (A,B) with $A \rightarrow B$, $a_i \# a_j$ and $b_i \# b_j$ $\forall a_i, a_j \in A$ $\forall b_i, b_j \in B$:

.	a	b	c	d	e
a	#	→	→	→	#
b	←	#	‖	#	→
c	←	‖	#	#	→
d	←	#	#	#	→
e	#	←	←	←	#

Table 5.1: The footprint of the event log L

$$X_L = \{(a,b),(a,c),(a,d),(b,e),(c,e),(d,e),(a,\{b,d\}),(a,\{c,d\}),(\{b,d\},e),(\{c,d\},e)\}$$

In Y_L (Step 5), we extract the maximal pairs out of X_L:

$$Y_L = \{(a,\{b,d\}),(a,\{c,d\}),(\{b,d\},e),(\{c,d\},e)\}$$

Each element of (A,B) in Y_L corresponds to a place in the Petri net connecting a transition from A to a transition from B. P_L (Step 6) holds these places along with two extra places i_L (initial place) and o_L (last place).

$$P_L = \{p_{(\{a\},\{b,d\})}, p_{(\{a\},\{c,d\})}, p_{(\{b,d\},\{e\})}, p_{(\{c,d\},\{e\})}, i_L, o_L\}$$

Next, in Step 7, the arcs linking between transitions and places are generated.

$$F_L = \{(a,p_{(\{a\},\{b,d\})}),(p_{(\{a\},\{b,d\})},b),(p_{(\{a\},\{b,d\})},d),(a,p_{(\{a\},\{c,d\})}),(p_{(\{a\},\{c,d\})},c),$$
$$(p_{(\{a\},\{c,d\})},d),(b,p_{(\{b,d\},\{e\})}),(d,p_{(\{b,d\},\{e\})}),(p_{(\{b,d\},\{e\})},e),(c,p_{(\{c,d\},\{e\})}),$$
$$(d,p_{(\{c,d\},\{e\})}),(p_{(\{c,d\},\{e\})},e),(i_L,a),(e,o_L)\}$$

After applying the α-algorithm on the event log L, we obtain a Petri net $\alpha(L) = (P_L, T_L, F_L)$ having places of P_L, transitions of T_L and arcs of F_L.

Limitation of the α-algorithm :
The α-algorithm is simple and well adapted for structured processes, but it has problems with noise, incomplete behaviour and complex loops. However, it provides a good starting algorithm to illustrate our methodology.

5.2 Analysis of of the Sliding Door System (SDS)

In this section, we detail the tests applied on the SDS whose process model is discovered using the $\alpha - algorithm$. Then, we show if we can detect malicious actions performed by an attacker or failures of system components, such as actuators or sensors.

5.2.1 Model inference

We first captured packets where the system is well functioning in the **automatic mode**. No attacks on the system were performed and no system failure was reported. In this way, the inferred model was used as a reference model to detect anomalies in next tests.

As we can see in Figure 5.2, the inferred model contains several places labelled in {1..n}. The table 5.2 makes the mapping between the labels and the places previously presented in the manually made Petri Net as described in Figure 3.8.

The obtained model represented as an Inferred Petri Net (IPN) has some proprieties that we will detail below.

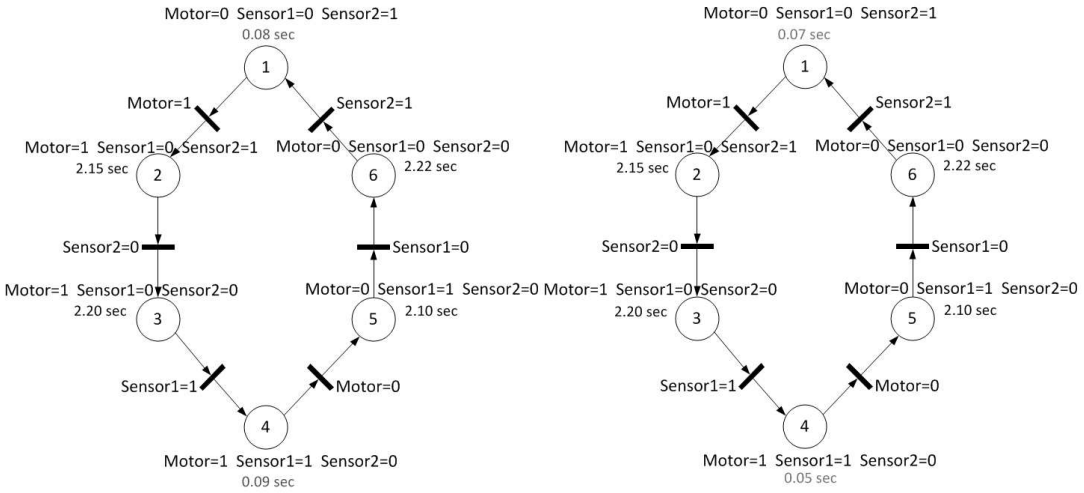


Figure 5.2: Model of SDS with 2 decimal precision timing

Number	State
1	Closed
3	Opening
4	Open
6	Closing

Table 5.2: Mapping between the state labels and their meanings

1. We observe that the IPN obtained from captured packets has several places and transitions: the circles are the places and the rectangles between the states are the transitions that denote changes between two consecutive places.

2. Each place has two associated parameters. The first parameter represents all the variables with their respective values (0 or 1). The variable Motor is the only exception among all the other variables because the values 0 and 1 do not mean that the motor is respectively off and on. They are used to represent the two rotation directions the motor can have. The second parameter is the time during which the system remains at the same state before transiting to the next state. We call it here the residence time. The timestamp linked to a place i on the Figure 5.2 is the residence time of the state i.

3. Each transition has one associated parameter which is the variables causing the transition from the previous place to the next one.

4. In the IPN, some places were not previously defined which are [2, 5]. The appearance of more places comparing to those which we defined manually is not unexpected. We have already noted this when we made the table 3.1 with every possible combination of the three binary variables of this system. In the other hand, this fact is good for the determinism of the system, because we have now less unknown combinations of inputs and outputs for the controller and it will be more difficult for an attacker to find available options to inject packets.

5. In Figure 5.2 there are two IPNs which are very similar. They have a slight difference in the values of residence times of places 1 and 4. In the first IPN, it takes the system 0.08 sec to move from place 1 to place 2 and 0.09 sec to move from state 4 to state 5. In the second IPN, these times have respectively the values 0.07 sec and 0.05 sec. This is due to the fact that the captured packets between the I/O block and the PLC have not the same capturing timestamps. The capturing timestamps depend on the load of the communication network. So if we use more precise timing, we can have many IPNs that differ in timestamps values.

6. In the IPN, the transition between [Opening and Closing] states does not appear, neither the one between [Closing and Opening] states. These transitions are active when the system is in the manual mode, and while the door is opening or closing, the operator changes, in the middle of the cycle, the actuator state.

To check the validity of what we had explained previously, we modified the script to reduce the significant decimals of packets timestamps. The Figure 5.3 depicts the obtained IPN. We observe that this IPN is the same as the manually made Petri Net for the automatic mode as shown in Figure 3.8. If we make the sum of timestamps in places between 1 and 3, we get approximately 4.3 seconds. Remember that we have designed the system to take 5 seconds to open the door and the same period of time to close it. This small difference appears because the position sensors are activated 10% before the door reaches the final states. We can verify this by observing the Figure 3.5 where the integrator block is connected to 2 other blocks [Motor ON1 and Motor ON2] that become active with 10% and 90% of the door position.

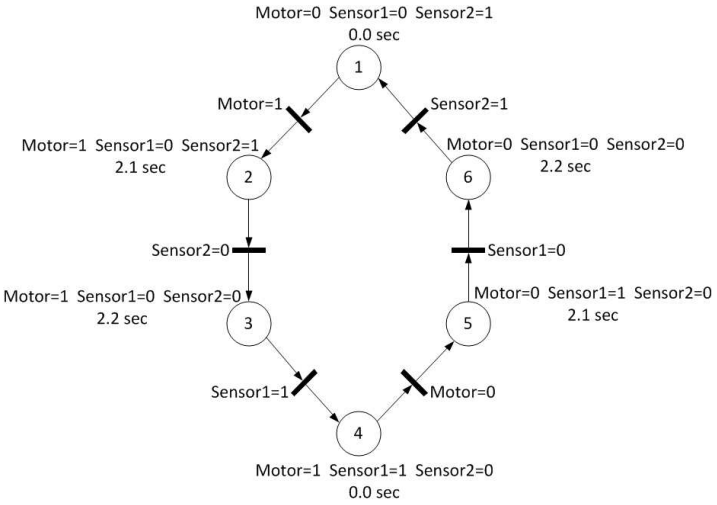


Figure 5.3: Inferred model of the SDS with 1 decimal precision timing

5.2.2 Effect of delays

Test of a slow sensor

This test will show the impact of a slow sensor on the inferred model. To emulate the effect, we introduced a pure delay of $t = 0,5sec$ on one sensor. The Figures 5.4 and 5.5 show the differences between the obtained models when sensor 1 is slower. As we expected, only the timing changes. Timestamp in place number 2 is 0.5 seconds greater than the original one and timestamp in place number 3 is 0,5 seconds shorter.

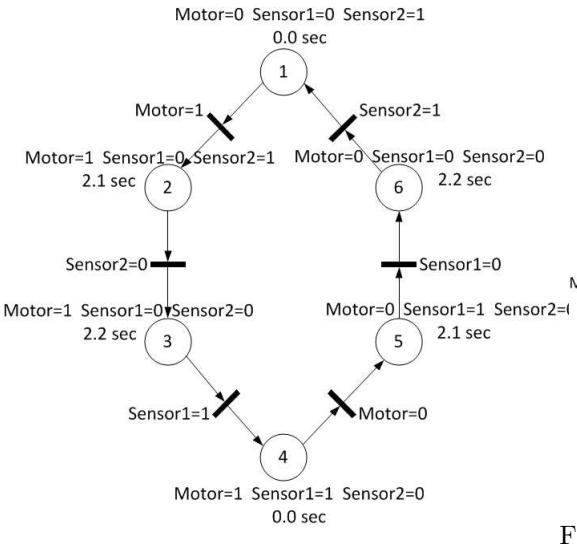


Figure 5.4: Baseline SDS inferred model

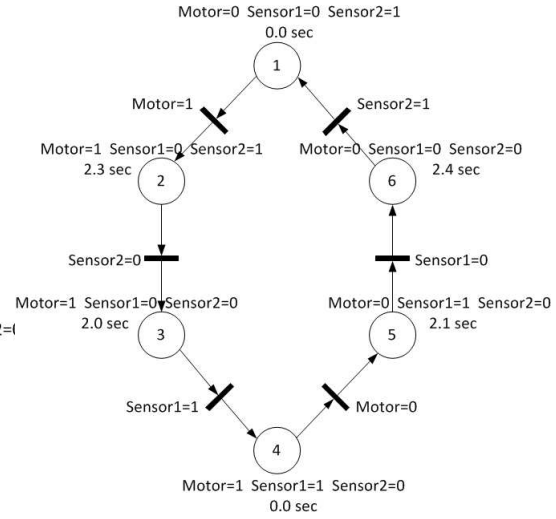


Figure 5.5: SDS inferred model with a slow sensor

Figures 5.6 and 5.7 show the differences that appear when sensor 2 is slower.

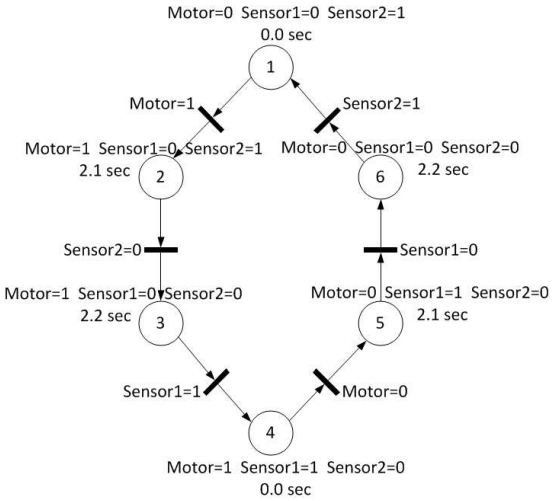


Figure 5.6: Baseline SDS inferred model

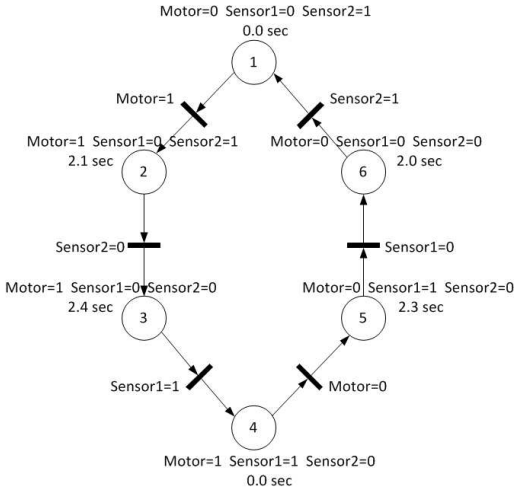


Figure 5.7: SDS inferred model with a slow sensor 2

In both cases, we observed that only the timing changes in the inferred models with sensor delay compared to the baseline model.

Test of a slow actuator

In this test, we added a pure delay to an actuator. Figures 5.8 and 5.9 show the inferred model before and after adding the delay to the motor.

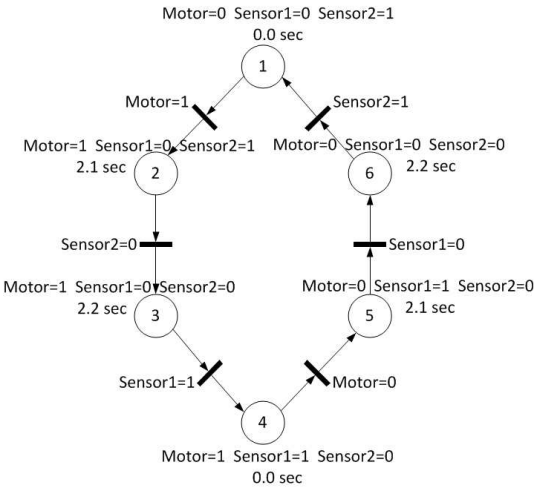


Figure 5.8: Baseline inferred model before adding delay to the motor

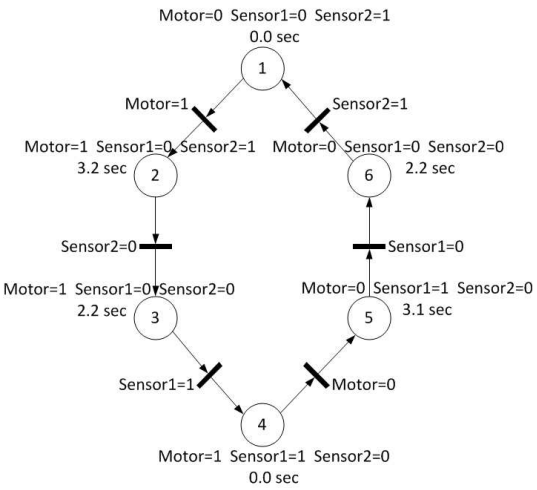


Figure 5.9: Inferred model after adding delay to the motor

5.3 Data Injection

The last test made on the SDS is data injection. In this test, we wait until the door reaches the *Opening* state (place number 3), and before it gets completely *Open*, we send one packet to the PLC with the value of $Sensor1 = 1$ which means that the door is open. The door starts then to *Close* again, but never really reaches the *Open* state.

This packet injection allows an attacker to keep the door always closed. The resulting inferred model is depicted in Figure 5.10. Dashed lines represent the normal working mode.

The injection starts in place 3 (*Opening*). The system changes its normal behaviour and instead of reaching place 4 (*Open*), it jumps immediately to place 7 (*Close*).

The place 8 appears because during our packet injection, the I/O block continues sending data to the PLC. As it takes a few milliseconds to the control to evolve, we observe this delay in the timestamp associated to place 7.

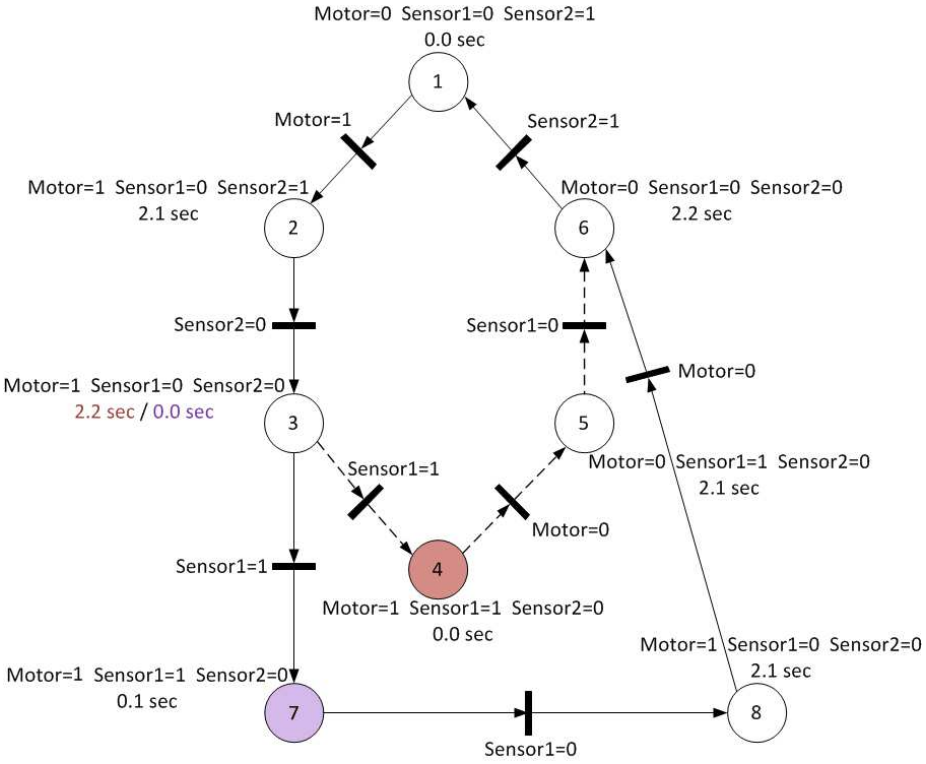


Figure 5.10: Inferred model of SDS with a data injection of value 1 for sensor 1

5.4 Analysis of the Paint Mixing System (PMS)

In this section, we detail the tests that we have made on the paint mixing system. Here we will see if based on the inferred model we will be able to detect a system malfunction or an attack performed on the Control. We will also modify some internal system parameters, as the density of paint colors and inject malicious packets. After that we will rebuild the model to analyse their impact.

5.4.1 Model inference

In the same way as in the Sliding Door System, the first data capture was performed on a stable system without any attacks or failures. In this way, the Inferred Petri Net (IPN) can be used as a reference model to detect anomalies in next stages. We know in advance that the PMS Petri Net will be lager than the SDS model. In the next figures, for each place, we will specify only the I/O variables with the value 1.

We have to note that this system will be tested with the paint color code [3 1 2]. As we explained in Section 3.4, we are able to create 64 different colors and each one has its own Petri Net. For example, the code [3 1 2] is interpreted as follows:

- level 3 of red paint,
- level 1 of green paint,
- level 2 of blue paint.

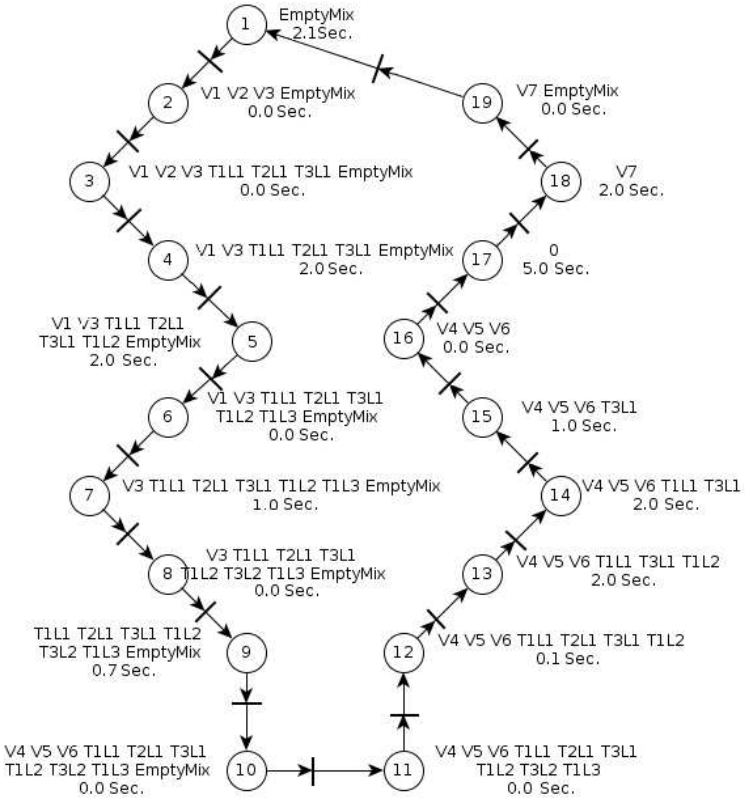


Figure 5.11: Inferred model of PMS with 1 decimal precision timing

In the inferred model of the PMS shown in Figure 5.11, we observe a cyclic behaviour. This is due to the fact that we have used a capture file with more that one mixing paint cycle. This means that the system makes a cycle and when it finishes, it restarts again. This can be seen in the model with the transition from place 19 to place 1.

5.4.2 Test of paint density change

In this test, the goal is to detect the impact of changing the density of one color on the inferred model. The values we selected for testing the system are synthetic and do not represent real values.

The current density values of paints are:

- Red: 2 g/cm^3
- Green: 3.33 g/cm^3
- Blue: 5 g/cm^3

We modified the Blue color density from 5 g/cm^3 to 1.42 g/cm^3. The inferred model is shown in Figure 5.12.

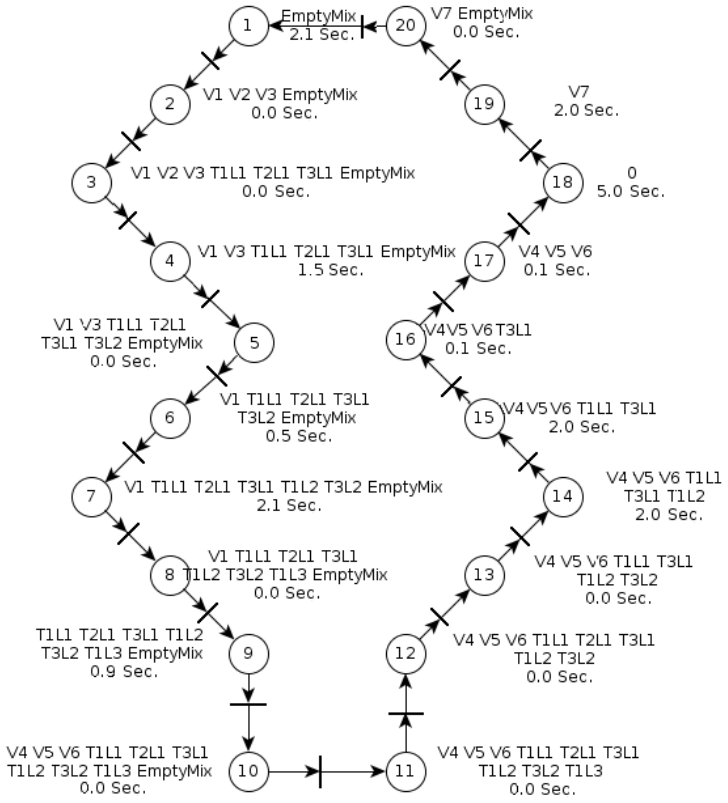


Figure 5.12: Inferred model of PMS with a Blue color density of 1.42 g/cm3

We observed several changes in this inferred model:

1. We see that this inferred model has one more place. This happens because the order of events has changed and between places 11 and 12 there only is 1 change, while in the original inferred model there are 2 changes.

2. Several places have been disordered and many variables in places have changed. An example of this is place number 5. In the original inferred model, the tank number 1 reaches its level 2 before the the tank number 3. However, in this case, its reaches the level 2 after it.

3. An other important observation is that many timestamps values have changed.

We were expecting some behaviour modification because we assumed that modifying the density of the blue color to a lower value would make the paint filling in tank 3 faster than filling tank 1.

5.4.3 Data Injection

The last test we made on the PMS is data injection. We waited until the system finished the 5 seconds for mixing and opening the valve V7 in order to empty the mixer, and 1 second after the valve was opened, we sent to the PLC a packet with the data **EmptyMix=1**. Based on this injected packet, the controller would assume that the mixer tank was empty and close the valve V7 while, in reality, there was still paint in the mixer. In the place 18 of the baseline inferred model, the valve V7 remains open for 2 seconds. So if the system control is subject to attacks to close the valve after 1 second, we can suppose that half of the paint will stay in the mixer. Figure 5.13 shows the inferred model of the paint mixing system after packet injection. We observe that the place 18 only remains 1 second before the system reaches the state number 1. However, in the baseline model, this value was of 2 seconds as depicted in Figure 5.11.

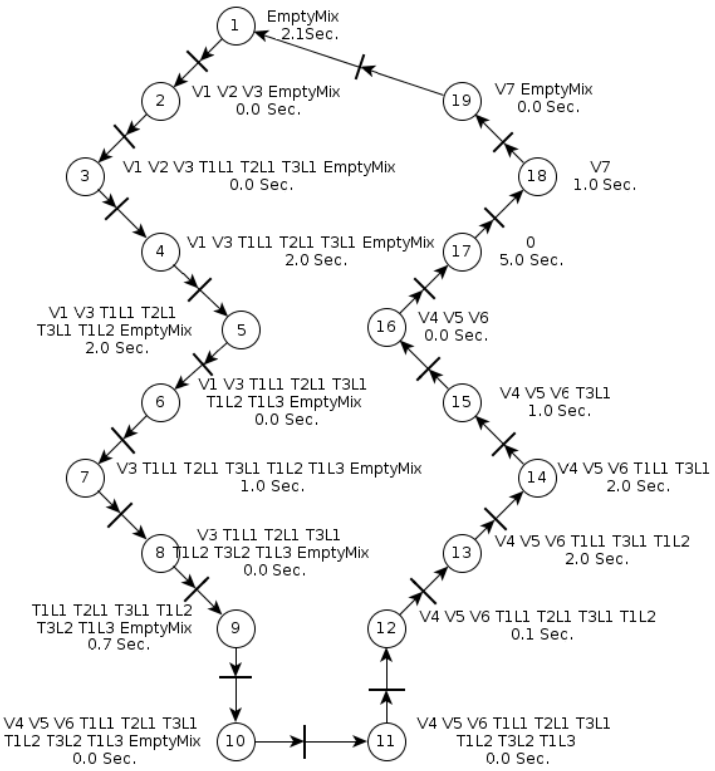


Figure 5.13: Inferred model of PMS with packet injection targeting valve V7

Chapter 6

Conclusion and future work

In this work, we have developed an experimental testbed to extract and analyse the behaviour of several networked discrete process control systems relying on the PROFINET protocol. Our methodology relies on the incorporation of knowledge of the physical system and its control within an inferred model by capturing and decoding system data available in exchanged PROFINET messages. We designed two experimental process systems and their controls. The first system is an automatic sliding door and the second is a paint mixing system. We performed several tests over them, including the inference of their models based on the exchanged PROFINET packets between the controller and I/O blocks, the modification of their internal parameters to emulate the failure of their components by introducing delays, and also through packets injection to modify their behaviours. We obtained for each testing case an inferred model of the running process. We used these models to detect the different elaborated attack and malfunctioning scenarios. During this work, we acquired more knowledge about the PROFINET protocol, the role of the different frames and the importance of timing precision in packet capture and delivery. Time constraints imposed by the protocol make inadequate packet capture and injection through a computer because it does not meet time requirements. Therefore, it is important to rather use a dedicated device that works faster and allows higher timing precision.

In future work, we plan to use NetFPGA devices to perform data capture, modification and injection in very short time intervals, in the order of nanoseconds. The idea is to build a network bridge able to modify some packets in order to move the system into different states. In this way, we can test the control part in real time and make the system evolving into states that are not previously defined.

Bibliography

[1] W.M.P. van der Aalst. *Process Mining: Discovery, Conformance and Enhancement of Business Processes*. Springer Publishing Company, Incorporated, 1st edition, 2011.

[2] S. Keith, F. Joe, and S. Karen. Guide to industrial control systems (ics) security. 2011.

[3] M. Xue, S. Roy, Y. Wan, and S.K. Das. Security and discoverability of spread dynamics in cyber-physical networks. *IEEE Transactions on Parallel and Distributed Systems*, pages 1694–1707, 2012.

[4] A.A. Cárdenas, S. Amin, Z.S. Lin, Y.L Huang, C.Y. Huangand, and S. Sastry. Attacks against process control systems: risk assessment, detection, and response. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, New York, NY, USA, 2011.

[5] T.A. Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, LICS '96, 1996.

[6] J. Feld. Profinet - scalable factory communication for all applications. In *IEEE International Workshop on Factory Communication Systems*, Vienna, Austria, 2004.

[7] Profinet io another innovative distributed automation solution, 2008. Real Time Automation.

[8] R. Ierusalimschy, L.H. De Figueiredo, and W. Celes. The evolution of lua. In *Proceedings of the 3rd ACM SIGPLAN conference on History of programming language*, (HOPL-III) ACM SIGPLAN '07, San Diego, California, USA, 2007.

[9] A.P. Estrada-Vargas. A comparative analysis of recent identification approaches for discrete-event systems. *Mathematical Problems in Engineering*, 2010.

[10] W.M.P. van der Aalst, A.J.M.M. Weijters, and L. Maruster. Workflow mining: Discovering process models from event logs. In *IEEE Transactions on Knowledge and Data Engineering*, pages 1128 – 1142, 2004.

Appendix A

Definitions

System : The concept of system can be defined in several ways. One possible definition is: a system is a set of interacting or interdependent components forming an integrated whole or a set of elements often called components and relationships which are different from relationships of the set or its elements to other elements or sets.

Control : It is a preventive and corrective mechanism adopted by the administration of an agency or entity that allows timely detection and correction of deviations, inefficiencies and inconsistencies during the design, implementation, execution and evaluation of the actions, in order to procure the compliance with regulations governing them, and the strategies, policies, objectives, targets and resource allocation.

Real time Simulation : The main difference between a Real Time simulation and other simulations is that they must meet certain time constraints regarding the behaviour of the system under study. In general a rule widely applied in these simulations is that they must comply with a response time **Tr**, defined as the period of time between a data input and the output response. One of the conditions that can be put to talk about real-time is to limit this parameter to a maximum value, or even require that this response time is constant along the simulation.

Petri Net : A Petri net (also known as a place/transition net or P/T net) is one of several mathematical modelling languages for the description of distributed systems. A Petri net is a directed bipartite graph, in which the nodes represent transitions (i.e. events that may occur, signified by bars) and places (i.e. conditions, represented by circles). The directed arcs describe which places are pre- and/or postconditions for which transitions (represented by arrows) occurs. Some sources state that Petri nets were invented in August 1939 by Carl Adam Petri — at the age of 13 — for the purpose of describing chemical processes.

Appendix B

PROFINET Terms

The full list of PROFINET terms is available at [7].

Term	**Definition**
Channel	A single I/O point. A Channel can be discrete or analogue.
Subslot	A group of one or more channels. Subslots can be real or virtual.
Slot	A group of one or more Subslots. Slots can be real or virtual.
Module	Modules are user defined components that plug into slots. Modules can be real or virtual.
Submodule	A component of a module that is plugged into a subslot. A submodule is real or virtual.
FrameId	The two byte field in the Ethernet frame which defines the type of PROFINET IO message.
Real Time (RT)	The Real Time PROFINET IO Channel. I/O and Alarm Data are transferred over the RT Channel.
Provider Status	The Status an I/O device provides to an IO Controller with the data transferred to the Controller.
Consumer Status	The Status an I/O device provides to an IO Controller for the data it consumes from IO Controller.

Appendix C

Data conversion program

We implemented a Lua script to intercept the network packets captured with Wireshark and filter the PROFINET frames between the PLC and the I/O block. The filtered frames are stored in a Syslog database. Below is the Lua script code.

```
-------------------------------------------------------------------------------
-- Author      : Ayoub SOURY
-- Modified by : Bilel SAADALLAH <bilel.saadallah[at]inria.fr>
-- Copyright (C) 2013 INRIA.
-------------------------------------------------------------------------------

do
        client = require("socket")
        client = socket.connect("127.0.0.1",25000)

        packets = 0;
        ------------------------------------------------
        tab = {}
        tab[1] = {};
        ------------------------
        --INPUT--
        tab[1] = {"i8","i4","i7","i3","i6","i2","Sensor1","Sensor2","i16","i12","i15","i11","i14","i10","i13","i9","o8","o4","o7","o3","o6","o2",
        "o5","Motor","o16","o12","o15","o11","o14","o10","o13","o9"}
        ------------------------
        n = 1;

        ------------------------------------------------------------------------------------------------------------------------------------------
        ---- Listener for PNIO-CM packets ------------------------------------------------------------------------------------------------------
        ------------------------------------------------------------------------------------------------------------------------------------------
        local function init_cm_listener()
                local tap = Listener.new("frame","pn_io.opnum==0 and eth.dst==00:1b:1b:19:7a:51 and eth.src==00:1b:1b:16:25:1d"); --Filter PNIO-CM
                packets between the PLC and the Digital I/O module
                --local Maxlen_extractor = Field.new("pn_io.args_max");                 --Max length
                local IocrType_extractor = Field.new("pn_io.iocr_type");                --Input/Output Block
                local LT_extractor = Field.new("pn_io.lt");                             --Type of protocol (0x8892 => PN_RT)
                local Number_IODataObjects_extractor = Field.new("pn_io.number_of_io_data_objects")     --Number of data objects
                local Number_IOCS_extractor = Field.new("pn_io.number_of_iocs");                --Number of consumer status
                local Opnum_extractor = Field.new("pn_io.opnum");                               --Operation number
                local FrameID_extractor = Field.new("pn_io.frame_id");                          --Frame Id
                local DataLength_extractor = Field.new("pn_io.data_length");
                local SlotNumber_extractor = Field.new("pn_io.slot_nr");
                local DataObjects_frame_offset_extractor = Field.new("pn_io.io_data_object_frame_offset");
                local ConsumerStatus_frame_offset_extractor = Field.new ("pn_io.iocs_frame_offset");
                local ts_extractor = Field.new("frame.time");

                function tap.reset()
                        packets = 0;
                end--tap.reset

                function tap.packet(pinfo,tvb,ip)
                        local opnum = Opnum_extractor();
                        --local max = Maxlen_extractor();
                        local iotype = {IocrType_extractor()};
                        local frameid = {FrameID_extractor()};
                        local datalength = {DataLength_extractor()};
                        local dataobjects = {Number_IODataObjects_extractor()};
                        local consumerstatus = {Number_IOCS_extractor()};
                        local lt = LT_extractor();
                        local c = string.format("0x%.2X",lt.value);
                        local ts = ts_extractor();
                        des = tostring(pinfo.dst);
                        src = tostring(pinfo.src);
                        local inputSlots = "";
                        local inputOffs = "";
                        local outputSlots = "";
                        local outputOffs = "";
```

```
                if opnum.value==0 then
                        op="Connect("..opnum.value..")";                        --Checks if operation = Connect(0)
                        print(op);
                end;-- if opnum.value==0 then
                for t in pairs(iotype) do                                       --INPUT OR OUTPUT
                        if (t == 1) and (c == "0x8892") then                    --t==1 => Input, c=0x8892 => PN_RT
                                --frameidInput = frameid[t].value;
                                frameidInput = 32865;
                                datalengthInput = datalength[t];
                                Inputdataobjects = dataobjects[t];              --Number of INPUT Data Objects
                                Inputconsumerstatus = consumerstatus[t];        --Number of INPUT Consumer Status
                        elseif  (t == 2) and (c == "0x8892") then               --t=2 => Output, c=0x8892 => PN_RT
                                --frameidOutput = frameid[t].value;
                                frameidOutput = 32768;
                                datalengthOutput = datalength[t];
                                Outputdataobjects = dataobjects[t];             --Number of OUTPUT Data Objects
                                Outputconsumerstatus = consumerstatus[t];       --Number of OUTPUT Consumer Status
                        end;--if(t == 1) and (c == "0x8892") then
                end;--for t in pairs(iotype) do

                -- Complete the first line
                Input=Inputdataobjects.value+Inputconsumerstatus.value;         --Number of Input variables
                slotnr = {SlotNumber_extractor()};
                dataobjectsoffset = {DataObjects_frame_offset_extractor()};
                consumerstatusoffset = {ConsumerStatus_frame_offset_extractor()};
                slotInput = {};         --A Two Dimensional Array for Input Slots/Offsets
                slotInput[1] = {};      --Input Slots (Slots of Data Objects followed by Slots of Consumer Status)
                slotInput[2] = {};      --Input Offsets of Data Objects/Consumer Status
                slotOutput = {};        --A Two Dimensional Array for Output Slots/Offsets
                slotOutput[1] = {};     --Output Slots (Slots of Data Objects followed by Slots of Consumer Status)
                slotOutput[2] = {};     --Output Offsets of Data Objects/Consumer Status
                if Inputdataobjects ~= 0 then
                        for i = 1, Inputdataobjects.value do
                                slotInput[1][i] = slotnr[i].value;
                                slotInput[2][i] = dataobjectsoffset[i].value;
                                inputSlots = inputSlots..tostring(slotInput[1][i])..","
                                inputOffs = inputOffs..tostring(slotInput[2][i])..","
                        end;--for i =1,Inputdataobjects.value do
                        if Inputconsumerstatus ~=0 then
                                for i = Inputdataobjects.value+1, Inputdataobjects.value+Inputconsumerstatus.value do
                                        slotInput[1][i] = slotnr[i].value;
                                        slotInput[2][i] = consumerstatusoffset[i-Inputdataobjects.value].value;

                                end;--for i=Inputdataobjects.value+1,Inputdataobjects.value+Inputconsumerstatus.value do
                        else
                        end;-- if Inputconsumerstatus ~= 0 then
                else
                end;--if Inputdataobjects ~= 0 then
                Output=Outputdataobjects.value+Outputconsumerstatus.value;              --Number of Output variables
                if Outputdataobjects ~=0 then
                        for i = 1+Inputdataobjects.value,Inputdataobjects.value+Outputdataobjects.value do
                                slotOutput[1][i-Inputdataobjects.value] = slotnr[i].value;
                                slotOutput[2][i-Inputdataobjects.value] = dataobjectsoffset[i].value;
                                outputSlots = outputSlots..tostring(slotOutput[1][i-Inputdataobjects.value])..","
                                outputOffs = outputOffs..tostring(slotOutput[2][i-Inputdataobjects.value])..","
                        end;-- for i==..
                        if Outputconsumerstatus ~= 0 then
                                for i = 1, Outputconsumerstatus.value do
                                        slotOutput[1][i+Outputdataobjects.value] = slotnr[i].value;
                slotOutput[2][i+Outputdataobjects.value] = consumerstatusoffset[i+Inputconsumerstatus.value].value;
                                end;--for output..
                        else
                        end;--outputconsumer..
                else
                end;--outputdataobjects..
                packets = packets + 1;

                ---------------------------------
                client:send("\n<13>DoorSystem: "..tostring(ts).." : CM : "..src.." : "..des.." : "..Inputdataobjects.value.." :
                 "..inputSlots.." : "..inputOffs.." : "..Outputdataobjects.value.." : "..outputSlots.." : "..outputOffs)
                client:settimeout(0.0000032)
                ---------------------------------

        end--tap.packet

        function tap.draw()
        --io.close();
        end-- tap.draw

end --listener


-------------------------------------------------------------------------------------------------------------------------------------
---- Listener for PN-RT packets -----------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------------
local function init_rt_listener()
        local packets2 = 0;
        local tap = Listener.new("frame","pn_rt and !(eth.dst[0] & 1)");
        --local CycleCounter_extractor = Field.new("pn_rt.cycle_counter");
        local UndecodedData_extractor = Field.new("pn.undecoded");
        local FrameIDRT_extractor = Field.new("pn_rt.frame_id");
        --local FrameN_extractor = Field.new("frame.number");
        local ts_extractor = Field.new("frame.time");

        function tap.reset()
                packets2 = 0;
        end
```

```
		und1 = {};
		und2 = {};
		out = {"00","00"};
		inp = {};
		lastts = 0;                             --Last timestamp
		function tap.packet(pinfo,tvb,ip)
			local frameidrt = FrameIDRT_extractor();
			--local CycleCounter = CycleCounter_extractor();
			--local timestamp = string.format("%.2f",pinfo.abs_ts);
			local ts = ts_extractor();
			local Undecoded = UndecodedData_extractor();
			local offset = Undecoded.offset;
			--local framenumber = FrameN_extractor();

			des = tostring(pinfo.dst);
			src = tostring(pinfo.src);

			-- INPUT
		if ((frameidrt.value == frameidInput) and (des == "Siemens_16:25:1d" and src =="Siemens_19:7a:51")) then
		-- Handle Input Packets

				---------------------------------
				client:send("\n<14>DoorSystem: "..tostring(ts).." : Input : "..src.." : "..des.." :
				 "..tostring(tvb(offset,Undecoded.len-20))..tostring(tvb(offset+20,Undecoded.len-20)))
				client:settimeout(0.0000032)
				---------------------------------

			-------------------------------------------------------------------------------------------------------------------
			--OUTPUT
			elseif (frameidrt.value == frameidOutput) and (des == "Siemens_19:7a:51" and src == "Siemens_16:25:1d") then
			-- Handle Output Packets

				---------------------------------
				client:send("\n<14>DoorSystem: "..tostring(ts).." : Output : "..src.." : "..des.." :
				 "..tostring(tvb(offset,Undecoded.len-20))..tostring(tvb(offset+20,Undecoded.len-20)))
				client:settimeout(0.0000032)
				---------------------------------

			end --if (frameidrt.value == frameidOutput) then
			-------------------------------------------------------------------------------------------------------------------

			packets2 = packets2 + 1;
		end
		function tap.draw()
			print("PN_RT Packets",packets2);
		end
	end

	init_rt_listener();
	init_cm_listener();
	-----------------------------------------------------------------------------------------------------------------------------------
	-- make sure LuaSocket is loaded
	local io = require("io")
	local base = _G
	local os = require("os")
	local math = require("math")
	local string = require("string")
	local socket = require("socket")
	local ltn12 = require("ltn12")

end --do
```

Appendix D

Paint system HMI program

The Paint Mixing System HMI runs a VB program to allow the operator to control the system through the screen. Below is the program code source.

VB Script

```
Sub Paint()
Select Case SmartTags("Code1")
  Case 0
    SmartTags("SP0T1")=0
    SmartTags("SP1T1")=0
    SmartTags("SP2T1")=0
  Case 1
    SmartTags("SP0T1")=1
    SmartTags("SP1T1")=0
    SmartTags("SP2T1")=0
  Case 2
    SmartTags("SP0T1")=0
    SmartTags("SP1T1")=1
    SmartTags("SP2T1")=0
  Case 3
    SmartTags("SP0T1")=0
    SmartTags("SP1T1")=0
    SmartTags("SP2T1")=1
End Select

Select Case SmartTags("Code2")
  Case 0
    SmartTags("SP0T2")=0
    SmartTags("SP1T2")=0
    SmartTags("SP2T2")=0
  Case 1
    SmartTags("SP0T2")=1
    SmartTags("SP1T2")=0
    SmartTags("SP2T2")=0
  Case 2
```

```
    SmartTags("SP0T2")=0
    SmartTags("SP1T2")=1
    SmartTags("SP2T2")=0
  Case 3
    SmartTags("SP0T2")=0
    SmartTags("SP1T2")=0
    SmartTags("SP2T2")=1
End Select

Select Case SmartTags("Code3")
  Case 0
    SmartTags("SP0T3")=0
    SmartTags("SP1T3")=0
    SmartTags("SP2T3")=0
  Case 1
    SmartTags("SP0T3")=1
    SmartTags("SP1T3")=0
    SmartTags("SP2T3")=0
  Case 2
    SmartTags("SP0T3")=0
    SmartTags("SP1T3")=1
    SmartTags("SP2T3")=0
  Case 3
    SmartTags("SP0T3")=0
    SmartTags("SP1T3")=0
    SmartTags("SP2T3")=1
End Select

Do Until (SmartTags("Valve1")<>0 Or SmartTags("Valve2")<>0 Or
    SmartTags("Valve3")<>0)
Loop
Do Until (SmartTags("Valve1")=0 And SmartTags("Valve2")=0 And
    SmartTags("Valve3")=0)
Loop
Do While (SmartTags("Level0T1")<>0 Or SmartTags("Level0T2")<>0 Or
    SmartTags("Level0T3")<>0) SmartTags("MValve456")=1
Loop
SmartTags("MValve456")=0
SmartTags("Timer1ON")
Do While SmartTags("EmptyMixer")=0
Loop
Loop
End Sub
```

Appendix E

Sliding door ladder program

Here is the sliding door program written in Ladder language. We can see that the program uses the resources I0.0 for Sensor 2, I0.1 for Sensor 1 and Q0.0 for Motor.

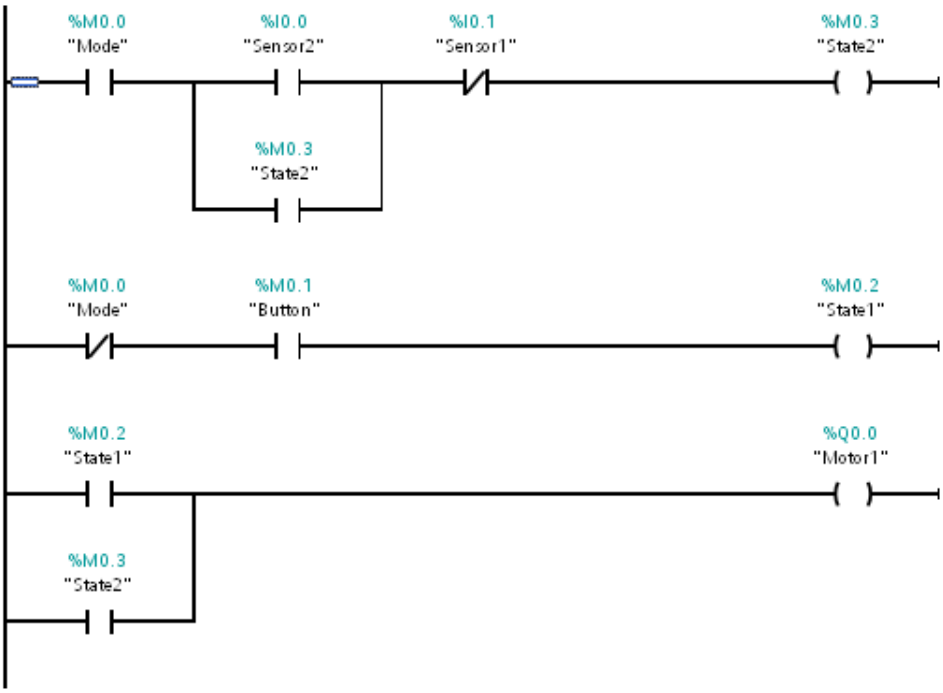


Figure E.1: Sliding door ladder program

Appendix F

Experiment user's guide

F.1 Testbed host machines

To make an experiment using the created testbed, we use 4 host machines. The following paragraph gives details about these machines, their roles and software/packages dependencies.

Machine A : This machine was used for the design of the physical processes with Matlab. We use it as well to compile the designed model and to build it into the xPC Target (machine B) with Simulink.

Perquisites:

- MATLAB (7.14)
- Simulink (7.9)
- xPC Target (5.2)

Machine B : It is used for real time simulation of physical processes created with Matlab. This machine boots on a Matlab LiveCD created with xPC Target.

Machine C : This machine is used for programming the PLC, I/O block and SCADA interfaces.

Perquisite:

- TIA Portal (11)

Machine D : We use this machine for network sniffing and packets capture. It is also used for analysing the acquired data and generating the inferred system model.

Perquisites:

- Wireshark (1.8.2)
- Lua5.2 package (5.2)

F.2 How to launch an experiment

The following section explains step by step the different instructions to launch an experiment using the Sliding Door system project. Before starting an experiment, make sure that the platform and all host machines are switched on.

On machine A : This machine was used to create the physical processes under Matlab. To use a process, we need to load it into the xPC Target machine (machine B) using Matlab-Simulink on Machine A.

- Open Matlab, then choose the menu File->Open to open the project. Browse the Sliding Door System folder and choose the file **DoorXPC.mdl**. A new Simulink window will open showing the designed model of the system.
- In the Simulink window, choose the menu Tools->Code generation->Build the model (Figure F.1). This will compile the model files and load the physical process into Machine B for simulation.

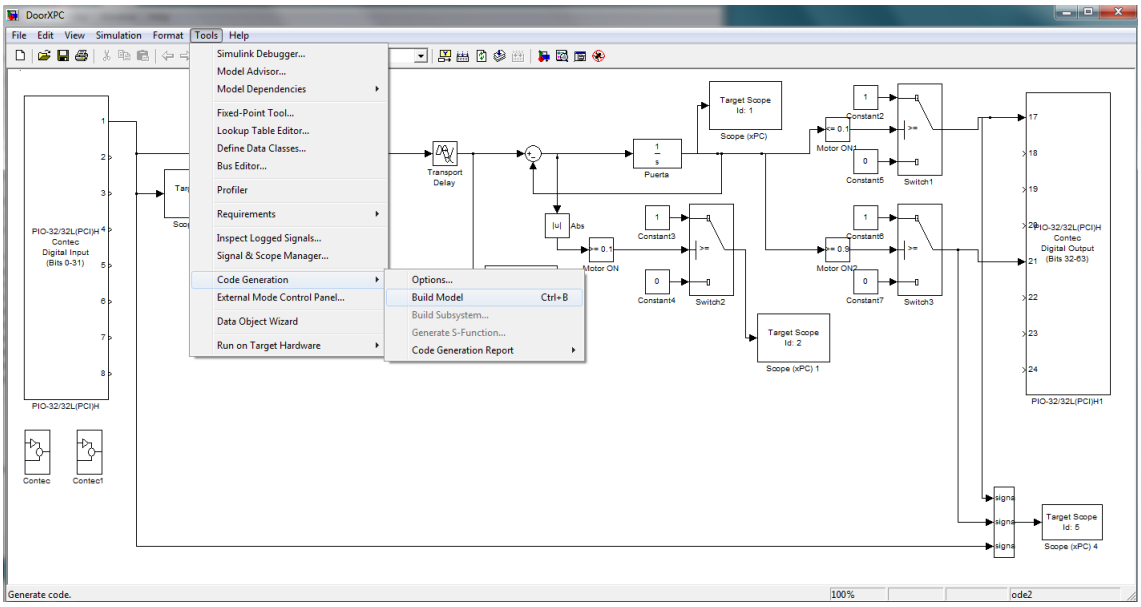


Figure F.1: Building the model with Simulink

On machine B : This machine is the Target PC. It runs xPC Target using a Matlab LiveCD. It simulates in real time the physical process uploaded from Machine A. Figure F.2 is a screenshot made before the process is loaded. Figure F.3 shows the system status after the simulation physical process is loaded. To start and stop the process simulation, you can use the command prompt. To show the command line, press *Shift + c*. Write *start* to start the process or *stop* to stop it then press the Enter key.

- Type *start* and press *Enter* to run the real time simulation of the Sliding Door system.

Note that the simulation time may reach an overflow state. This is an expected behaviour; just restart the process with the command *start*.

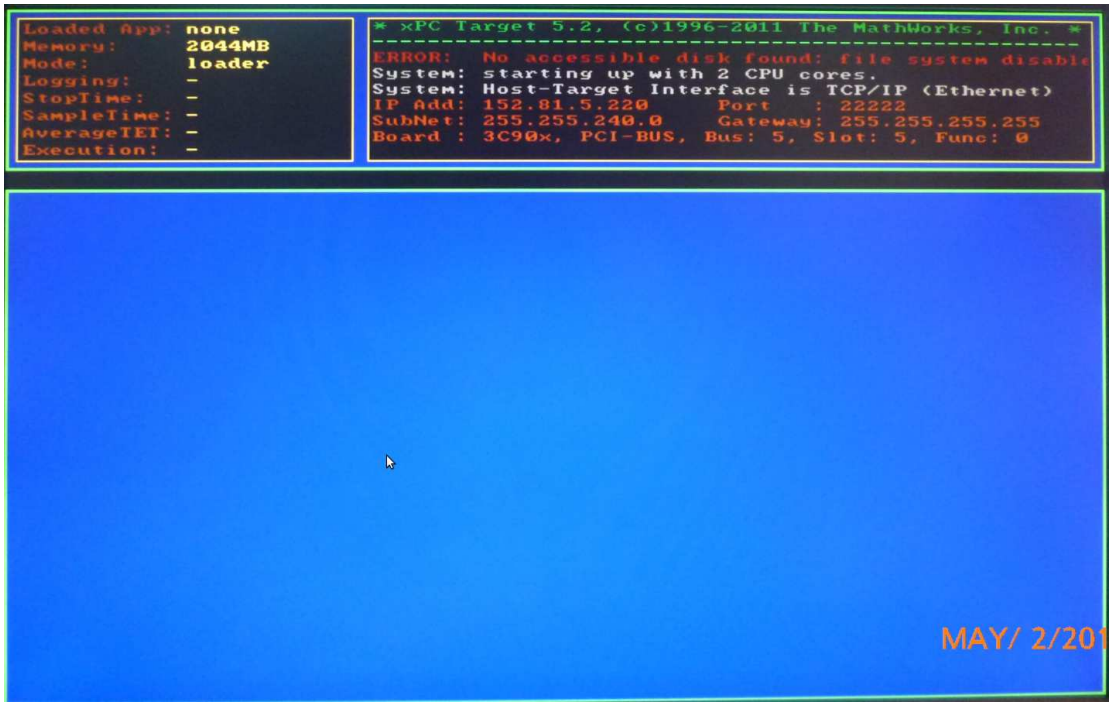


Figure F.2: Target PC before process upload

On machine C : We will use this machine to load the drivers to the PLC and the HMI.

- Open the TIA Portal. In the TIA interface (Figure F.4), select the menu *Start*, click on *Open existing project* and choose the file **Project1.ap11** from the corresponding Sliding Door System project.
- Choose the menu *Devices & networks* and click on *Show all devices*. Two devices will appear on the screen : one for the PLC and another one for the HMI. Double-click on the PLC driver icon to access the programming interface.
- In the programming interface, there is the project tree at the left. Click on *PLC_1[CPU 315F-2 PN/DP]* then double-click on *Device configuration*. Go to menu *Online* and choose *Download to device* or you can use the corresponding button in the toolbar (Figure F.5).
- A Load Preview window will open (Figure F.6). Click on the *Load* button. This will compile the driver and download it to the PLC. When the loading is successfully done, click on *Finish*. Then the PLC is configured and ready to run.

  Note here that in the previous step, we assumed that the PLC run/stop button was set at stop and the corresponding led light would be Orange. Once the driver's loading finishes, switch the button to Run (Figure F.7). The led light will turn to Green if everything goes fine. The PLC starts executing its program.

  If the run/stop button is set at Run before or while you are loading the driver, then a new window will appear when you click on the *Load* button (Figure F.8). You need to check the *Start all* checkbox then click on the *Finish* button.

Figure F.3: Target PC before process upload

- The next step is to configure the HMI. In the project tree, click on *HMI_ 1[TP700 Comfort]* then double-click on *Device configuration*. Go to menu *Online* and choose *Download to device* or you can use the corresponding button in the toolbar (Figure F.9).

- A Load Preview window will open (Figure F.10). Check the *Overwrite all* checkbox then click on the *Load* button. The Sliding Door interface program will be loaded into the HMI device. After this, click on the *Finish* button. To check that the operation was successfully done, take a look at the HMI screen; you should have the Sliding Door interface on screen (Figure F.11).

On machine D :

- Launch wireshark to start the network capture of PROFINET frames. Then save the capture into a pcap file.

Important Note :
The PNIO-CM packets are generated **exactly** when the driver is loaded into the PLC via the TIA Portal. This means that you have to launch the network capture with wireshark **before** uploading the PLC driver. These packets are necessary to detect data offsets in the following captured PN-IO data frames and are subsequently needed for parsing and analysing activities.

- Before parsing the captured packets, we recommend to empty the *logs* table in syslog database to avoid any confusion between the different experiments. The parsing script is not capable of detecting the limit between logs of two consecutive experiments. To empty the table, log in in mysql-admin or mysql workbench and apply the following SQL query :

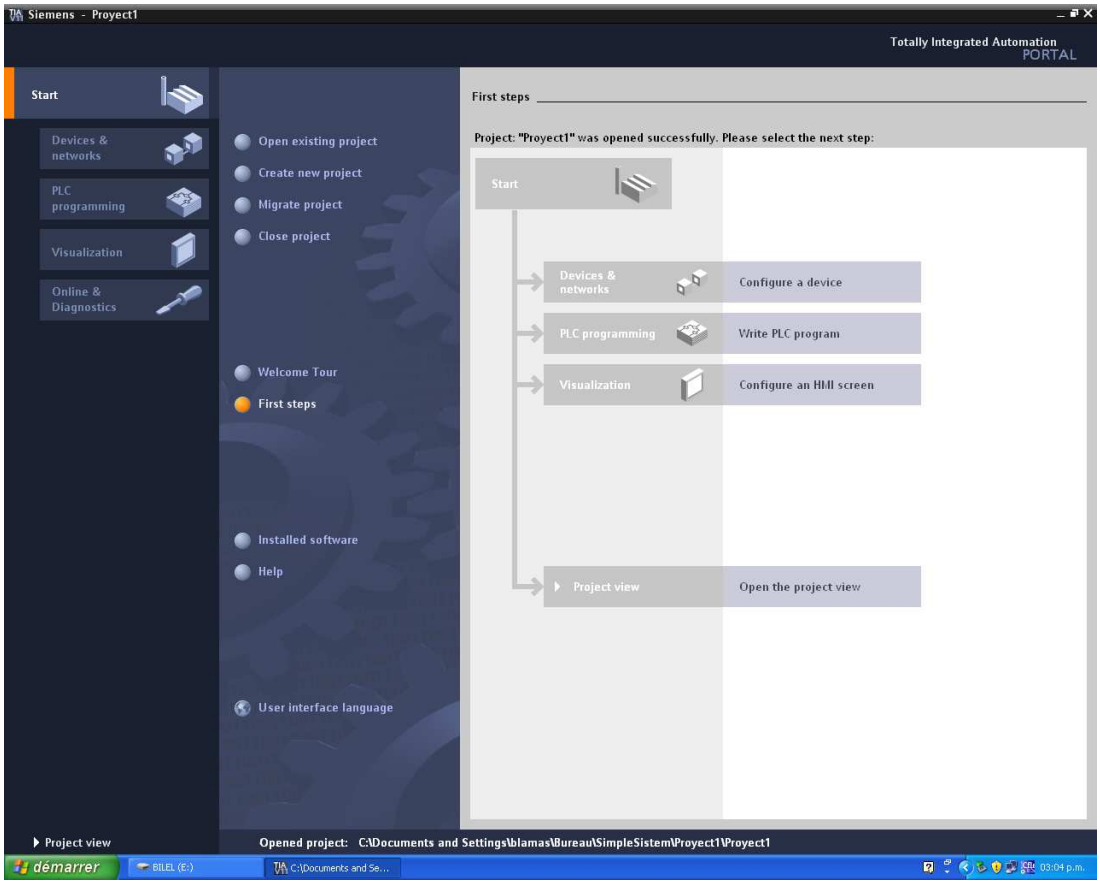


Figure F.4: The TIA Portal interface

```
TRUNCATE TABLE syslog.logs;
```

- Apply the LUA script on the pcap file using the following command at the command prompt:

  ```
  $ tshark -r file.pcap -X lua_script:file.lua
  ```

  The script filters the PROFINET frames between the PLC and the I/O block. For every kept frame, it transfers a set of information to a Mysql database (Syslog database) using the Syslog logging system.

- The information got from a filtered PROFINET frame is concatenated and stored in one field for every Syslog database entry. We created a Perl parser that creates a new Mysql database table where to store the parsed information of filtered frames. To run the parser, type the following command and indicate a name for the new database table:

  ```
  $ perl createDB.perl db_table_name
  ```

- We use a second script written in Perl to generate the XES model file. It scans the newly created database table and associates an event to each database entry where a change in input or output values is detected. Execute the script using the following command:

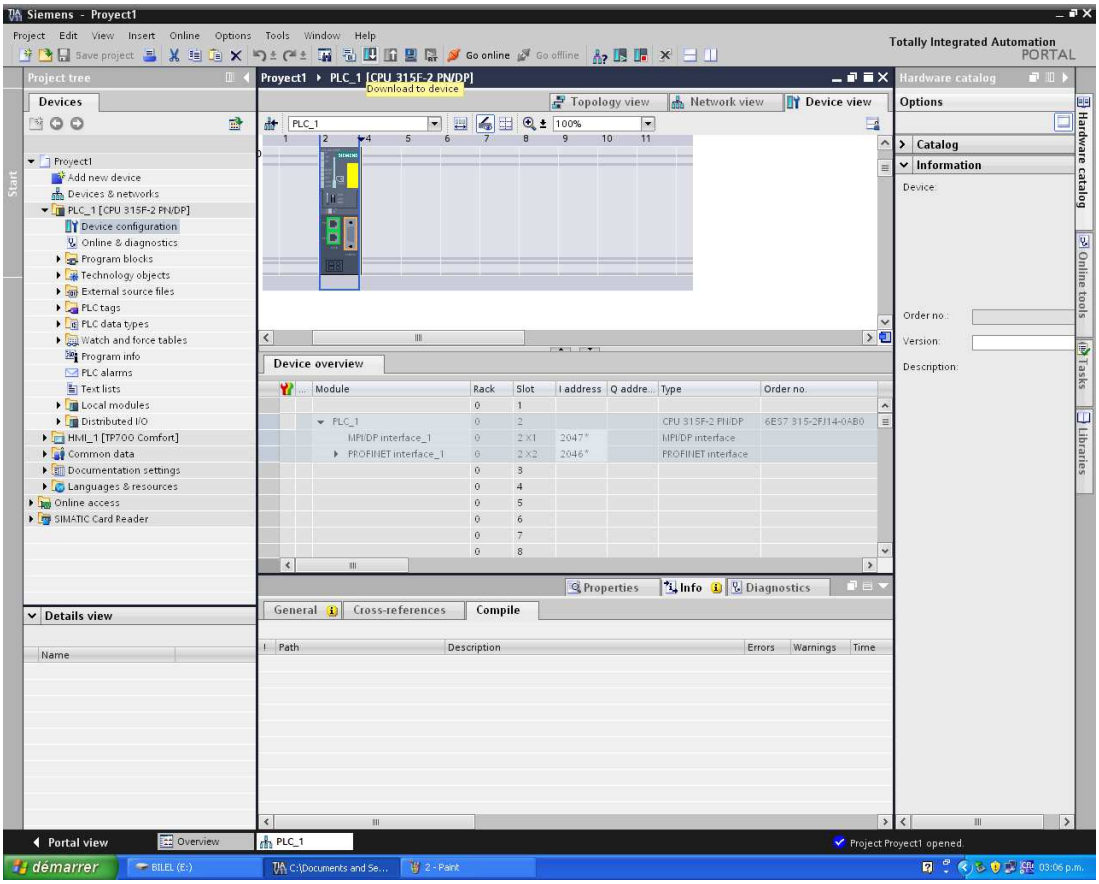


Figure F.5: The PLC programming interface

```
$ perl generateXES.perl db_table_name
```

Note that the script alters the database table by adding two columns: 'events' and 'timestamp'. Events field is set at 1 when the corresponding entry marks a system transition (a new event). Timestamps is the time in which the system remains at the state following that transition.

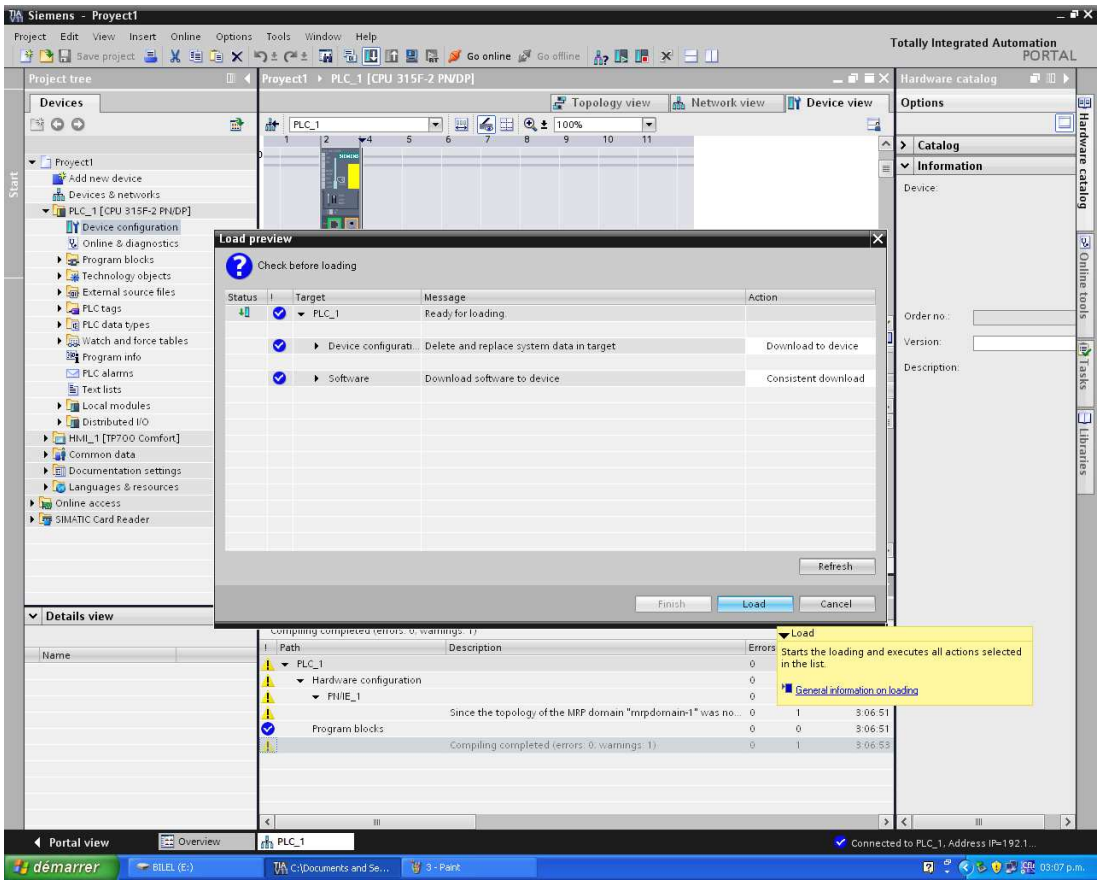


Figure F.6: Loading the PLC driver

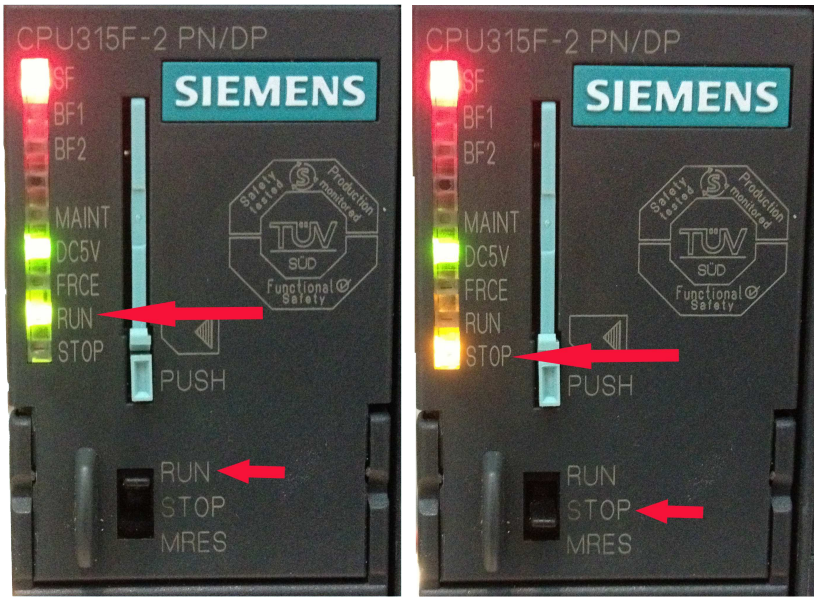


Figure F.7: The PLC run button

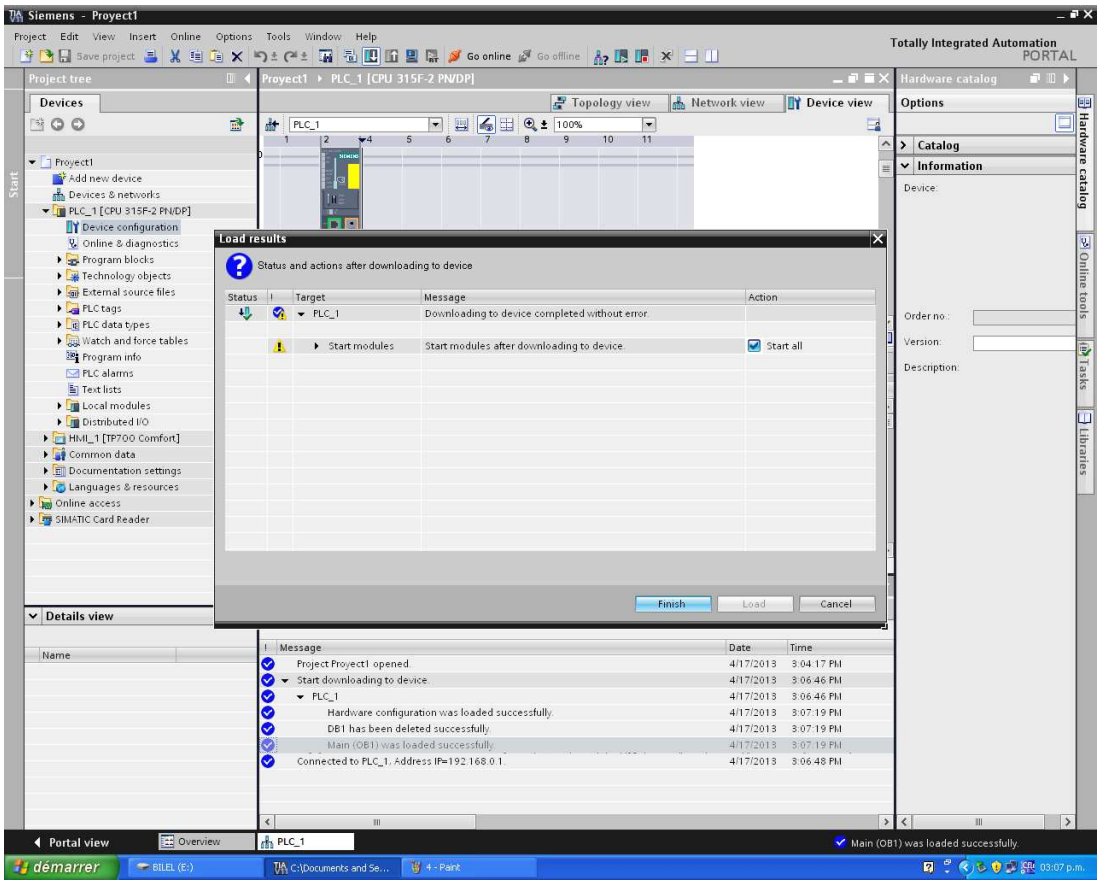


Figure F.8: Start the PLC module

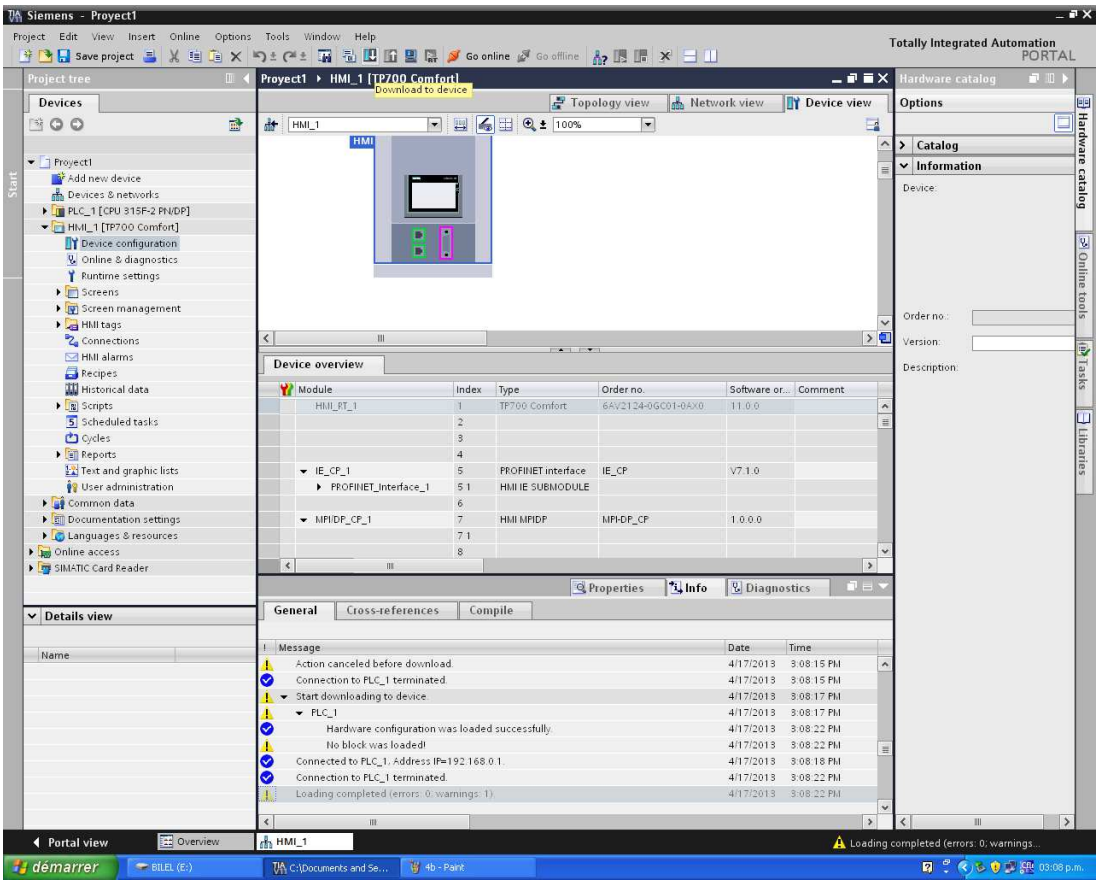


Figure F.9: The HMI programming interface

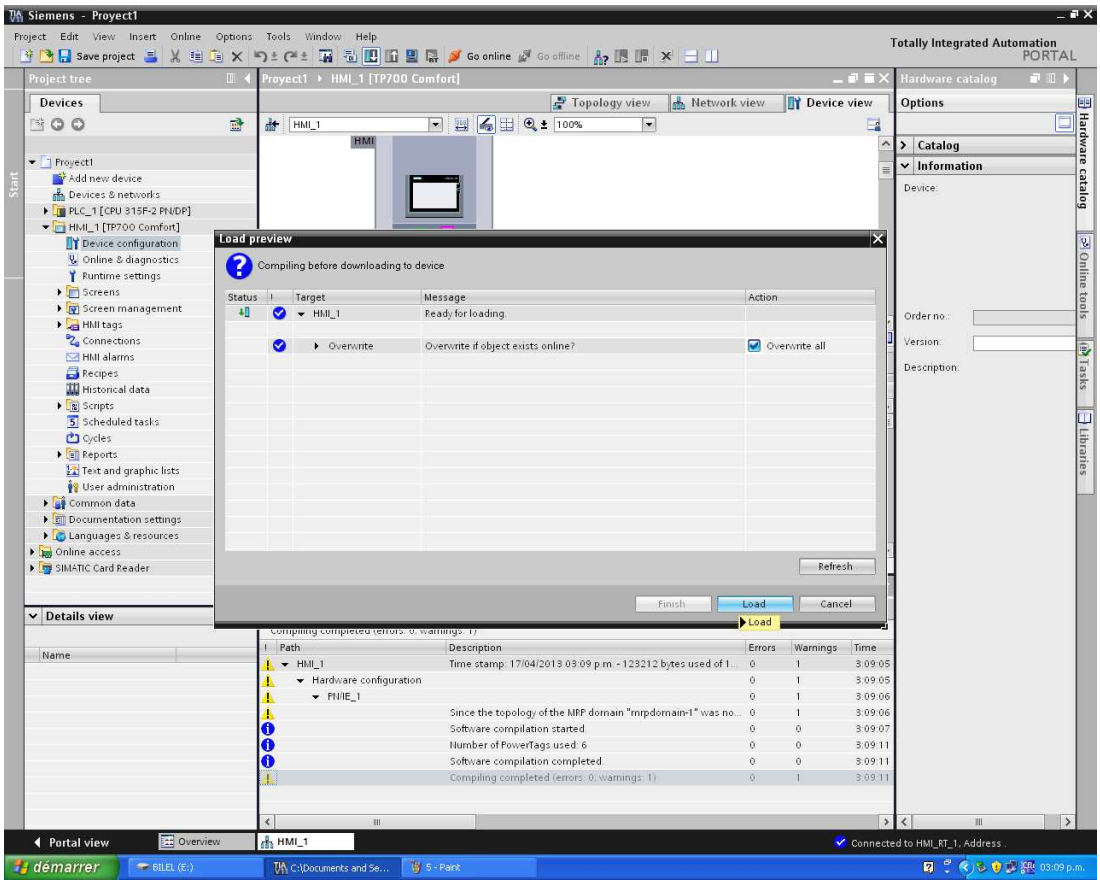


Figure F.10: The HMI driver loading window

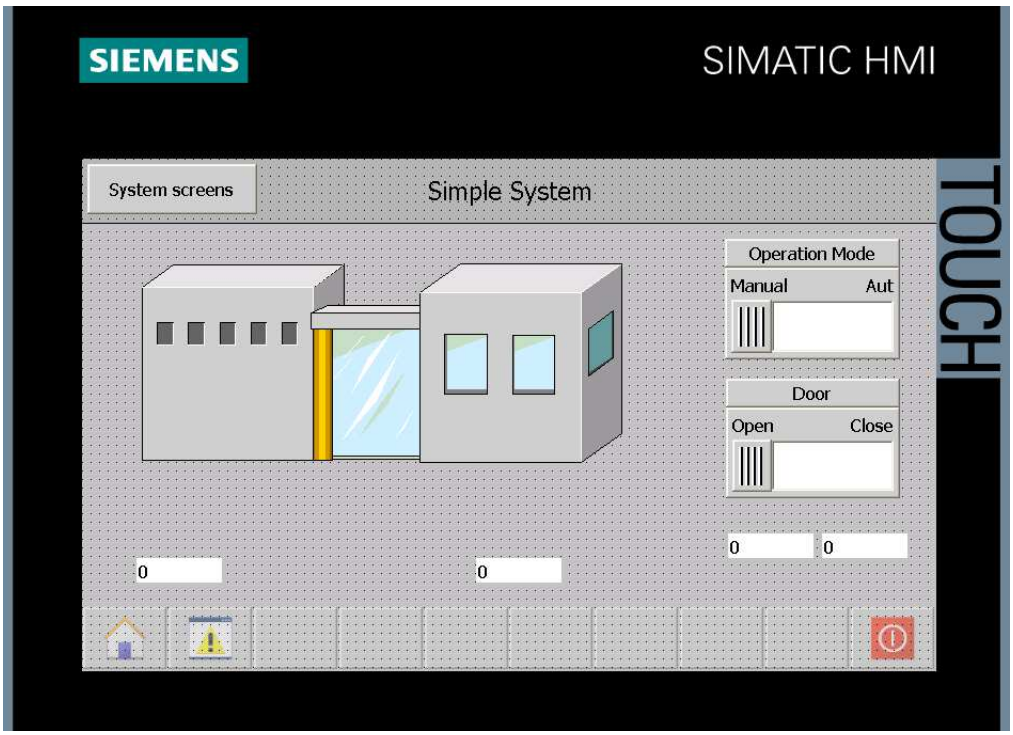
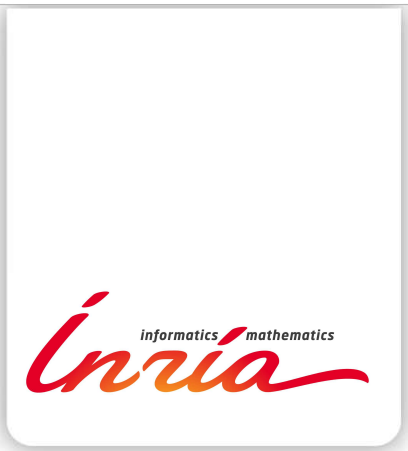


Figure F.11: The Sliding Door HMI interface



RESEARCH CENTRE
NANCY – GRAND EST

615 rue du Jardin Botanique
CS20101
54603 Villers-lès-Nancy Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-0803