TÉCNICO LISBOA

Validation of Industrial Processes Implemented on PLCs based on Petri Nets

Hugo José Damas Mora Conde Barroso

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Supervisor: Professor José António da Cruz Pinto Gaspar

February 2021

Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Resumo

Os Sistemas de Eventos Discretos (SED) são ubíquos na indústria, aparecendo normalmente como supervisores implementados em Controladores Lógicos Programáveis (CLP). Projetar e implementar um SED em CLP é um processo moroso que normalmente requer várias fases de verificação e validação.

As redes de Petri são um paradigma de modelação prático, aplicável a uma grande variedade de SED, principalmente devido às suas representações claras e expressivas, gráficas e matemáticas. Ao modelar os sistemas com Redes de Petri IOPT, uma classe de redes de Petri estendida com recursos de entrada e saída, caso as redes sejam seguras (ou limitadas), pode-se traduzir tais modelos diretamente para programas de CLP. Isto fornece uma maneira eficaz de criar programas de CLP harmonizando a complexidade de representar SEDs.

Uma rede de Petri, representante de um processo, permite encontrar o conjunto de estados alcançável, que é uma via para verificar e validar a produção de código CLP. Uma árvore de cobertura baseada no conceito de dominância de nó é proposta para considerar apenas processos com conjuntos alcançáveis finitos. Finalmente, encontramos ciclos de operação, evitando assim casos de *deadlock*, e usamos as sequências de transições para avaliar se o código do processo atinge efetivamente todos os estados possíveis.

A ferramenta *IOPT tools*, criada na Faculdade de Ciências e Tecnologia para desenvolver controladores de sistemas embebidos, é utilizada nesta dissertação para criar redes de Petri IOPT. As redes de Petri projetadas são então utilizadas como entrada da ferramenta geradora de controladores CLP/SED, criada no Instituto Superior Técnico para ensino de automação de processos industriais.

Foram obtidos resultados promissores em casos práticos de validação da produção de código CLP, pelo teste exaustivo de estados alcançáveis determinados a partir da rede de Petri que representa o processo industrial. Adicionalmente, a metodologia proposta mostrou ser possível detectar antecipadamente problemas de design e estudar os efeitos de restrições impostas por hardware.

Palavras chave: Sistemas de Eventos Discretos (SED), Redes de Petri, Redes de Petri IOPT, Alcançabilidade, Árvore de Cobertura, Controlador Lógico Programável (CLP), Structured Text, Validação, Verificação.

Abstract

Discrete Event Systems (DES) are commonly found as supervisors in industrial applications which are, in general, implemented by Programmable Logic Controllers (PLC). Designing and implementing a DES in a PLC is a thorough process typically requiring multiple verification stages and process validation.

Petri nets are a powerful modeling paradigm for a variety of DES, mostly because of their clear and expressive, graphical and mathematical, representations. By modeling the systems with IOPT Petri Nets, a class of Petri nets extended with input and output capabilities, in the case the nets are safe (or bounded), one may translate those models directly to PLC programs. This provides an effective way to create PLC programs by mediating the complexity of representing DES.

Verification and validation are required to assess the PLC code production. A Petri net representing a process allows finding the reachable set of states. A coverability tree based on the node dominance concept is proposed to consider just processes with finite reachable sets. Finally, we find operation cycles, avoiding therefore deadlock cases, and use sequences of transitions to assess whether the process code effectively reaches all possible states.

The *IOPT tools* toolchain, created in Faculdade de Ciências e Tecnologia to develop embedded system controllers, is used in this thesis to design Petri nets. The designed Petri nets are then used as the input of the PLC/DES controller maker toolchain, created in Instituto Superior Técnico for teaching automation.

Promising results were obtained in practical cases of validation of the production of PLC code, by the exhaustive test of reachable states determined from the Petri net that represents the industrial process. In addition, the proposed methodology showed that it is possible to anticipate the detection of design problems and study the effects of restrictions imposed by hardware.

Keywords: Discrete Event System (DES), Petri Net, IOPT Petri net, Reachability, Coverability Tree, Programmable Logic Controller (PLC), Structured Text, Validation, Verification.

Contents

List of Figures

List of Tables

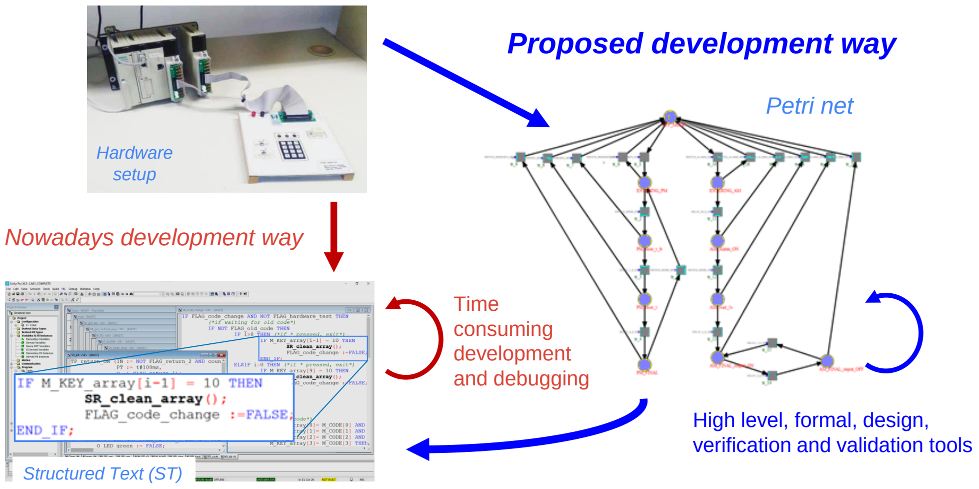
Chapter 1

Introduction

Industrial processes are procedures involving chemical, physical, electrical or mechanical steps to aid in the manufacturing of an item. When each step or set of steps is thought of as the *state* of the process and the inputs given by a user or sensor as *events* defined at discrete time instants only, one can describe every industrial process as a *Discrete Event System* (DES). A DES is a method for the modelling of systems as sequences of operations (events) - it is described as a discrete-state and event-driven system. Reasons as to why discrete-time systems might be a good approach include the fact that any digital computer that might be used as a component of a system is equipped with an internal discrete-time clock, meaning variables or controls given are only evaluated at those time instants.

Programmable Logic Controllers (PLCs) are the most common devices for integrating and controlling industrial processes. PLCs are industrial solid-state computers that monitor inputs and outputs, and make logic-based decisions for automated processes or machines. PLCs are robust and can survive harsh conditions including severe heat, cold, dust, and extreme moisture. However, despite the widespread usage based on standard programming languages, it is still time consuming their direct programming.

In this thesis, we focus on a more convenient way to create PLC program: convert higher level design (IOPT Petri nets) to PLC languages (Structured Text) [40, 41], as shown in Figure 1.1. However, there may be errors in the conversion from Petri net to Structured Text as well as conceptual errors in the Petri net design.


Proposed development way
Petri net
Hardware setup
Nowadays development way
Time consuming development and debugging
High level, formal, design, verification and validation tools
Structured Text (ST)

```
IF M_KEY_array[i-1] = 10 THEN
        SR_clean_array();
        FLAG_code_change :=FALSE;
END_IF;
```

Figure 1.1: Automated conversion from high level Petri net to Structured Text as an faster alternative to direct PLC programming. May induce errors in the conversion as well as conceptual errors in the Petri net design.

1.1 Programming and Validation based on Petri nets

Features, properties and other information may be lost in translation, causing the PLC implementation to not be exactly the desired one. An incorrectly programmed PLC can result in lost productivity and dangerous conditions. Testing the project in simulation improves its quality, increases the level of safety associated with equipment and can save costly downtime during installation and commissioning of automated control applications since many scenarios can be tried and tested before the system is activated. A validation process must be imposed to check that the software system meets the specifications and fulfills its intended purpose.

The fact that PLCs work with scan cycles have to be considered in the assessment that the implemented system meets the purpose. The PLC scan cycle is formed by scanning inputs, then executing the program while inputs and outputs are kept unchanged, and finally generating the output signals. PLCs execute programs using the input values read at the end of the input scan cycle. Therefore, input signals can be seen as discrete events. The same occurs when output values. The PLC will receive the inputs given by sensors belonging to the real system and will provide output to actuators for the same system.

The modeling of a PLC as a discrete events system (DES) allows testing and verifying programs by using input sequences. In a digital environment one can simulate and inspect how the designed program responds and actuates. Simulation tools also need to be verified and validated to assure their correctness so it can best translate both the DES model and its response to inputs.

Many industrial processes based on PLCs can be seen as discrete event systems, which in turn can be represented by Petri nets. Then, resorting to a toolchain that converts a given Petri net to PLC Structured Text code as well as Petri net and PLC simulation tools, it is possible to model, analyse, implement and control these systems. Validation of the toolchain is therefore a necessary step towards evaluating its correctness and robustness, eliminating any bug that might misrepresent the DES model or its relation to inputs and outputs. Focusing on the Input-Output Place-Transition Petri net subclass, validation testing is done to the toolchain simulation processes and new utilities are added to improve its robustness.

1.2 Related Work

Petri nets were first developed by C.A. Petri in the 1960s [39] for the purpose of describing automata communications. That doctoral thesis was followed by a series of papers by Petri himself and his research group on the applicability of Petri nets in other fields of science.

In 1969 [23], Karp and Miller wrote a paper on parallel program schemata and vector addition systems, also known as Petri nets, and dwelled on properties particular to parallel computation.

Later in 1981 [33], E. Mayer proved decidability for the reachability problem for Petri nets, a central problem of net theory. In the same year, due to the continuous increase of Petri net usage, J.L. Peterson published the first complete book [38] compiling the major parts of on Petri net theory scattered among many sources, presenting them in a coherent and consistent manner.

In 1986, a general formalization of Interpreted Petri nets was then established by Manuel Silva [43], where actuator signals and sensor signals are modeled as one or more input and output alphabets respectively.

Still regarding Petri Nets, Cassandras and Lafortune published a book in 2008 [2] intended to be a comprehensive introduction to the field of discrete event systems. This book is renowned within the academic community. It also contains an easily understandable description of the coverability tree algorithm that is used within this thesis.

In 2005, R. Pais, S. P. Barros and L. Gomes [36] developed a new Petri net class based on place/transition nets and well-known concepts from Interpreted Petri nets: the IOPT Petri net class [14]. This Petri net class allows the association of external input signals to transitions and the association of external output signals to transitions and place markings. Additionally, the class provides support for the specification of input and output events. The same authors explored this concept for several applications in other works [17]. In one of them, in 2013 [37], they developed software to convert Petri Nets to VHDL that inspired Gonçalves, in 2015 [15], to explore a way to convert a Petri net into PLC programming language Structured Text (ST).

The IOPT Petri net authors also took part in developing a web-based IOPT Petri net tool that converts a Petri net to PNML [16], among other formats and utilities. Taking on the work of Gonçalves, the DES to PLC conversion toolchain was improved and worked up by J. Meleiro [35], R. Rei [40] and R. Reis [41].

1.3 Objectives and Challenges

In this thesis we propose validation tools for a toolchain that creates PLC programs, in Structured Text, from Petri nets. Are used IOPT Petri nets in order to include input and output, signals and events. The toolchain allows the simulation of the IOPT Petri nets in addition to the translation into Structured Text. Use and validation of the toolchain involves verification and validation of intermediate steps, including specific objectives and challenges:

(i) Design IOPT Petri nets representing DESs performing tasks at hand. Challenges are found in using the IOPT web tool to design Petri nets modeling the intended DESs. In particular is important to properly read, translate and organize the data extracted from the tool, safeguarding all details and structure of the designed Petri net;

(ii) Generation of the coverability tree as per Cassandras' algorithm [2] in order to detect unbounded Petri nets. Only bounded Petri nets are considered in this thesis. From the tree we obtain the sequence of events for testing. Obstacles laid in formulating the data structure to represent the tree while making available organized information to compile the sequence of events;

(iii) Validating a Petri net with the necessary test sequences on a PLC or in simulation. Case study Petri nets are considered. Traditionally, this is a time consuming step, as developers are required to check each output of the simulation and compare it to the expected results. The challenge consists in finding and using, automatically, test sequences;

(iv) Assessment of toolchain robustness to problems originated from the connection of digital systems to noisy, transient-prone, *bouncing* inputs. Solutions for the focused problem are also proposed.

1.4 Thesis Structure

Chapter 1 introduces the problem to approach in the thesis, in particular it presents a short discussion on the state of the art of Petri Nets, the importance of PLCs in modern industry and the advantages of digitalisation for testing before implementation. Chapter 2 states the distinction between verification and validation. It also presents the relevant definitions and concepts regarding Petri nets and the coverability tree, as well as an algorithm to generate it. Moreover, some insight is given on the subject of Petri nets complemented with input and output. Chapter 3 introduces both the DES to PLC conversion toolchain to be verified and further developed and the chosen alarm case study. Chapter 4 contains the implementation of the coverability tree algorithm. It details the generation of event sequences and the extension of the Petri net model to include the notion of time. Chapter 5 provides an overview of three different use cases for the validation of the toolchain. Chapter 6 summarizes the work performed, its achievements, and proposes further work to extend the activities described in this document.

Chapter 2

Background

One of the objectives of this thesis is to implement and test the algorithm presented in [2] for constructing the finite version of an infinite reachability tree called a *coverability tree*.

In this chapter the issue of validation versus verification is addressed. The Petri Net mathematical modelling language is introduced and narrowed down to the class of Input-Output Place-Transition (IOPT) Petri Nets. After presenting the reachability problem as one of the central problems in Petri net theory, the concept of *coverability tree* is distinguished from *reachability tree*. The description and implementation of the aforementioned algorithm is then presented, preceded by the introduction of some necessary notation, and concluding with the testing of a few examples.

2.1 Verification vs Validation

Quality control is essential to building a successful business that delivers products that meet or exceed customers' expectations. It also forms the basis of an efficient business that minimizes waste and operates at high levels of productivity. Verification and Validation (V&V) are two of the most important terms in the industry when it comes to creating a process or product as they are mandatory steps for the quality management process: it makes sure that the process or product meets the purpose intended. Regulatory authorities like EMA (European Medicines Agency) and FDA (Food and Drug Administration) have published guidelines relating to process validation[1].

Verification evaluates the on-going development phases and the product on regular basis

[1]`https://www.fda.gov/regulatory-information/search-fda-guidance-documents/process-validation-general-principles-and-practices`

against the specified requirements. Verification testing is a type of static testing, which means it involves the correct functioning of each part on its own. It can be seen as low level testing. The verification of system configuration can ensure system behavior is trusted if system components are configured properly. This means that all system services and applications should run as specified and contain no bugs or vulnerabilities that could be exploited to affect the functioning of the whole system.

Validation ensures that the product meets the predefined and specified business requirements as well as the end users/customers' demands and expectations. Validation testing is a form of dynamic testing, meaning it is performed in an environment where the code is actually executed. This testing takes on the final software product obtained after the development process, it can be seen as a high level testing.

The pharmaceutical industry is well know by its V&V standards, as ISO 9000. Electronics and computer hardware and software industries also follow similar terminology and body of knowledge[2]. For software systems it is called Software Verification and Validation (SVV) and has been formalised in IEEE 1012-2016 [3]. When it comes to PLCs, the IEC evaluates the complete projects of PLCs, including hardware, installation, testing, documentation, programming and communication, and develops International Standards for all electrical, electronic and related technologies.

In software project management, engineering and testing, the terms Verification and Validation are defined as the process of checking whether or not a software system meets specifications and fulfills its desired purpose. SVV for control systems can be simply addressed considering formal approaches or simulation. On the one hand, formal techniques are less used due to the complexity of formalisms and languages to be approached by the industrial user. Usually, implemented algorithms for PLCs use languages proposed in the standard IEC 61131-3 [22]. Even though the standard has harmonized how PLCs are programmed, the standardized languages do not force programmers to implement their algorithms in a formal way. Thus the definition of the control system formal model is not as simple and intuitive as to build a simulation model. On the other hand, simulation is a widely recognized and adopted technique for verification and validation of industrial automation systems. The main open problem of this approach is the definition of test cases to analyse the model in a complete and exhaustive manner.

While in formal verification [20, 21] are used formal methods of mathematics to prove or disprove the correctness of intended algorithms, we consider software tools associated to DES and Petri nets to confirm the specifications are met.

[2]https://www.pmi.org/pmbok-guide-standards/foundational/pmbok/software-extension-5th-edition

In this thesis, we focus on validation of the simulation processes of the DES to PLC conversion toolchain . We propose an automatic generation of the test sequence that helps us simulate the Petri net. Through manual comparison, since we know the expected evolution of the DES given a test sequence, we can test and analyse multiple critical cases to ensure the simulation environment behaves correctly.

2.2 Petri Nets

In this work, Petri nets are chosen for the description of DESs, since it is the most common representation for models of DES. Their basic characteristic is that they provide an excellent tool for capturing concurrency and conflict within a system. They have an appealing graphical and mathematical representation and they are well accepted when it comes to the implementation of tools [44].

Definition

A Petri Net is a collection of *oriented arcs* connecting *places* and *transitions*. Places may hold any non-negative number of *tokens*, having infinite capacity by default. The marking of a Petri net is a vector formed by the number of tokens in each place. The marking of all places characterizes the state of the Petri net. Arcs have weight 1 by default; if other than 1, the weight is marked on the arc. Arcs can only connect transitions to places and vice-versa, there are no arcs connecting transitions to transitions or places to places. Transitions are associated with events and can be fired, i.e., are enabled when the number of tokens in each of its input places is at least equal to the arc weight going from the place to the transition. An enabled transition may fire at any time. When fired, the tokens are subtracted from the the input places and are added to the output places, according to arc weights, resulting in a new marking of the net. This way tokens may disappear or be created.

The procedure of defining a Petri net involves two steps. First, the Petri net graph is defined, meaning the places, transitions and oriented arcs are drawn as desired. Places are graphically represented by circles, transitions by bars and oriented arcs by lines with an arrow setting orientation. Secondly, an initial set of marked places, also named *initial marking*, is added alongside a *transition function* resulting in the complete Petri net model and the corresponding dynamics as well as the languages that it generates and marks.

From [2], a Marked Petri net is formally defined as a a five-tuple (P, T, A, w, μ_0), where

1. P is the finite set of places defined as $P = (p_1, p_2, ..., p_n), n \in \mathbb{N}$

(P, T, A, w, μ_0)

$P=\{p_1, p_2, p_3, p_4, p_5\}$

$T=\{t_1, t_2, t_3, t_4\}$

$A=\{(p_1, t_1), (t_1, p_2), (t_1, p_3), (p_2, t_2), (p_3, t_3),$
$(t_2, p_4), (t_3, p_5), (p_4, t_4), (p_5, t_4), (t_4, p_1)\}$

$w(p_1, t_1)=1, w(t_1, p_2)=1, w(t_1, p_3)=1, w(p_2, t_2)=1$
$w(p_3, t_3)=2, w(t_2, p_4)=1, w(t_3, p_5)=1, w(p_4, t_4)=3$
$w(p_5, t_4)=1, w(t_4, p_1)=1$

$\mu_0 = \{1, 0, 0, 2, 0\}$

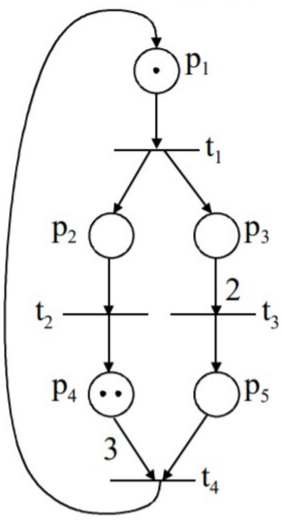


Figure 2.1: Example of a Petri net.

2. T is the finite set of transitions defined as $T = (t_1, t_2, ..., t_m), m \in \mathbb{N}$

3. A is the finite set of arcs from places to transitions and from transitions to places in the graph defined as $A \subseteq (P\text{x}T) \cup (T\text{x}P)$

4. w is the weight function on the arcs defined as $A \to 1, 2, 3, ...$; one assigned weight for each arc in A. The matrix comprised of all $w(t_j, p_i)$ values is called the postconditions matrix D^+ and the matrix comprised of all $w(p_i, t_j)$ is defined as preconditions matrix D^-, with $D^+,\ D^- \in \mathbb{R}^{nxm}$.

5. μ_0 is the initial state of the Petri net given by the markings of all the places. A Petri net state is defined as $\mu = [\mu(p_1), \mu(p_2), ..., \mu(p_n)] \in \mathbb{N}^n$.

For simplicity, a marked Petri net shall henceforth be referred to as just a Petri net. Figure 2.1 shows an example of a Petri net graphically represented.

Regarding the Petri net algebraic representation, Petri nets can be represented in matrix form by the *Incidence Matrix* $D \in \mathbb{R}^{nxm}$, which in turn is obtained from the weight function w as seen in (2.1).

$$D_{ij} = w(t_j, p_i) - w(p_i, t_j) \text{ , where } i = 1, ..., n \text{ and } j = 1, ..., m. \tag{2.1}$$

The incidence matrix for Petri net in Figure 2.1 is given by

$$D = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} - \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \Leftrightarrow D = \begin{bmatrix} -1 & 0 & 0 & 1 \\ 1 & -1 & 0 & 0 \\ 1 & 0 & -2 & 0 \\ 0 & 1 & 0 & -3 \\ 0 & 0 & 1 & -1 \end{bmatrix} \tag{2.2}$$

In the interest of understanding the dynamics of Petri nets, the two following concepts must be introduced

1. *Enabled Transition*. For a given marking, a transition t_j is said to be *enabled* if all input places of t_j contains at least the number of tokens equal to the weight of the directed arc connecting each input place to t_j, i.e., $\mu(p_i) \geq w(p_i, t_j), \forall(p_i, t_j) \in w$.

2. *Firing Rule*. Only enabled transitions may fire. The firing on an enabled transition t_j removes from each input place of t_j the number of tokens equal to the weight of the directed arc connecting each input place to t_j. It also deposits in each output place of t_j the number of tokens equal to the weight of the directed arc connecting t_j to each output place of t_j.

The Petri net dynamics is given by equation (2.4), also known as state transition mechanism. A state transition occurs by moving the tokens along the net according to the firing rule by firing an enabled transition. In Petri nets, enabled transitions may fire at any time. If more than one transition is enabled, the fired transition is chosen arbitrarily. The state transition function of a Petri net is defined through the change in the state of the Petri net due to the firing of an enabled transition.

The state transition function, $f : \mathbb{N}^n$ x $T \rightarrow \mathbb{N}^n$, of Petri net (P, T, A, w, μ_0) is defined for transition $t_j \in T$ if and only if transition t_j preconditions are met. Mathematically it is written as in (2.3).

$$\mu(p_i) \geq w(p_i, t_j) \; \forall(p_i, t_j) \in w \tag{2.3}$$

If $f(\mu, t_j)$ is defined, then the next state is given by

$$\mu' = \mu + D \cdot q(j), \tag{2.4}$$

where q is the firing vector, being $q(j)$ a vector representing the firing of transition t_j.

For the scope of this thesis, another important observation about the dynamic of Petri nets must be made. It is not necessarily true that all states in $\mathbb{N}^n$ can be reached by a Petri net model. It may also occur that different initial markings lead to a different set of states reached by the same Petri net. This leads to the defining of the Reachable set $\mathscr{R}$ as being the set of all reachable states by the given Petri net for the given initial marking.

Input and Output

We want to validate the tool that converts a DES to a PLC program. After uploading the program to a PLC, one obtains a DES interacting with real systems. The Petri net representing the DES models just the structural information, state dynamics, of the system. In addition, it is important to test how the proposed DES responds to external inputs and acts on external outputs. To do so, we consider a class of Petri net that complements our common Petri net with input-output interactions. The following statements on the IOPT Petri net class mention only the characteristics necessary to learn how to use the IOPT tool software [16] to design an IOPT Petri net model abiding by the rules and conditions that must be met before feeding it to the DES to PLC conversion toolchain .

IOPT Petri nets [36] are based on Place-Transition nets and well-known concepts from Interpreted Petri nets and allow the specification of models with input and output signals and events. More specifically, it allows the association of input events to transitions and output events to places. The IOPT Petri net supports the specification of input and output events which simulate the readings of sensors and manipulation of actuators. The input and output signals guide the controller through each execution step by defining the system current state while the input or output events are associated to changes in input or output signals.

The IOPT Petri dynamics are very similar to the Petri net dynamics as explained in section 2.2. Regarding transition firing, now a transition has to be both *enabled* and *ready* to be firable. A transition is ready when the associated input event happen. After a transition fires, the marking changes according to the usual Petri net dynamics. If the marked places are associated with an output event, these events will be triggered and the corresponding output signal's value will change.

Properties

Modeling with Petri nets by itself is of little use. It is necessary to analyze the modeled system to better understand its behaviour. The objective of analysing a Petri net is to determine the answers to questions about the Petri net. To better understand the usefulness of Petri net analysis techniques we introduce the questions by stating the adjacent property. Let C be the Petri net defined by (P, T, I, O, μ_0).

1. **Safeness** - important property for Petri nets modeling hardware devices. If a place is safe, then the number of tokens in that place is either 1 or 0. The interpretation made is that of a place representing a logical condition, it is either correspondingly true or false making a place easily implemented as a flip-flop.

   A place $p_i \in P$ of C is safe if $\forall \mu' \in R(C) : \mu'_i \leq 1$.

   A Petri net is safe if all its places are safe.

2. **Boundedness** - Safeness is a special case of the boundedness property. In this more general property, one can interpret a place as a counter. Any hardware counter has a limit to which it can count, therefore the boundedness of each place is relevant to prevent the hardware from malfunctioning.

   Boundedness defines the maximum number k of tokens a place can hold. A place $p_i \in P$ is k-bounded if $\mu'_i \leq k \; \forall \mu' = (\mu'_1, ..., \mu'_i, ..., \mu'_N) \in R(C)$.

   A Petri net is k-bounded if all places are k-bounded.

3. **Conservation** - Petri nets can be used to model resource allocation systems. In these systems some tokens may represent said resources. A Petri net should conserve the resources which it is modeling, i.e., tokens that represent resources are neither created nor destroyed.

   If for every marking the number of tokens remains the same, a Petri Net is strictly conservative. Mathematically

   $$\sum_{p_i \in P} \mu'(p_i) = \sum_{p_i \in P} \mu(p_i)$$

   .

4. **Liveness** - The system transitioning into a deadlock is not desirable for obvious reasons. A deadlock in a Petri net is a transition (or a set of transitions) which can never fire. A transition is *live* if it is not deadlocked. This does not mean that the transition is enabled, but rather that it can be enabled. The liveness property studies the possibility of a given transition being able to eventually fire, or not. There are 5 levels of liveness. We say a transition t_j is live of

   - Level 0 - if it can never be fired (transition is Dead).

- Level 1 - if it is potentially firable an upper-bounded number of times, that is, if there exists a $\mu' \in R(C)$ such that t_j is enabled in μ'.
- Level 2 - if for every integer n, there exists a firing sequence such that t_j occurs n times, i.e., the upper-bound number transition t_j can be potentially fired can be set by a certain firing sequence or by the repetition of a certain firing sequence.
- Level 3 - if there exists an infinite firing sequence such that t_j occurs infinite times. This is the same as saying transition t_j can be fired an infinite number of times until the Petri net reaches a state where t_j can never fire again.
- Level 4 - if the transition is live of level 1 for every possible state reached from μ_0. In other words, if for each $\mu' \in R(C)$ there exists a sequence σ such that the transition t_j is enabled (transition is Live).

5. **Reachability** - When modeling a real system it is important to know if the systems reaches all expected steps of functioning or if a known deadlock can happen. Since these steps can be seen as states in the Petri net, the reachability problem arises from the need to know whether from the given initial marking there exists a sequence of valid execution steps that reaches the given final marking.

   The common definition of the reachability problem is: given a Petri net C, does the marking μ' belong to the set of all markings that can be obtained, i.e., $\mu' \in R(C)$?

6. **Coverability** - The coverability problem is similar to the reachability one. Answering this problem plays a central role in the verification of concurrent shared-memory programs.

   Given a Petri net C and states $\mu,\ \mu' \in R(C)$, then μ' is covered by μ if $\mu'(p_i) \leq \mu(p_i)\ \forall p_i \in P$.

7. **Temporal Invariance** - This property addresses the ability of a machine being able to return to a specific state from itself, if cycles of operation can be found. An example for a real system is the ability to to perform a reset on its own without external intervention. For a given Petri net C, one can infer the existence of temporal invariance if there exists a sequence of transitions fired that evolve the Petri net from each state to itself.

Petri Net Analysis, Method of Matrix Equations

The Method of Matrix Equations (MME) is based on the dynamics of the Petri net as seen in (2.4) and conditions are imposed to study each property. This methodology allows the immediate study of three of the above mentioned properties:

- **Conservation** - To maintain the (weighted) number of tokes one writes:

$$w^T \cdot \mu' = w^T \cdot \mu + w^T \cdot D \cdot q, \tag{2.5}$$

where w^T is the vector of weights of each transition. As seen before, the total number of tokens can not vary, therefore $w^T \cdot \mu' = w^T \cdot \mu$. Since the total number of tokens is the same regardless of the transition firing vector q, one needs only to find the vector w such that

$$w^T \cdot D = 0. \tag{2.6}$$

$\exists w(t_i) > 0 \; \forall t_i \in T$ is a necessary and sufficient condition to prove the Petri net is conservative.

- **Reachability** - Given the final state $\mu(k+1)$ and the starting state $\mu(k)$, one can assess if state $\mu(k+1)$ is reachable from state $\mu(k)$ by solving the equation of Petri net dynamics in order to the transition firing vector q, which is the only unknown.

$\exists q$ such that $Dq(k) = \mu(k+1) - \mu(k)$ is a necessary condition to prove reachability for the presented conditions, but not sufficient to derive the correct order in which the transitions must be fired. Even if the system holds a solution it may not be possible to reach the desired state, since the MME mathematical description does not take into account restrictions found in the Petri net's dynamics. It can happen if, for example, for all possible order of transition firings the system evolves into a deadlock. The difficulty of finding a set of constraints that guarantees that the solution is feasible makes the reachability problem one of the main questions in Petri net theory, which will be discussed on section 2.3.

- **Temporal Invariance** - Since one wants to determine the transition firing vectors that make the Petri net return to the same state one makes $\mu' = \mu$ and hence one needs only to calculate the firing vector q such that

$$D \cdot q = 0. \tag{2.7}$$

$\exists q$ is a necessary to prove temporal invariance, but not sufficient to conclude on the correct firing vector.

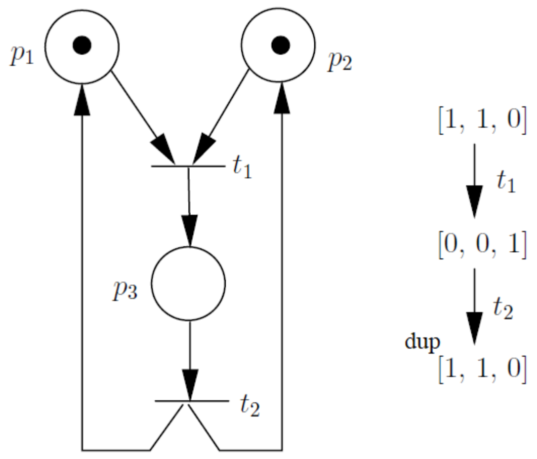


Figure 2.2: Example of a Petri net (left) and its corresponding reachability tree (right), if the initial marking is $[1; 0; 0]$.

Petri net Analysis, Reachability Tree

The set of all states reachable by a Petri net C from the initial state μ_0 is called reachable set $\mathscr{R}(C, \mu_0)$. The reachability set can be graphically represented by a reachability tree. One can then say a reachability tree is a tree of reachable states. Each reachable state is represented by a node in the tree. The components used to build a tree are described in subsection 2.4.

As one can see in the example of a Petri net and its reachability tree shown in Figure 2.2, the transitions that are fired to change between states are also represented attached to the arcs in the tree, providing extra information such as the sequence of transitions needed to reach a state from any other, if possible.

Depending on the Petri net, more precisely, on whether the Petri net is bounded or unbounded, the reachability tree can be finite or infinite, correspondingly.

When the reachable set is finite it may be represented by the finite reachability tree, allowing the study of all previously enunciated properties.

When the reachable set is infinite the reachability tree becomes infinite. There exists a way of representing an infinite reachability tree in a finite form by introducing the infinity symbol ω and the notion of *node dominance*, which are presented in section 2.4. The advantage of getting a finite representation is accompanied with loss of information regarding the reachability property. It is often possible, for instance, to determine that some state is not reachable, while being unable to check if some other state is reachable. Such instances give way to the reachability problem.

2.3 The Reachability Problem

The reachability problem is a decision problem, meaning it is a problem with yes or no answer. To solve a problem one develops procedures using computing resources. The formal descriptions of these procedures are called algorithms. A problem is said to be undecidable if there is no algorithm that takes as input an instance of the problem and determines whether the answer to that instance is yes or no.

The reachability problem for Petri nets consists of deciding, given a Petri net C with initial marking μ_0, if a possible marking μ can be reached from μ_0, i. e. if $\mu \in \mathscr{R}(C, \mu_0)$.

This problem was first proposed by Karp and Miller [23] within the scope of Vector Addiction Systems, but left unsolved.

Decidability of the Reachability Problem

Hack [19] and Keller [24] observed that many other problems were recursively equivalent or reducible to the reachability problem turning it into on of the most studied decision problems in computer science theory.

Inspecting Esparza's research on decidability issues for Petri nets [9], we uncover that, after an incomplete proof by Sacerdote and Tenney [42], decidability of the problem was established by Mayr in his seminal STOC 1981 work [33]. Presenting a structure called *regular constraint graphs*, Mayr introduces an algorithm whose nondeterministic computation will determine if there is a firing sequence or not. Hence, a deterministic implementation of the algorithm provides a decision procedure for the general Petri net reachability problem. This algorithm uses a slightly modified version of Karp and Miller's original reachability tree construction [23] and is based on conditions given by Presburger's Arithmetic.

The proof was then simplified by Kosaraju [25]. Kosaraju does not add any "significantly new ideas", he just simplifies the proof by disposing of the complicated tree constructions used by Sacerdote and Tenney, and Mayr, introducing a "more general model o VASS's" known as *Generalized Vector Addiction Systems with States* (GVASS). Further refinements were made by Lambert [26], where he completely suppressed the use of Presburger's Arithmetic.

Lambert's proof outline

Lambert [26] introduced the concept of *Marked Graph-Transition Sequences* (MGTS), which are the result of the decomposition of the *precovering graphs* implicit in Mayr's and Kosaraju's proofs.

Lambert's algorithm finds a MGTS having some properties which allow the computation of a sequence belonging to its language. The language L of a MGTS is the set of the sequences firable in the Petri net R which are made of paths in the *initiated precovering graph* (IPG) of the *graph-transition sequences* (GTS) and respect the initial and the final markings of each IPG.

He states that if a MGTS on a Petri net R, with $Ax = b$ as its characteristic equation, is *perfect* then it is possible to find any element of L. A MGTS is perfect if it contains a covering for both the initial and final marking, and if there exists a solution to the characteristic equation respecting the proper conditions.

Lambert proceeds to show how to compute the finite (possible empty) set Γ of perfect MGTS by decomposition of a MGTS $\mathscr{U}_0$, with marking φ_0 as the pair (μ_i, μ_f), on a Petri net R and initial an final markings μ_i and μ_f, respectively. This decomposition means

$$L(R, \mu_i, \mu_f) = L(\mathscr{U}_0, \varphi_0) \tag{2.8}$$

which is read as, for a Petri net R, the language of the sequences firable at μ_i for which the resulting marking is $\mu_f \in \mathbb{N}^P$ ($\mathbb{N}^P$ =markings of Petri net R) is equal to the language of the MGTS $\mathscr{U}_0$ on R, with marking φ_0 as the pair (μ_i, μ_f).

In other words, we can compute a finite (possible empty) set Γ of perfect MGTS having μ_i and μ_f as input and output marking such that

$$L(R, \mu_i, \mu_f) = \bigcup_{(\mathscr{U}, \varphi) \in \Gamma} L(\mathscr{U}, \varphi) \quad . \tag{2.9}$$

He states that reachability is decidable if the algorithm can compute for a Petri net R and two markings μ_i and μ_f a finite (possibly empty) set Γ of perfect MGTS. The algorithm is shown to converge as a consequence of the well-foundedness of multiset ordering [7] thus confirming decidability.

In the computer science community, the *decomposition technique* that lies at the heart of the proofs is called the Kosaraju-Lambert-Mayr-Sacerdote-Tenney (KLMST) decomposition.

Complexity

Regarding its complexity, Lipton showed in 1976 that the reachability problem's lower bound is EXPSPACE-hard [32].

Leroux made substantial progress over the past ten years [27, 28, 29] culminating in 2015 in the first upper bound on the complexity [30].

In 2019, Leroux and Schmitz [31] published a new upper bound to be non-primitive recursive Ackermannian. Later in the same year, Czerwinski [4] established a non-elementary lower bound, i.e. that the reachability problem needs a tower of exponentials of time and space.

2.4 Coverability Tree

In 1969, R. M. Karp and R. E. Miller [23] introduced the *rooted tree* $\mathscr{T}(\mathscr{W})$, for any vector addition system $\mathscr{W}$, and the *infinity symbol* ω terminology to help represent an infinite reachability set $\mathscr{R}(\mathscr{W})$ in a finite form, in order to discuss effective tests of schemata properties.

Later in 1981, J. L. Peterson [38] used ω to represent "a number of tokens which can be made arbitrarily large", that can be thought of as "infinity", and described an algorithm to reduce the infinite reachability tree to a finite representation. Peterson chose to name both the finite and the infinite reachability tree as reachability tree.

Finally, 1993, C. G. Cassandras and S. Lafortune [2] introduced the notation of *node dominance*, which the previous authors also used but did not label, to present the technique of the *coverability tree*. The authors named *coverability tree* as the finite representation of the infinite reachability tree, which contains the infinity symbol ω.

In this work, the latter terminology is chosen since the implemented algorithm is based on theirs.

Note that seeking the finite version is always possible, meaning the algorithm always terminates, as proven in [38] who based the proof on [18] and [23].

Generating a coverability tree allows us to discard Petri net with an associated tree that contains the infinity symbol, which translates into an unbounded Petri net. Being presented with a bounded Petri net with a corresponding coverability tree containing no ω one can obtain all possible sequences leading to all reachable states. Access to this information makes the creation of test sequences possible. We first introduce some notation to better comprehend the algorithm presented in [2] and then we explain the general steps.

Notation

Before describing the algorithm, some notation regarding the components used to build a tree must be introduced as to better understand the technique.

1. ***Node***. Represent a reachable state of the Petri net.

    - *Root node*. The node at the top of the tree structure, has no parent. Corresponds to the initial state of the Petri net.

- *Terminal node*. Also known as leaf node, does not have child nodes. Corresponds to a state with no enabled transitions (e.g., deadlock).
- *Duplicate node*. Corresponds to a state that is identical to a state already present in the tree. For a node to be considered duplicate, the already existing identical node must be in the path from the root node to the node under consideration. No successors of a duplicate node need to be considered; all these successors will be produced from the first occurrence of the node in the tree.

2. ***Infinity symbol*** ω. Represents a number of tokens that can be made arbitrarily large. For any constant a

$$\begin{aligned} \omega \pm a &= \omega \\ a &< \omega \\ \omega &\leq \omega \end{aligned}$$

The symbol ω will appear as the marking of any unbounded place belonging to a state that ***dominates*** another.

3. ***Node dominance***. Let any state $\mu = [\mu(p_1), \mu(p_2), ..., \mu(p_n)]$. Consider states μ and μ' belonging to the coverability tree and n the total number of places in the Petri net. If there is a node μ' on the path from the root node to μ such that

   (i) $\mu(p_i) \geq \mu'(p_i), \forall i = 1, ..., n$ (state μ covers state μ'),

   (ii) $\exists i : \mu(p_i) > \mu'(p_i), \forall i = 1, ..., n$ (there exists at least one place of μ that has more tokens than the corresponding place of μ'),

   then $\mu >_d \mu'$, i.e. μ dominates μ'.

   Allied with the infinity symbol, node dominance is the concept that allows the representation of the coverability tree (for an example see subsection 4.3.1).

Algorithm Steps Outline

The algorithm as presented in [2] builds a coverability tree for a given Petri net. The only information required from the Petri net is the preconditions matrix, post-conditions matrix and initial state. With both matrices one can obtain the incidence matrix needed for the state evolution according to the Petri nets dynamics equation.

The tree starts out with the initial state, which is the first "new state".

For each new state, if there are no enabled transitions then the state is branded "terminal", since it cannot be expanded, otherwise the algorithm computes the next state for each one of the enabled transitions. One can think of each enabled transition as a branch on the tree.

Mark each new state according to the Petri net dynamics equation 2.4 and considering the ω properties and node dominance. Correct the marking if *node dominance* is verified. The last step on each new state is to check if there already exists a state in the tree identical to this new state. If so, brand the new state as "duplicate". Else it stays branded "new state".

Repeat for all new states until all states have been branded as either "duplicate" or "terminal".

The general steps of the algorithm to construct the coverability tree are given on Table 2.1.

Coverability Tree Algorithm - General Steps

Input: Preconditions matrix, post-conditions matrix and initial node
Output: Coverability Tree

```
Step 1 Initialize $\mu$ as the root node (initial state): $\mu_1 = \mu_0$.
Step 2 For each new node in $\mu$, evaluate the state transition function $f(\mu_i, t_j)$ for all $t_j \in T$:
    Step 2.1 If $f(\mu_i, t_j)$ is undefined for all $t_j \in T$ (i.e., no transition is enabled at state $\mu_i$), then mark $\mu_i$ as a terminal node.
    Step 2.2 If $f(\mu_i, t_j)$ is defined for some $t_j \in T$ (i.e., there exists at least one transition enabled at state $\mu_i$), for each enabled transition, create a new node $\mu'$ with the marking as given by the state transition function ($\mu' = f(\mu_i, t_j)$). Adjust as follows:
        Step 2.2.1 For each place in state $\mu_i$ that is marked with $\omega$ tokens, if there is any, set the corresponding place in $\mu'$ with $\omega$ tokens: for all $\mu_i(p_i) = \omega$ found, set $\mu'(p_i) = \omega$.
        Step 2.2.2 If there exists a node $y$ in the path from the root node $\mu_0$ (included) to $\mu'$ such that $\mu' >_d y$, set $\mu'(p_i) = w$ for all $p_i$ such that $\mu'(p_i) > y(p_i)$.
        Step 2.2.3 Else $\mu'$ remains as obtained in Step 2.2.
        Step 2.2.4 If there already exists a state in the tree identical to $\mu'$, then mark $\mu'$ as a duplicate node, else add $\mu'$ to $\mu$ as a new node.
Step 3 Stop when all nodes have been marked as either duplicate or terminal.
```

Table 2.1: Coverability Tree Algorithm General Steps as described by C.G. Cassandras and S. Lafortune in [2].

2.5 Thesis Approach

We use Petri nets to model industrial processes, namely in the aspect of creating DES supervisors implemented on PLCs. More precisely, we use extended Petri nets, allowing the communication with real systems.

The extended Petri nets provide means for simulating the interactions between the PLC (implementing the Petri net) and the inputs and outputs associated to real world (PLC external) systems. In particular we use IOPT Petri nets which are converted to structured text (PLC programming language).

Besides converting a IOPT Petri net to Structured Text, the considered DES to PLC conversion toolchain also provides a MATLAB simulation environment that allows the testing and validation of the Petri net before implementation, as well as a PLC simulator.

The simulation environment is designed to work regardless of the modeled DES, however, as any environment created by a software tool, may have translation induced errors. The simulated Petri net may behave incorrectly even if the DES is properly modeled. We propose tools to assess the correctness of the PLC/DES implementations.

In this thesis we propose validation of the simulation environment through the study of three use cases on a alarm system based on a PLC. We also propose implementing the automatic generation of all possible cyclic input event sequences, extracted from the reachable set, to simulate all possible state evolutions. Knowing how the IOPT Petri net is supposed to react to each input event we can validate if the simulation is done correctly through the observation of the state evolution.

Chapter 3

Implementation of Industrial Processes on PLCs using Petri Nets

A PLC program implemented by a Petri net with I/O interacts with a system or, in other words, supervises a system. A Petri net typically has inputs at the transitions and outputs at the places. See in Figure 3.1 the arrows *PN actuation* meaning the outputs required to drive the system and *PN inputs* meaning the signals observed in the system and used to drive the Petri net. To run a Petri net it is required a real or a simulated supervised system that can handle the inputs and the outputs that make the interaction possible.

A DES to PLC conversion toolchain started to be developed by H. Gonçalves [15] based on the tools [12] and [13] presented in the Industrial Processes Automation course taught at Instituto Superior Técnico. In the years that followed, the toolchain was improved by J. Meleiro [35], R. Rei [40] and R. Reis [41]. Currently, the toolchain allows converting a Discrete Event Sys-

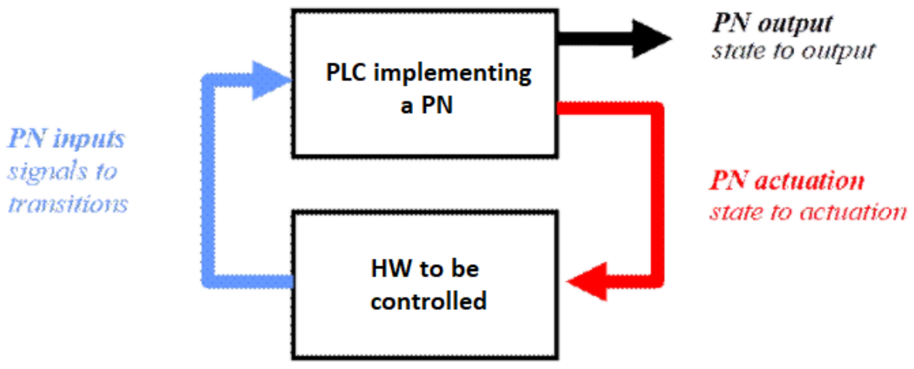


Figure 3.1: Petri net supervising the system *HW to be controlled*.

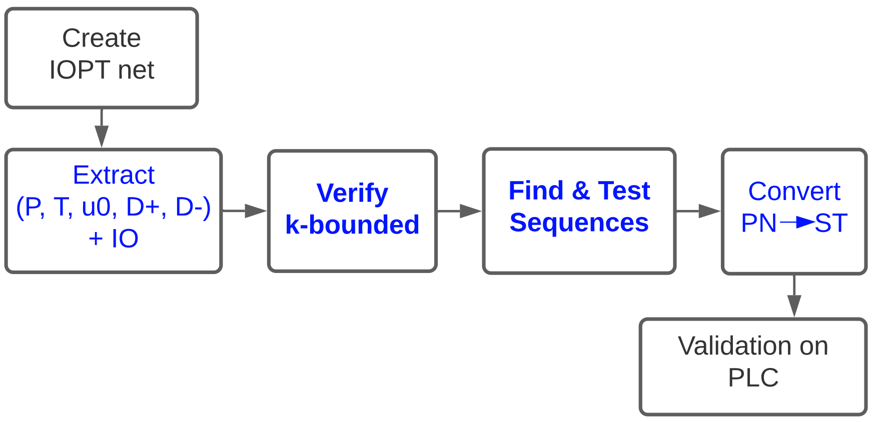


Figure 3.2: IOPT PN to ST Converter. A decision method to reject unbounded PN is added as well as to automatically generate testing sequences.

tem modeled by a Petri net into a PLC Structured Text program, and, in addition, provides a controlled simulation environment.

In this thesis is proposed an automatic generation of PLC-code to implement an industrial process represented by a Petri net, starting by assessing the net is bounded, in order to ensure the states have finite markings which translate directly to finite PLC representations. In addition, are proposed automatic tests to guide the PLC-program (controller) through all its states to confirm they are reachable.

This chapter is divided in two parts. On the first part we describe the DES to PLC conversion toolchain chosen as the object of analysis for this thesis (see Figure 3.2). On the second part we present the alarm system to be supervised.

3.1 DES To PLC Conversion Toolchain

Unbounded Petri nets imply infinite reachable sets, therefore do not allow exhaustive testing of the reachability set. We start by adding a decision method to reject unbounded nets to the PLC-code production toolchain (Figure 3.2). Given a bounded Petri net, we propose a method to automatically generate testing sequences.

3.1.1 Finite Reachable Set

As referred, we want to consider just bounded Petri nets since in those cases one finds finite reachable sets and can automatically generate sequence of events for testing the system. More in detail, we want to start by finding an algorithm to decide whether a Petri net is bounded. We start by the reachability problem introduced in section 2.3.

Given a Petri net C with initial marking μ_0, the reachability problem consists of finding if a state μ is in the reachable set, i.e. $\mu \in \mathscr{R}(C, \mu_0)$. This problem was shown to be decidable [33, 26, 9]. While approaching the reachability problem in [23], Karp and Miller introduced the *node dominance* concept to build a coverability tree. The coverability tree has a finite number of nodes, but may contain the symbol ω indicating unbounded places on the Petri net. A coverability tree not containing ω symbols indicates a bounded Petri net (see section 2.4).

Our proposal, complementing the DES to PLC conversion toolchain , involves two additional steps. The first step is to include a coverability tree construction algorithm in the toolchain to reject unbounded Petri nets, which are associated to coverability trees containing ω. This is introduced in section 2.4 and is implemented in section 4.3.

The second step is to extract the finite sequence of transitions for testing, given a finite reachability tree, i.e. a coverability tree which has no ω symbols as the Petri net did not require its use. We find operation cycles, avoiding therefore deadlock cases, and use sequences of transitions to assess whether the process code effectively reaches all possible states. This methodology is developed in section 4.4.

Before detailing the additional methodologies, we explain the DES to PLC conversion toolchain general steps.

3.1.2 DES Implementation

We assume that the modelling of the DES is made by a system designer. More in detail, we assume the IOPT Petri net properly models the system and that the Petri net meets the format required by the toolchain. The already mentioned IOPT Tools is the only acceptable modelling program, at the moment. This web tool allows the download of the Petri net as a *pnml* extension file, which is the required file format.

We are only interested in bounded Petri nets. Unbounded Petri nets allow infinite length non-cyclic transition firing sequences; it would be impossible to test an DES modeled by an unbounded Petri net to its full extent. It would also be rather difficult to choose the set of most important and sufficient sequences to properly verify and certify each DES independently in an automatic manner, let alone generalize the extraction of such sequences to be independent of the DES. By creating a coverability tree, we can automatically discard Petri nets that originate a tree containing the ω symbol, which means the Petri net is unbounded.

Another reason to focus solely on bounded Petri nets is their direct translation to the PLC programming language *Sequential Function Charts* (SFC) (also known as Grafcet or IEC 60848), which in turn is equivalent and easily mapped to all other PLC programming languages [6], assuming that the ambiguities of the standard are first resolved [1]. In the best case scenario (safe Petri nets), a Petri net place corresponds directly to a step on Grafcet. In the worst case scenario (k-bounded Petri nets), the steps on Grafcet will relate one to one to the reachability tree nodes, i.e. the Grafcet will have as many steps as nodes in the tree.

Before describing the toolchain steps, note that the toolchain offers some options of usage, one of which is worthy of mentioning for the scope of this thesis: the user has the option of setting a simulation that considers timed transitions or normalize all transitions to non-timed.

We present below the general steps of the DES to PLC conversion toolchain , accompanied a block diagram of the complete process to convert an IOPT Petri net to ST and its testing on Unity Pro presented in Figure 3.3. For more detailed information on the functions used in each step, please refer to the master thesis [15, 35, 40, 41] and the tools [12, 13].

Data extraction The first step of the toolchain starts with extracting all the information it requires from the IOPT Petri net prototype and parse it. It produces lists of related information about the Petri net's places, transitions, arcs, input signals, output signals, and determines both the preconditions and postconditions matrices and initial marking matrix.

In this thesis we added three new utilities within this parsing function. Now, it is also responsible for defining timed transitions for simulation as well as attributing a time delay to each type of input. It also attributes initial values to the input signals according to their initial definition in the IOPT Petri net. Finally, this tool can now identify a *storyboard*.

Computing Coverability Tree [Added] The second step of the toolchain is to compute the coverability tree. This is done using the preconditions and postconditions matrices stored as well as the initial marking as input to the coverability tree construction algorithm, which is explained in detail on subsection 4.3. This function implements a version of Cassandra's algorithm for constructing the coverability tree and outputs a structure MP containing all information pertaining said tree (refer to section 4.3 for more detail on MP). The main function uses three subroutines:

- **transition function**, returns the adjusted marking of the new node by checking node dominance and infinity symbol carrying. More detail on its implementation please refer to Table 4.2.

- **add path**, adds a new sequence of transitions fired or nodes achieved on the path from the root node to the current node under evaluation. Returns the updated structure of path sequences.
- **tracker**, returns the updated structure that records essential information to generate the dot file used by Graphviz. This file is used to generate a visual representation of the coverability tree.

Computing Test Sequence [Added] An internal function receives as input the $\boldsymbol{MP}$ structure and extracts all sequences of transitions fired that allow the Petri net starting at a given initial state to return to that same initial state, performing a soft-reset. Then it compiles a table U_t that represents the timeline for the input events to happen. I.e., it assigns chronological timestamps to input signals changes so that the Petri net evolves in accordance to the cyclic sequences of transitions.

Each row in table U_t contains a timestamp on the first column. The other columns are paired consecutively two by two representing the OFF and ON component of each input signal, respectively. On each of these columns, the integers represent the values to be set for each component.

More detail on the computation of both the test sequence and the table of events on section 4.4.

Model Simulation The Petri Net simulator used is an adaptation of the one found in [13]. Note that for the original simulator, the user is responsible for creating the entire testing sequence (see vector `tu`, in the function `filename_IO`, in philosophers example of [13]). For this toolchain, a custom table U_t can be provided by the user.

The simulation of a Petri net consists of two main steps namely (i) the verification that the preconditions are met and (ii) the state evolution after selecting just the possible-to-fire events.

A loop is run where each iteration corresponds to a time value (default is 0.5 seconds). This can be interpreted as the scanning times. The simulation loops over the following actions:

- **Handling Outputs** - updates the timers assigned to the transitions and returns an array containing the information on how to actuate on the hardware for that iteration, given the marking of the places;

- **Enabled Transitions** - finds the transitions that are enabled at the current time of the loop. This is achieved by comparing the transitions enabling conditions against the hardware inputs and the readied timers at the time instance;
- **Filter Possible Firings** - compares the enabled transitions with the marked places from the previous loop iteration, taking into consideration the Petri net's incidence matrix, and finds the transitions that are ready. The program triggers the readied transitions and updates the marking of the places.

ST compiler The method chosen to transcribe Petri net to ST, creating a PLC program from a Petri Net and IO mapping, uses three auxiliary functions to prepare a problem's specific data, like timed transitions and physical inputs or outputs, to be used straightforward by the main script. The auxiliary functions enable the following: (1) Load a Petri Net from file, that contains the incidence matrix and the Petri net's initial marking; (2) Define the mapping of PLC inputs to PN transitions, creating a structure with the PLC inputs and the selected corresponding transitions, and with the conflicts between transitions; (3) Define the mapping of PN places to PLC outputs, creating a structure with the outputs that should change value when the places are marked.

- **PN definition** - loads a Petri Net from a file that contains the incidence matrix and the Petri net's initial marking and splits D into postconditions matrix D+ and preconditions matrix D-. Next, the priority and timed transitions are declared. Finally, all the previous information is stored on a structure, that is the Petri net representation of this program.
- **Input mapping definition** - defines the system inputs that trigger each untimed transition. There are transitions that are triggered due to the conjugation of multiple inputs, or due to the negation of some inputs. Each untimed transition is represented as an entry on the cell array which contains all the input mappings.
- **Output mapping definition** - defines the system outputs that are set to HIGH on each place of the Petri Net. The cell array which contains the output mappings has an entry for each output of the system and an array with the associated places. The available PLCs can have a single IO module or two modules, one for each IO function, and consequently different addresses for IO pins. In order to correctly define those addresses, on this function a pop-up window is shown to allow the user to select the correct PLC configuration.

Presented with the Petri net representation and its input and output mapping, as well as a list of the free PLC output addresses, the toolchain is ready to generate the PLC code. Simply put, it creates n variables on PLC addresses $\%MW2xx$, where n is the number of places of the specified Petri Net and xx the number associated to each place. All those variables are of type integer, and their initial value is 0 or 1, according to the specified initial marking of the Petri Net.

Like is done with all the places, all the transitions timed or untimed, are also declared as variables on PLC addresses $\%MW1xx$ where xx is the number associated to each transition. The value of each one of the *transitions variables*, is the combination of the input signals that trigger each transition converted on an integer, `BOOL_TO_INT`, or the timers output flags. Then the Petri Net is encoded in Structured Text, resulting on a block of code with m `IF` conditions where each `IF` represents a set transition-place. Those conditions will be verified in order by the PLC, to check in which state (place) is the system, on each PLC cycle.

Finally, the function creates the block of code that writes the outputs on the output addresses of PLC, according to its actual state, and outputs a text file with the generated Structured Text code.

Unity Pro simulation given table U_t The simulation of the ST code obtained from the ST compiler with Schneider Unit Pro software takes two ordered steps:

1 **Unity Pro** - we place the ST code on a Unity Pro section, connect the PLC in simulation mode, transfer the project to PLC and run.

2 **Matlab** - we upload the desired table U_t to the `myterminal5` application and run it. `myterminal5` is a digital alarm interface that communicates using MODBUS protocol to Unity Pro, created for the Industrial Processes Automation course taught at Instituto Superior Técnico [11]. This application is responsible to inject the inputs as given by the table U_t into the simulated PLC.

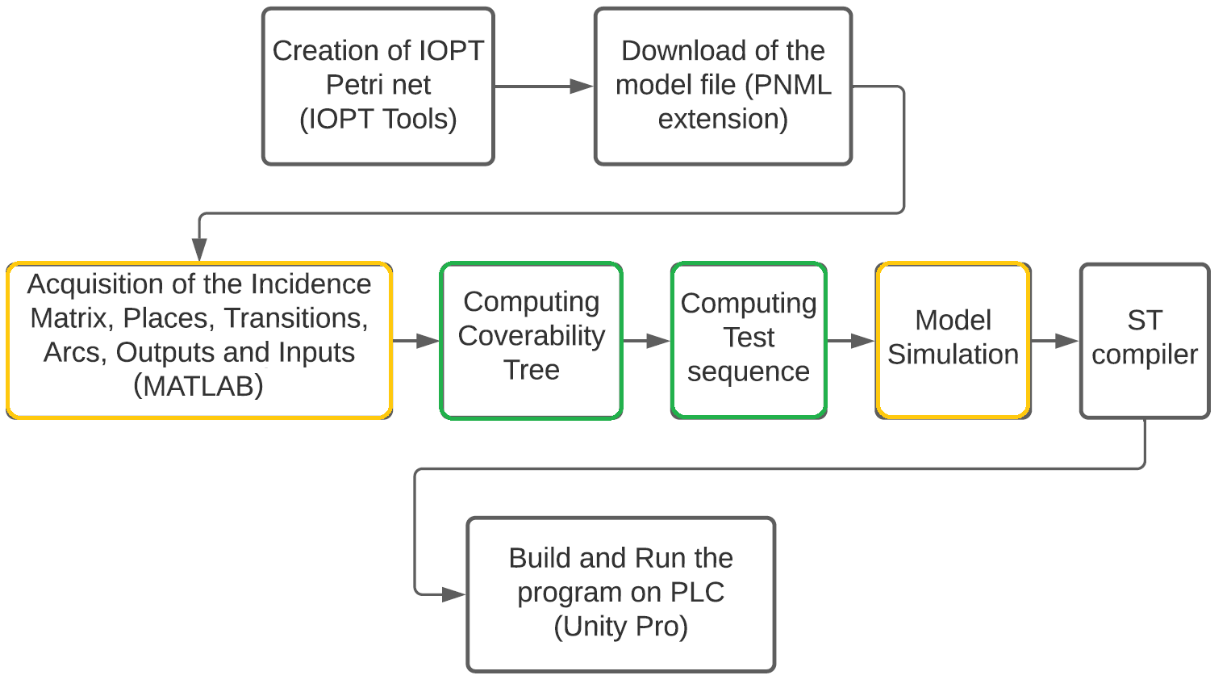

Creation of IOPT Petri net (IOPT Tools)
Download of the model file (PNML extension)
Acquisition of the Incidence Matrix, Places, Transitions, Arcs, Outputs and Inputs (MATLAB)
Computing Coverability Tree
Computing Test sequence
Model Simulation
ST compiler
Build and Run the program on PLC (Unity Pro)

Figure 3.3: A more detailed block diagram of the IOPT PN to ST Converter. The green box represents proposed additions with this thesis. The yellow box represents where main changes were introduced.

3.2 Case Study - Alarm System

The role of the PLC and the world plant, in this case an alarm, is clearly illustrated in Figure 3.4. It shows a mock up of an alarm formed by one PLC and one user input/output terminal.

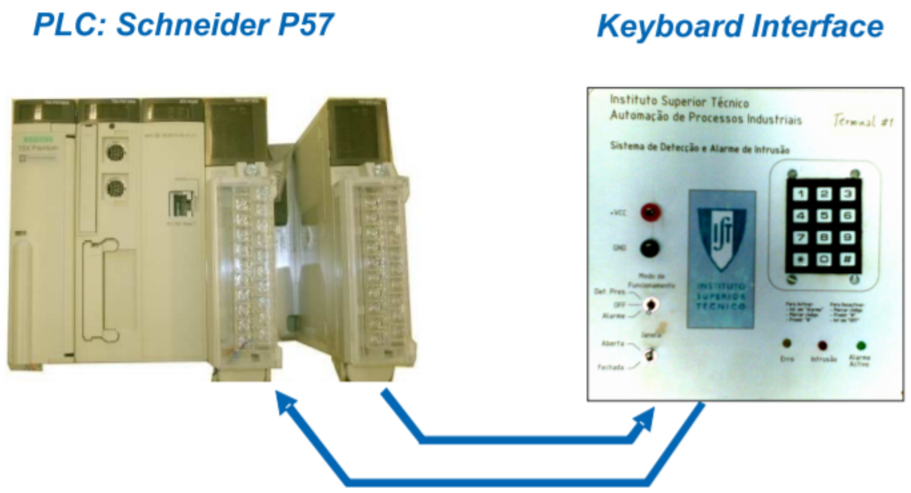

PLC: Schneider P57
Keyboard Interface
Instituto Superior Técnico
Automação de Processos Industriais
Sistema de Detecção e Alarme de Intrusão

Figure 3.4: Alarm system, based in one PLC and one terminal - extracted from [15].

In order to design a Petri net with I/O that interacts with the alarm or, in other words, that supervises the alarm, we need to provide the alarm with the means to drive the Petri net and to be driven by it in return. Three actuators are considered in the alarm to provide input to the Petri net: a presence switch, an alarm switch and a door switch. Regarding the outputs of the Petri net, four signals are observed by the alarm to turn ON or OFF a corresponding red LED, yellow LED, green LED or buzzer. Refer to Table 3.1 for further information on these signals.

3.2.1 Modes of Operation

(Mode 0) Off mode - Initial mode of operation. In this mode, all outputs are OFF. From this mode, the alarm may evolve to Alarm mode by turning ON the alarm switch and to Presence Detector mode by turning on the presence switch. Whenever the two previously mentioned switches are turned OFF the alarm always returns to Off mode.

(Mode 1) Presence Detector mode - This mode serves the purpose of detecting a person entering the room, represented by the opening of the door (turning ON the door switch). It can be thought of has the signaling sound found at the entrance of stores to inform a client has arrived. To enter this mode, turn ON the presence switch while in Off mode. Then, if the door switch is turned ON, the buzzer buzzes for 1 second and then stops, while the yellow LED turns ON for 5 seconds before turning OFF. Turning OFF the door switch will restart the Presence Detector mode while turning OFF the presence switch will return the alarm to Off mode.

Name	Mode	Description
Presence Switch	Input	While in Off mode, if switched ON then the alarm enters the Presence Detector mode. While in the latter mode, if switched OFF then exit back to Off mode.
Alarm Switch	Input	While in Off mode, if switched ON then the alarm enters the Alarm mode. While in the latter mode, if switched OFF then exit back to Off mode.
Door Switch	Input	Represents the opening and closing of the door. It is used in both Presence Detector mode and Alarm mode.
Green LED	Output	Used in Alarm mode to indicate the Alarm has been successfully activated.
Yellow LED	Output	Used in Presence Detector mode and Alarm mode to signal when the door is opened.
Red LED	Output	Used in Alarm mode to signal that the alarm has been set off and alert that there is an intruder.
Buzzer	Output	Used in Alarm mode to signal that the alarm has been set off and alert that there is an intruder. Used in Presence Detector mode to signal the door was opened.

Table 3.1: Inputs and Outputs of the IOPT Petri net that controls the alarm system.

(Mode 2) Alarm mode - This mode functions as an intruder detection alarm (the intrusion is represented by turning ON the door switch). To enter this mode, turn ON the alarm switch while in Off mode. It takes 30 seconds for the alarm to activate so that the user can leave the room. During this time, the alarm can be turned OFF by turning OFF the alarm switch. Also, the opening of the door is not detected during this activation period. After 30 seconds have passed, the green LED turns ON and stays ON until the the alarm returns to Off mode. The alarm is now ready to detect the opening of the door. When the door switch is turned ON, the yellow LED turns ON and after a 5 second delay, both the red LED and the buzzer are turned ON for 2 seconds and OFF for 1 seconds continuously in a square wave-like blinking until the alarm switch is turned OFF, returning the alarm to the Off mode.

3.2.2 Proposed IOPT Petri net model

To model the alarm system, we created an IOPT Petri net, shown on Figure 3.5, based on the work of R. Rei [40].

The marking of each place is given by an integer inside each place, if there is no integer

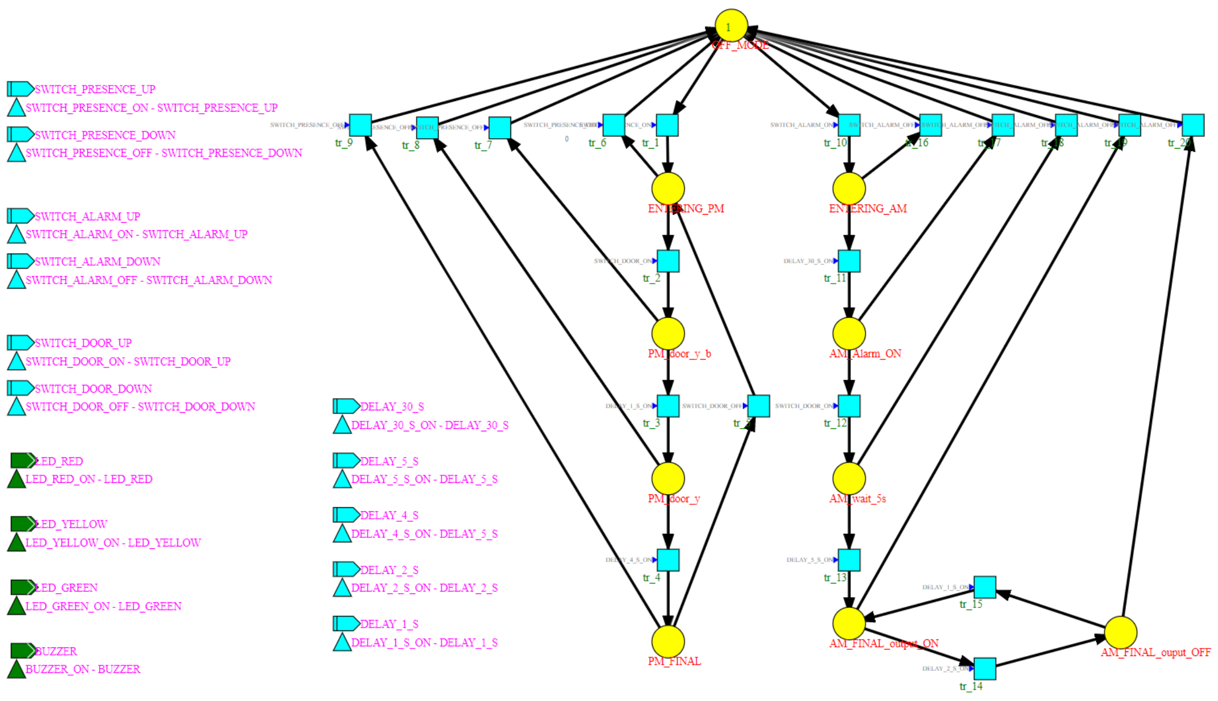


Figure 3.5: Petri net edited with IOPT Tools [16] to control the alarm.

then the marking is 0. Places are represented by yellow circles and transitions by blue squares. There are a total of 10 places and 20 transitions and the definition of each place and transition can be seen in Table 3.2 and Table 3.3, respectively.
With the IOPT Tools, one is able to define input signals, output signals, input events and output events. Input signals are used to obtain information from the external world, for instance, to read the state of sensors, read user interface buttons, or read signals from other systems. Output signals may be used to manipulate mechanical actuators, illuminate LEDs, or send information to other systems. Events are usually associated with changes in signal values. Input events are triggered by changes in input signals and output events will cause changes in output signals. An IOPT controller might wait for specific input events and react accordingly, by changing the value of output signals. There are 10 input signals depicted as blue rectangles (3 switches in ON position + 3 counterpart switches in OFF position + 4 timed inputs), 10 input events corresponding to each input signal drawn as blue triangles, 4 output signal represented as green rectangles (3 LEDs + 1 Buzzer) and 4 output events corresponding to each output signal presented as green triangles. For each proposed signal there is a corresponding event. Signals are used by the places to indicate changes to be applied to the outputs whereas events are used to

Condition Identifier	**Description**
OFF_MODE	Off mode. No output.
ENTERING_PM	First stage of Presence Detector mode. No output.
PM_door_y_b	Second stage of Presence Detector mode. Yellow LED and Buzzer are ON.
PM_door_y	Third stage of Presence Detector mode. Yellow LED is ON.
PM_FINAL	Fourth and final stage of Presence Detector mode. No output.
ENTERING_AM	First stage of Alarm mode. No output.
AM_Alarm_ON	Second stage of Alarm mode. Green LED is ON.
AM_wait_5s	Third stage of Alarm mode. Green LED and Yellow LED are ON.
AM_FINAL_output_ON	Fourth and last stage of Alarm mode together with AM_FINAL_output_OFF place. Green LED, Yellow LED, Red LED and Buzzer are ON.
AM_FINAL_output_OFF	Fourth and last stage of Alarm mode together with AM_FINAL_output_ON place. Green LED and Yellow LED are ON.

Table 3.2: Identification of DES Conditions (Places).

Event Identifier (tr_)	**Description**
1	Fires when the presence switch is ON.
2	Fires when the door switch is ON.
3	Fires after 1 seconds (timer controlled transition).
4	Fires after 4 seconds (timer controlled transition).
5	Fires when the door switch is OFF.
6, 7, 8, 9	Fires when the presence switch is OFF.
10	Fires when the alarm switch is ON.
11	Fires after 30 seconds (timer controlled transition).
12	Fires when the door switch is ON.
13	Fires after 5 seconds (timer controlled transition).
14	Fires after 2 seconds (timer controlled transition).
15	Fires after 1 seconds (timer controlled transition).
16, 17, 18, 19, 20	Fires when the alarm switch is OFF.

Table 3.3: Identification of DES Events (Transitions).

enable transitions by providing changes on the corresponding signal.

The incidence matrix D corresponding to the Petri Net shown in figure 3.5 is:

$$D = \begin{bmatrix} -1 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \\ 1 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{3.1}$$

and the initial marking is:

$$\mu_0 = \begin{bmatrix} 1\,0\,0\,0\,0\,0\,0\,0\,0\,0 \end{bmatrix}^T \quad . \tag{3.2}$$

Both D and μ_0 are obtained automatically from the DES to PLC conversion toolchain . To be noted that the chosen initial marking means the alarm initiates in the Off mode.

In the proposed Petri net, each state is represented by the corresponding marked place. Since only 1 token is necessary to mark a place, at each state, all arcs have the same weight equal to 1 so no new tokens are created.

3.2.3 Properties Study

This subsection presents a brief study of the proposed Petri Net properties already introduced in the subsection 2.2 of the Background chapter.

The coverability tree is obtained by the coverability tree algorithm previously proposed and is graphically represented with the aid of Graphviz in Figure 3.6. We can observe that there are no ω infinite symbols in the tree and thus it is a coverability tree representing a **finite reachability set** with 10 possible states.

The proposed Petri net is **safe** since all places can only be marked either with 1 or 0. This means that it is also **1-bounded**. We can see that for every marking the number of tokens remains the same, therefore the Petri net is **strictly conservative**. Regarding transition liveness, we conclude all transitions are **level-4 live** and the Petri net is free of both deadlock and livelock.

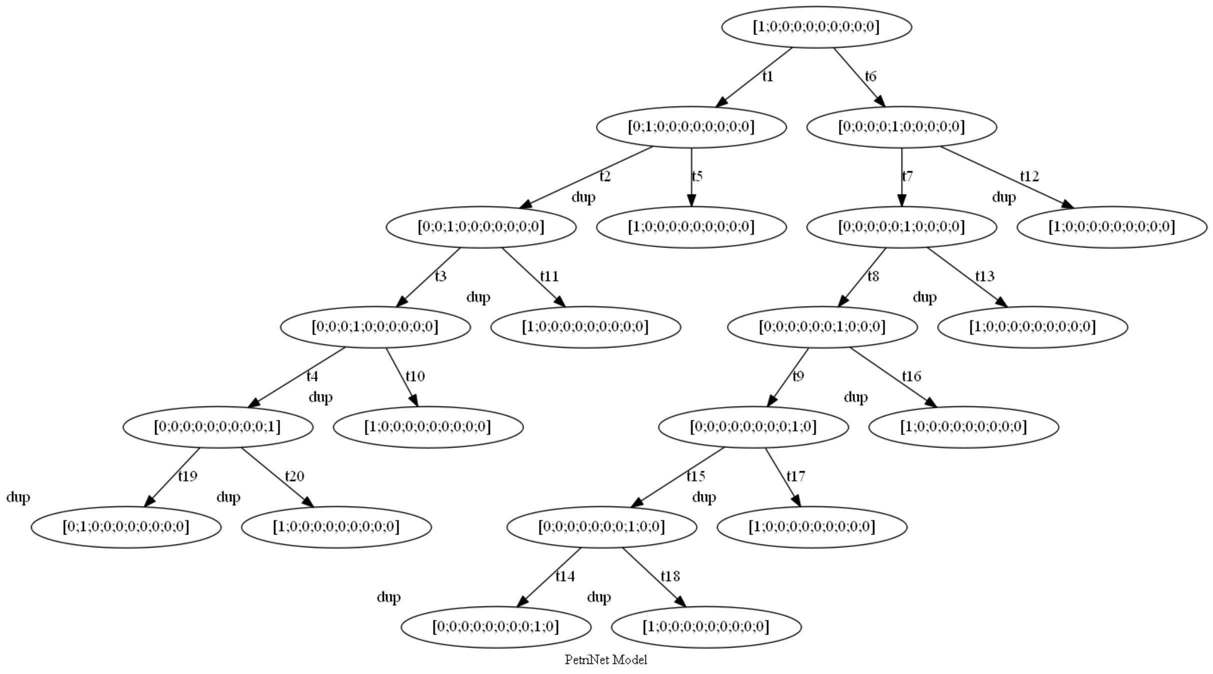


Figure 3.6: Coverability Tree for the proposed Alarm Petri net (Duplicate states are marked "dup" on the top left corner).

No state covers another state and so the Petri net does not present the coverability property. The Petri net is time-invariant since for any reachable marking there exists at least one transition sequence to go from that same marking to itself, thus allowing a software reset.

Chapter 4

Validation of Industrial Processes based on Petri Nets

In this chapter, we explain our implementation of C. Cassandras' algorithm to create coverability trees and propose a process to generate event sequences that allow testing discrete event systems.

4.1 System Validation based on the Coverability Tree

The reachable set of a Petri net, whenever finite, can be displayed as a comprehensive list. However, the graphical display of the reachable set as a tree is more convenient in various applications. It allows the immediate observation of paths from the initial state to any other state, as well as checking for cycles or deadlocks in a visual way.

States are represented by the nodes and transitions by the edges, therefore when one wants to obtain the input sequence needed to transition from the initial state to any other node all that is needed is to get the sequence of transitions that lead to the desired node. Such advantage is used to automatically generate the event sequence for DES validation testing. Although an algorithm is presented in [2], no original code was found, and with such advantage in mind a slightly modified algorithm is proposed in Table 4.1 and implemented in MATLAB, as shown in Appendix D. The two modifications made to Cassandras and Lafortune's algorithm take place in **step 2.2** and its **substeps**.

4.2 Petri net Graphs and Coverability Trees

Despite using in the thesis, and in the associated programming, literal representations for Petri nets and trees, both the Petri nets and the trees are shown also graphically.

A Petri net is formally defined as a five-tuple $C = (P, T, A, w, \mu_0)$, as introduced in section 2.2. The five-tuple, by itself, contains no graphical representation details. The Petri net can however be interpreted as a weighted bipartite graph, the two parts being places and transitions. As the graphs, trees are collections of nodes, the difference being trees have no cycles. This means we can convert a tree to a graph. Graphs can be represented graphically in two dimensional plots by accepting that in some cases arcs may intersect. A similar reasoning as performed for graphs can be used to state and obtain graphical representations of both Petri nets and trees.

Graphical representations of graphs and trees can be generated automatically using the Graphviz software [8]. Graphviz implements a four-pass algorithm for drawing directed graphs. The first pass finds an optimal rank assignment using a network simplex algorithm. The second pass sets the vertex order within ranks by an iterative heuristic incorporating a novel weight function and local transpositions to reduce crossings. The third pass finds optimal coordinates for nodes by constructing and ranking an auxiliary graph. The fourth pass makes splines to draw edges.

Some Petri nets created without graphical editors and all coverability tree shown graphically in this thesis were computed with Graphviz. The symbol indicating an infinity marking count for a place, ω, is represented by "Inf" to ease the generation of the DOT language file.

4.3 Coverability Tree Construction

The construction of a coverability tree is made possible due to the combination of the infinity symbol ω and the concept of node dominance (See Notation in section 2.4). For a better understanding of this concept, the following subsection presents an example of the construction of a coverability tree that verifies node dominance. Then we present the general steps for our implementation of the algorithm that builds the coverability tree.

4.3.1 Node Dominance

It is possible to build a coverability tree by applying the concept of *node dominance* (see Notations in section 2.4). For each state, node dominance is established taking into consideration

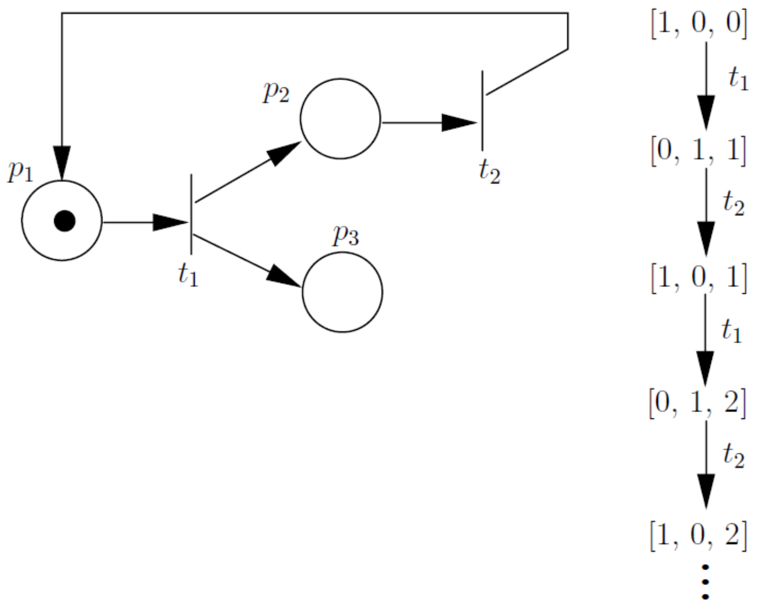


Figure 4.1: Unbounded Petri net (left) and part of its infinite reachability tree (right).

only the nodes on the path from the root node to the currently considered node.

Consider the Petri net of Figure 4.1, with initial state $[1, 0, 0]$. The reachability tree starts out with this state, from which only transition t_1 can fire. As shown in Figure 4.1, the next state is $[0, 1, 1]$. Now only t_2 can fire, so we create one branch, with corresponding next state $[1, 0, 1]$. The new branch allows transition t_1 to fire, resulting in new state $[0, 1, 2]$. Once again, only t_2 can fire, and the corresponding next state is [1, 0, 2]. This is sufficient to detect the pattern of this tree. Every branch repeats itself, but with one extra token in place p_3. In other words, we see that place p_3 is unbounded, meaning that the reachability tree is infinite, since the set of reachable states is infinite.

However, we do not build this infinite reachability tree. Rather, we build a coverability tree. We regress in the tree and observe that state $[1, 0, 1]$ is larger, component-wise, than state $[1, 0, 0]$ that precedes it in the path from the root to $[1, 0, 1]$, i.e. we say that state $[1, 0, 1] \ >_d \ [1, 0, 0]$ and replace the marking on p_3 with ω. Therefore we replace state $[1, 0, 1]$ by $[1, 0, \omega]$. The ω symbol propagates to states reachable from $[1, 0, \omega]$. Next, we fire t_1, resulting in state $[0, 1, \omega]$. This new state dominates $[0, 1, 1]$ in the path from the root to $[0, 1, \omega]$. However, the symbol ω has already been used in place p_3. We continue with transition t_2, which leads to the duplicate state $[1, 0, \omega]$. This completes the construction of the coverability tree, the result presented in Figure 4.2.

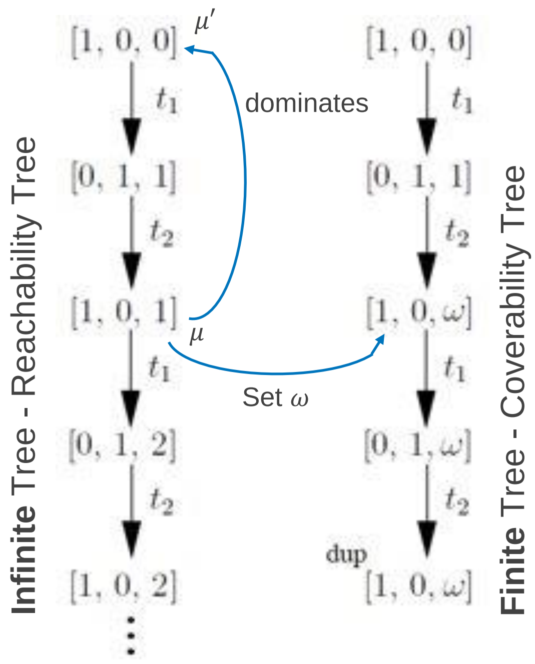


Figure 4.2: Infinite Reachability Tree (left) and corresponding Coverability Tree (right), for Petri net in Figure 4.1. When a state μ dominates a state μ' in the path from the root node to μ set the marking of the places that verify both conditions of dominance in state μ to ω.

4.3.2 Algorithm General Steps

The implementation of the coverability tree algorithm is based on two variables. The first variable, $\boldsymbol{R}$ is a matrix of reachable states. Each column vector of $\boldsymbol{R}$ represents the marking of a state. The second variable, $\boldsymbol{MP}$ is formed by three parts, $\boldsymbol{P}$, $\boldsymbol{S}$ and $\boldsymbol{M}$.

$\boldsymbol{P}$ is a matrix where each entry is a path (n-tuple of transitions) to reach a reachable state from the initial state. Reachable states with more than one path (i.e. with rows with more than one non-empty value) means that state has duplicate nodes in the tree.

$\boldsymbol{S}$ is a matrix where each entry is a path along the states (n-tuple of state indexes) listing the states passed-through to reach a reachable state from the initial state. Reachable states with more than one path, i.e. with rows with more than one non-empty value, means that state has duplicate nodes in the tree.

$\boldsymbol{M}$ (`tracker`) is a matrix where each row is a 4-tuple (state, fired transition, next state, information if next state is "duplicate" or "new" in $\boldsymbol{R}$). This structure is useful only for the Graphviz functions.

The mathematical implementation of the algorithm in Table 4.1 and the generation of the next state abiding by node dominance in Table 4.2. Implementation of both can be seen on

Appendix D as well the function used to update MP structure.

4.4 Generation of Event Sequences

The testing event sequence is obtained from the sequence of cyclic transitions which in turn is derived from the coverability tree by taking each sequence of fired transitions from the root node to each duplicate root node. The transition sequences are obtained as ordered in P for the root node. This extraction is better explained through an example.

Consider the simple bounded IOPT Petri net C presented in Figure 4.3 along its coverability tree. The green nodes are the duplicate root nodes.

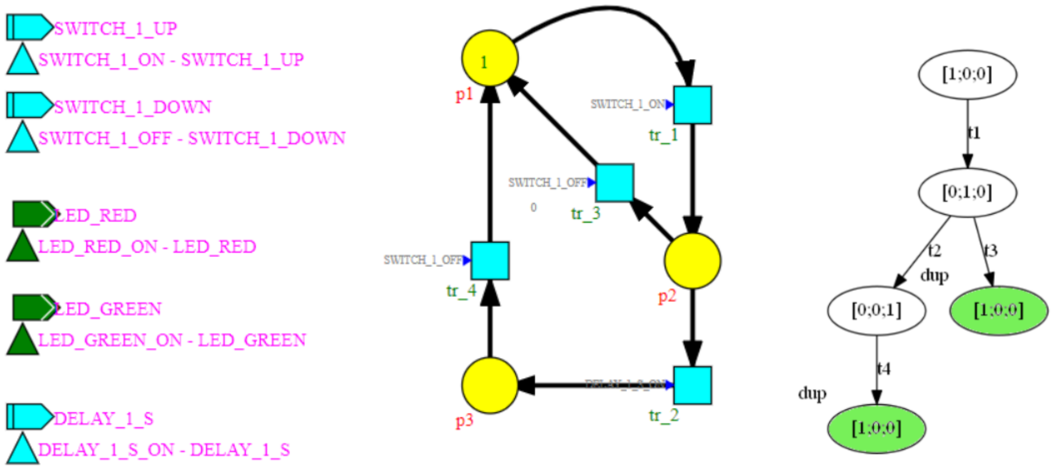


Figure 4.3: IOPT Petri net C (middle), IO signals and events (left) and Coverability Tree (right).

After running the coverability tree construction algorithm for Petri net C we get the MP structure, presented in Figure 4.4.

Coverability Tree Algorithm - Implementation

Input: Preconditions matrix $\boldsymbol{D^-}$, post-conditions matrix $\boldsymbol{D^+}$ and initial marking $\boldsymbol{\mu_0}$.

Output: Reachable markings $\boldsymbol{R}$, Structure $\boldsymbol{MP}$.

```
$\boxed{\textbf{Step 1}}$ - Initialize $R = \{\mu_0\}$, $P = \emptyset$, $S = \emptyset$, $M = \emptyset$.
$\boxed{\textbf{Step 2}}$ - Let $R = \{\mu_0,\ \mu_1, ...,\ \mu_N\}$, $N$ be total number of reachable states found so far,
$i = 1, ..., N$ and $D_t^-$ represent column vector $t$ of $D^-$.
For each state $\mu_i \in R$ generate vector of enabled transitions as
    $x : (\mu_i \geq D_t^- \to 1) \wedge (\neg(\mu_i \geq D_t^-) \to 0)$.
    $\boxed{\textbf{Step 2.1}}$ - If $x(t) = 0, \forall t$, then $\mu_i$ is terminal node.
    $\boxed{\textbf{Step 2.2}}$ - Else for each $t : x(t) \neq 0$ create a new node $\mu'$ mapped by transition
function $f$ defined in Table 4.2.
Let $p_n$ be the set of sequences of transitions to $\mu_n$ and $p_{n_j}$ be the sequence $j$. Let $s_n$ be
set of sequences of states indexes in $\boldsymbol{R}$ to $\mu_n$ and $s_{n_j}$ be the sequence $j$. All sequences are
initially empty.
        $\boxed{\textbf{Step 2.2.1}}$ - If $\exists \mu_n \in R : \mu_n = \mu'$, then $\mu'$ is duplicate node. Do:
            $p_n \leftarrow p_{i_1} \cup \{t\}$,
            $s_n \leftarrow s_{i_1} \cup \{i\}$,
            $M \leftarrow M \cup \{(\mu_i, t, \mu_n, "duplicate")\}$.
        $\boxed{\textbf{Step 2.2.2}}$ - Else $\mu'$ is new node. Do:
            $\mu_{N+1} = \mu'$,
            $p_{N+1} \leftarrow p_{i_1} \cup \{t\}$ and $P \leftarrow P \cup \{p_{N+1}\}$,
            $s_{N+1} \leftarrow s_{i_1} \cup \{i\}$ and $S \leftarrow S \cup \{s_{N+1}\}$,
            $M \leftarrow M \cup \{(\mu_i, t, \mu_{N+1}, , "new")\}$,
            $R \leftarrow R \cup \{\mu_{N+1}\}$,
$\boxed{\textbf{Step 3}}$ - If all new nodes have been marked as either terminal or duplicate nodes, then
stop.
```

Table 4.1: Coverability Tree Algorithm Implementation.

Transition Function Algorithm - Implementation

Input: Reachable markings $\boldsymbol{R}$, index $\boldsymbol{i}$ of node under evaluation, incidence matrix $\boldsymbol{D}$, enabled transition $\boldsymbol{k}$ under evaluation, cell matrix $\boldsymbol{S}$.

Output: Next reachable state $\boldsymbol{\mu'}$ from state $\boldsymbol{\mu_i}$ by firing transition $\boldsymbol{k}$.

$\boxed{\textbf{Step 1}}$ Initialize $\mu' = \mu_i + D \cdot q_k$, where q_k is a unit vector ($q_k = (0, ...0, 1, 0, ...0)^T$, kth entry is 1). To be noted that all infinite symbols ω in μ_i propagate to the same places in μ' obeying to the operation rules applied to the infinity symbol as presented in Notation 2 in section 2.4 (MATLAB solves the ω operations on its own, else an additional step would be needed to correct the propagation).

$\boxed{\textbf{Step 2}}$ Let s_i be the set of sequences of states indexes of μ_i and s_{i_j} be the sequence j. For each s_{i_j}, check state dominance as presented in Notation 3 in section 2.4 by doing:

$\boxed{\textbf{Step 2.1}}$ - For each state index $p = s_{i_j}(m)$:

$\mu_y = R(p)$,

If $\mu' >_d \mu_y$, then $\forall a : \mu'(a) > \mu_y(a)$ set $\mu'(a) = \omega$.

$\boxed{\textbf{Step 3}}$ Otherwise, μ' is as obtained in $\boxed{\textbf{Step 1}}$.

Table 4.2: Transition Function Implementation.

$$\begin{bmatrix} [1\ \ 3] & [1\ \ 2\ \ 4] \\ 1 & \\ [1\ \ 2] & \end{bmatrix} \qquad \begin{bmatrix} [1\ \ 2] & [1\ \ 2\ \ 3] \\ 1 & \\ [1\ \ 2] & \end{bmatrix} \qquad \begin{bmatrix} [1;0;0] & & & \\ [1;0;0] & 1 & [0;1;0] & 'new' \\ [0;1;0] & 2 & [0;0;1] & 'new' \\ [0;1;0] & 3 & [1;0;0] & 'dup' \\ [0;0;1] & 4 & [1;0;0] & 'dup' \end{bmatrix}$$

(a) (b) (c)

Figure 4.4: The $\boldsymbol{MP}$ structure: (a) P cell matrix, (b) S cell matrix, (c) M cell matrix.

The generation of the event sequence is divided in two main parts: (1) concatenation of the testing sequences and (2) generation of table U_t.

(1) Testing Sequence It is extracted all transition sequences that drive the Petri net from its initial state back to itself. These sequences are found on the first row of the P cell matrix in $\boldsymbol{MP}$. They are concatenated orderly as found from left to right. Then all alternative paths that are not included in the transition sequences found on the first row that lead to other states are completed with the transition sequences that make the Petri net evolve from these nodes back to the initial node.

These sequences can be extracted either from the Petri net to be supervised or from a Petri net that tells a story, called a storyboard, from which the sequence of events is extracted. We established that the storyboard is created in the same IOPT model as the alarm modelling Petri net. Additional code was added to detect a storyboard and present it separately from the system. Any event sequence extracted is then converted into a table that orders the events in a time line. For such a table we use the notation Table U_t.

(2) Table U_t Each transition in the sequence generates a row in the table. Each row is comprised by a timestamp followed by integers representing the values for each existing input at the given timestamp. This means the table contains as many rows as transitions in the complete sequence, plus two extra rows, a first row that sets up the initial values of the inputs and a last row that turns OFF all inputs.

Due to the way the IOPT Petri net is built and parsed there exists 2 parts for the same input: one ON part and one OFF part. Both are represented on the table. The creation of the table is done so that whenever the input is ON then the ON part takes the value 1 and the OFF part takes the value 0 on the same instant. The inverse happens when the input is turned OFF. So, the table contains as many columns as the number of available inputs and their ON/OFF parts of the Petri net, plus one for the timestamp. If the option "untimed" is chosen, then all input events associated to a transition are set to occur 1 second apart from the inputs associated with the previous transition. If the option "timed" is chosen, then all input events associated to a transition are set to occur right after inputs associated with the previous transition of after the corresponding timer times out (refer to the subsection 4.5 on Extension of Petri net model to include Time).

The proposed Table U_t (4.1) was created from the IOPT Petri net presented in Figure 4.3.

$$U_t = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 2 & 1 & 0 \\ 3 & 0 & 1 \\ 4 & 0 & 1 \\ 5 & 1 & 0 \end{bmatrix} \tag{4.1}$$

Since the IOPT Petri net has only one token and no new tokens can be created, with three places we expect the Petri net can only reach three states. There are only two possible cyclic sequences to go from the initial state to itself, so we expect two returns to the initial state.

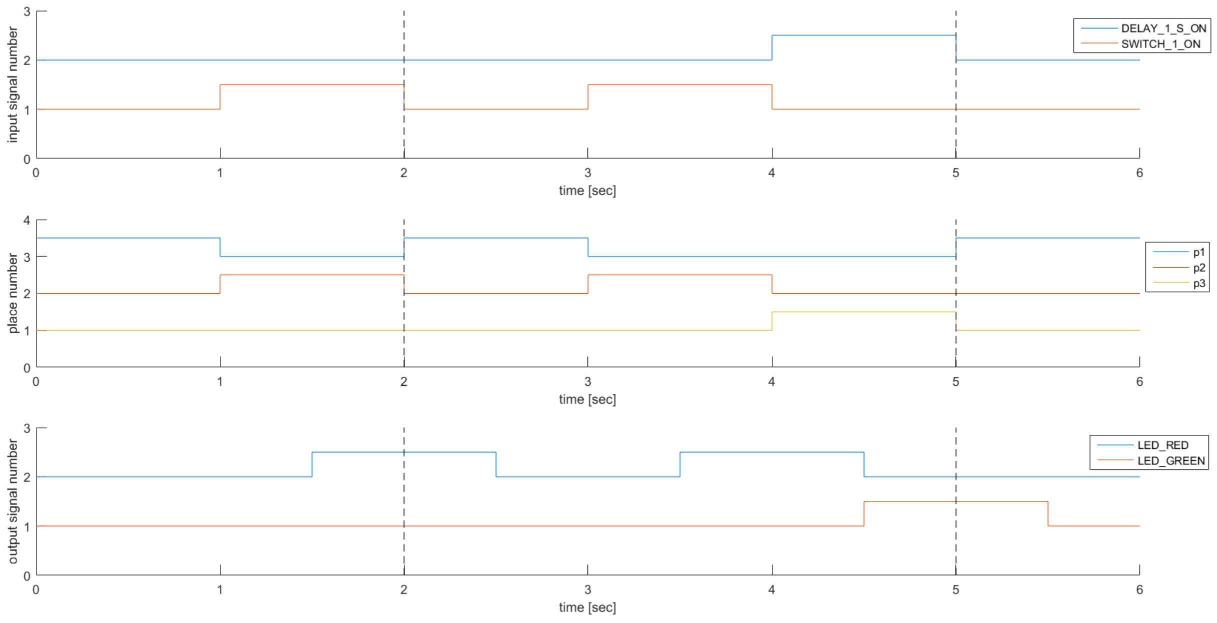


Figure 4.5: Simulation results.

By simulating the same IOPT Petri net with Table U_t (4.1) we obtain the results presented in Figure 4.5. We observe that all states are reached and two returns to the initial state occur following the correct and expected state evolution. These returns are marked by the vertical dashed lines.

4.5 Extension of Petri net models to include Time

Considering the objective of simulating real systems encompassing timeouts, it is necessary an extension to Petri net models with the intent to introduce the notion of timing in the DES to PLC conversion toolchain . Following along the lines of Timed Petri Nets [45], which are well known extensions of Petri net, the extension consists of adding timed transitions, i.e. transitions controlled by timers.

In implementation terms, the extension is made by expanding the data structure, that stores an IOPT Petri net, to include time related parameters on the input signals fields. One information is the activation time duration for timed inputs, and to include a timer representation associated to each transition. In detail, each input now has associated a time value that represents the delay it needs to be recognized as active, pressed or even timed out.

This takes us to the new information added to each transition regarding time. It represents the timestamp at which the associated timer finishes. These parameters are updated within the simulation as necessary and are compared with the current time in each iteration to check which timers are finished, which allows deciding which timed transitions are ready.

To be noted that within the simulation a timer is started when its corresponding transition is enabled and it is stopped when the transition no longer verifies its preconditions. The activation of a timer is done by setting the transition's time value to current time t + associated timer value. Consider the quick example of a timed transition `t1` with a 5 second timer associated as its only input. If `t1` is enabled at time $t = 1$ seconds then the transition's time value is set to $1 + 5 = 6$ seconds. If at time instant $t = 6$ seconds `t1` is still enabled (and by this we mean its preconditions did not change within the time interval $[1, 6]$ seconds), since the timer is finished and the transition is enabled then for that iteration `t1` can be fired.

Chapter 5

Experiments

In this chapter are combined the tools introduced in chapter 3 for creating DES supervisors running on PLCs with the tools introduced in chapter 4 for verifying and validating the created DES supervisors. The main objective of the experiments is the assessment of the validation tools applied to identifying design or implementation problems, and testing the created DES supervisors.

Three use cases are considered based on the discrete event system that models the alarm system detailed in section 3.2. The alarm system is based on a PLC reading alarm inputs and sensors and generating alarm outputs. Each use case involves one or more experiments.

The first experiment consists on checking if the controller allows reaching all the desired states, i.e. test if each state of the reachable set can be reached.

In the second experiment, the controller is run together with a real system. In particular are considered timings of the real world. Petri nets extended with timed transitions are used both for modelling the supervisors and the real world systems.

The third experiment considers typical hardware constraints, more specifically the *input-steady reading times* of a PLC and inputs debouncing.

5.1 Use Case 1 - Reach All Possible States

The toolchain detailed in chapter 3 allows converting a high level DES design (PN) to an implementation running on a PLC or just as a simulation. The question in this use case is where the implementation matches well, or not, the high level design. In other words, we are assessing whether the conversion toolchain performed well, or not, the conversion.

Being the alarm a case study, its functionalities and behaviour have been fully explored

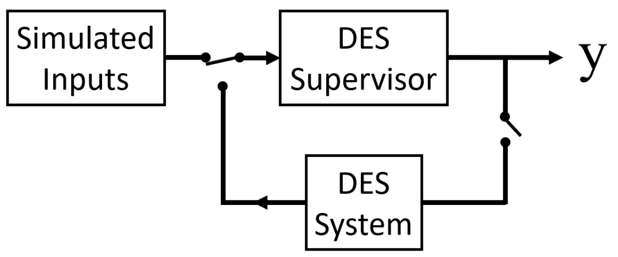


Figure 5.1: Illustration of use case 1 testing. The IOPT Petri net of the DES supervisor reacts only to the given input sequences, disregarding any possible interaction with the system it should supervise.

and so a flawless IOPT Petri net controller of the alarm can be created. We use the event sequence automatic generation tool to generate all possible cyclic event sequences that make the alarm controller reach all of its states. We perform the test to the simulation software by forcing the input values at given time instants according to the generated sequence of events. In other words, we test if each state of the reachable set can be reached. This is always possible because we do not consider the limitations of a real system which introduces situations such as functional deadlocks.

Knowing how the alarm responds to each given input we can use it to validate if the toolchain simulation environment is implemented correctly by comparing the simulation results to the expected ones. If anything is different than expected it is solely due to the toolchain. Otherwise, we can validate the success of the conversion from an IOPT Petri net to the toolchain representation of the IOPT Petri net.

Figure 5.1 illustrates the test we are performing, we feed the simulated inputs to the IOPT Petri net and evaluate the resulting state y. The test depends solely on customized sequences of inputs, ignoring all interactions with the system to be controlled.

General considerations on the creation of input sequence:

- inputs associated to two consecutive transitions have a 1 second delay between, as defined by the toolchain;

- all timed transitions are converted to untimed transitions, the toolchain ignores any added delay beside the one in previous statement;

- the sample frequency used is 0.5 seconds, meaning for each iteration there will be a delay of half a second between the moment an input is recognized and the moment the output values are updated;

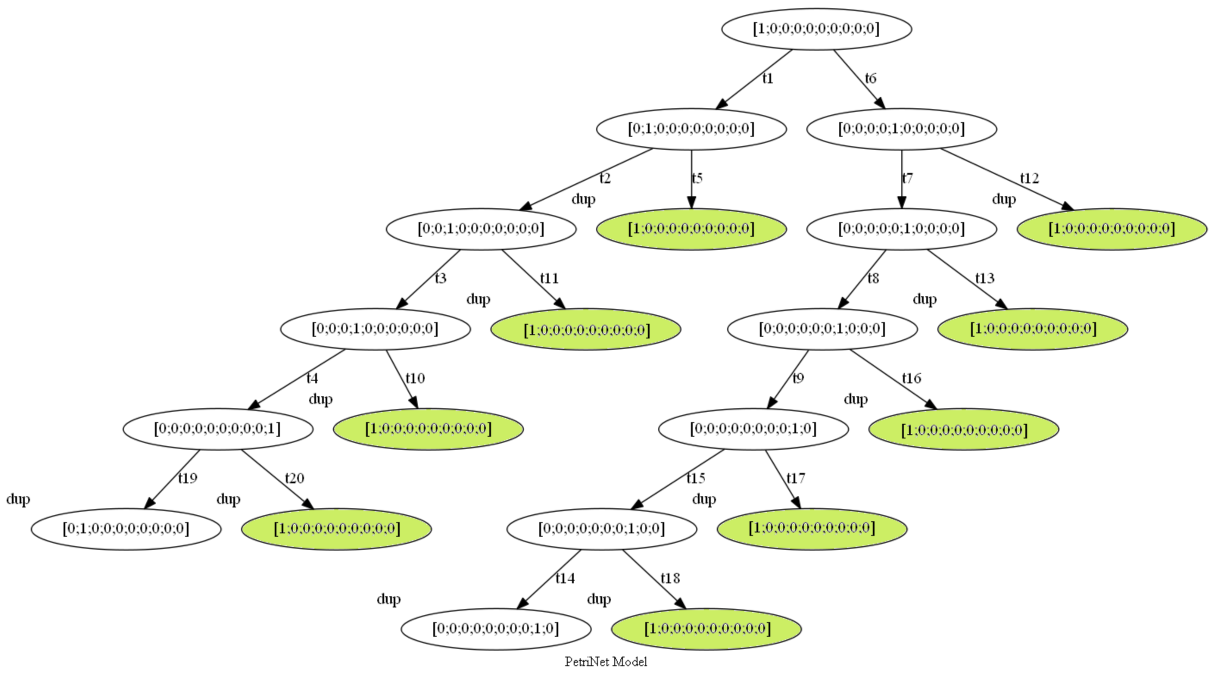


Figure 5.2: Coverability Tree of the proposed Petri net (PN) to supervise the alarm.

- inputs are reset to their starting values once the supervisor returns to its initial state, creating a soft reset.

The considered Petri net in Figure 3.5 produces the coverability tree shown in Figure 3.6. We replicate Figure 3.6 and compliment it into Figure 5.2 by locating the duplicate nodes and identifying the cyclic sequences of transitions.

In Figure 5.2 duplicate nodes are marked with *dup*. Duplicated nodes representing the initial state (*root node*) are colored green and are the nodes we consider to find cycles and visit the complete reachable set. We observe that the Petri net has no terminal nodes, therefore the Petri net has no deadlocks.

The sequence of operation cycles we consider to visit the complete reachable set is derived from the tree by taking each sequence of fired transitions from the root node to each duplicated root node (green nodes in Figure 5.2). They are taken orderly from left to right, top to bottom as

$$\begin{aligned}&(t_1,\ t_5),\ (t_6,\ t_{12}),\ (t_1,\ t_2,\ t_{11}),\ (t_6,\ t_7,\ t_{13}),\ (t_1,\ t_2,\ t_3,\ t_{10}),\ (t_6,\ t_7,\ t_8,\ t_{16}),\\ &\qquad (t_1,\ t_2,\ t_3,\ t_4,\ t_{20}),\ (t_6,\ t_7,\ t_8,\ t_9,\ t_{17}),\ (t_6,\ t_7,\ t_8,\ t_9,\ t_{15},\ t_{18}).\end{aligned} \tag{5.1}$$

These cycles of operation can be converted to a table U_t:

$$U_t = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 2 & 0 & 0 & 1 & 0 & 0 & 0 \\ 3 & 0 & 1 & 0 & 0 & 0 & 0 \\ & & & \vdots & & & \\ 34 & 1 & 0 & 0 & 0 & 0 & 0 \\ 35 & 1 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}. \tag{5.2}$$

Simulating the alarm supervisor IOPT Petri net with table U_t 5.2 yields the results observed in Figure 5.3. Note the vertical dashed lines, they represent a system soft reset, which means the system as returned to its initial state.

Observing the top plot and referring to Table A.2 to associate the transitions in the sequence to an input event, we can check that the input sequence is run correctly, properly stimulating the IOPT Petri net to reach all possible reachable states in a cyclic manner.

The middle plot represents the places marking. The current state corresponds to the marked place at the given time. Following the plot horizontally we can observe the state evolution. We confirm that the IOPT Petri net evolution is correctly simulated and all reachable states are reached.

The bottom plot shows the output values of the supervisor given the current state, with a 0.5 second delay as per specification of the toolchain. All output values are as expected at each current state.

Confirming that the IOPT Petri net reaches all states of the reachable in the expected sequence, we conclude that the DES to PLC conversion toolchain under validation properly extracts the input sequence that drives a given IOPT Petri net through all its reachable states. The toolchain correctly simulates the IOPT Petri net state evolution and represents the matching output values at all reachable states.

For information regarding implementation and/or modification of the toolchain functions, please refer to Appendix A.

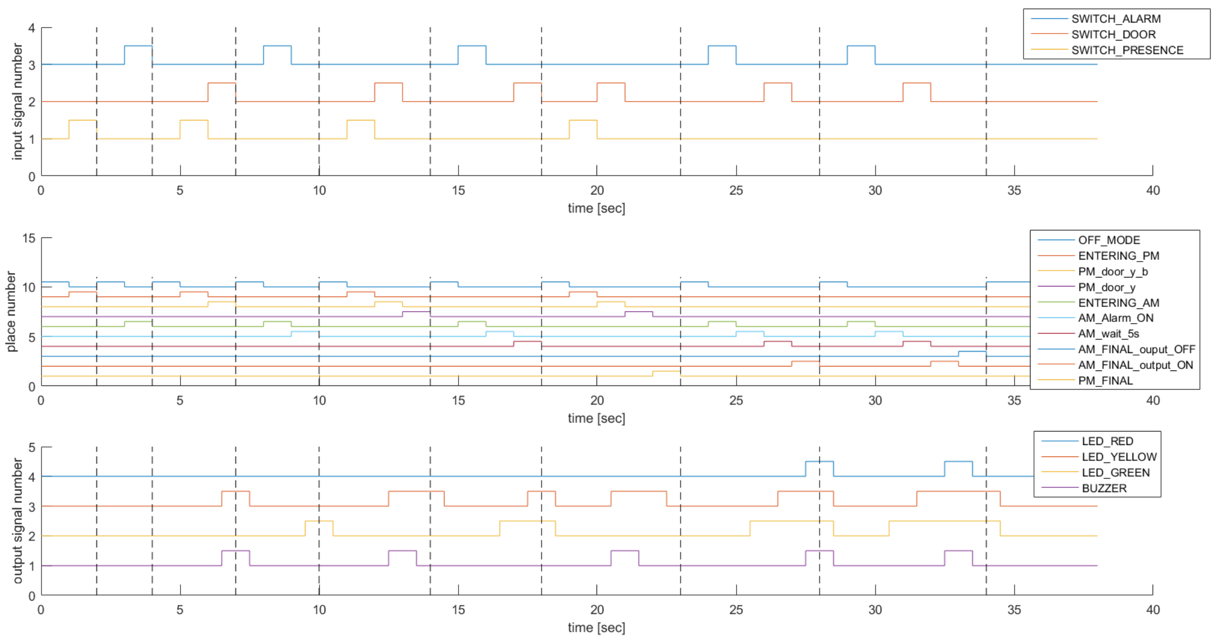


Figure 5.3: Behaviour of alarm supervisor IOPT from Figure 3.5 given the sequence of events extracted from its coverability tree.

5.2 Use Case 2 - PLC-DES interaction with World-DES

In this second use case we introduce two terms: PLC-DES and World-DES. The former refers to the IOPT Petri net modeled DES that is implemented on the PLC, i.e. the controller/supervisor. The latter names the DES modelling a real system, external to the PLC, i.e. the system to be controlled/supervised.

Testing a PLC-DES to the full extent, i.e. visiting all its possible states, would be a way to assess its robustness. However, validating the PLC while including the to/from world interaction may not be feasible: it may require much human intervention or the World-DES may introduce functional deadlocks or even induce unexpected behaviour in the PLC-DES. In this vein, one may consider focusing on specific tests, one may use storyboards covering the expected usages.

Considering the alarm supervisor IOPT Petri net as the PLC-DES and the alarm system as the World-DES, we present this second use case where the PLC-DES works together with a World-DES represented by a storyboard. In other words, the PLC-DES works with the timings of the real world. We incorporate the notion of time into the Petri net model and the testing environment through timers and timed transitions, going along the lines of Timed Petri Nets [45], which is allowed through the extension of the representation of a Petri net. The extension implementation details can be seen on Subsection 4.5.

Implementations details regarding use case 2 are provided in Appendix B.

5.2.1 Closed Loop Simulation, PLC and World Interaction

Two ways can be considered to create a storyboard. The first possibility is to create an IOPT Petri net that tells a story, from which the sequence of events is extracted. The second possibility is to directly provide a custom made sequence of events in the form a table of input events implemented with time tags (see Section 4.4).

The first way is more user-friendly as it shows the flow of events by following the laid out path of the storyboard. Consider the storyboard presented in Figure 5.4 to be the typical correct functioning of the alarm in Alarm mode. Information on timer inputs necessary for the proposed story is given by Table B.1, whereas the places and transitions details are given by Table B.2 and Table B.3, respectively.

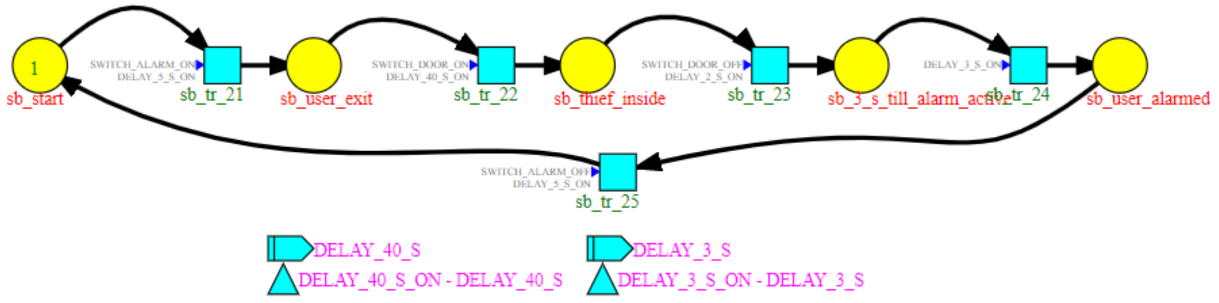


Figure 5.4: Storyboard IOPT proposed for Use Case 2.

The chosen story to showcase the alarm proper functioning starts with a shop owner inside the shop monitored by the alarm. After an initial wait of 5 seconds, the owner sets up the alarm by turning the alarm switch to the ON position and immediately exits the room. The alarm takes 30 seconds to be fully activated and thus the storyboard is designed to consider enough time for the activation of the alarm to be complete. We decide on a 40 second wait, after which a thief opens the door, turning ON the door switch. It takes the thief 2 seconds to close the door, meaning the door switch is turned OFF. Assuming the alarm is functioning properly, it starts a 5 second countdown from the moment the thief opened the door, since it took him 2 seconds to close it that means there are only 3 seconds left for the alarm to go off. To conclude the story, the owner returns to the shop after 5 seconds *to catch the thief red handed* and turns OFF the alarm.

The extraction of the test sequence from the storyboard follows the general steps of finding the associated coverability tree and converting the sequence of transitions into a table U_t. Since we are creating a specific chain of events to drive the alarm in a close-to-real situation, we want to be able to customize the passage of time and delay between events. Our solution is to use timed transitions that also require an input change and then generate the corresponding table U_t entries by defining the input change to happen at the timer timeout. In this use case, transition `sb_tr_23` in the considered storyboard indicates that the input `SWITCH_DOOR` will be set to 1 (ON) at the timeout instant of the corresponding 40 seconds timer.

Simulating the Alarm IOPT (Figure 3.5) with the Storyboard IOPT (Figure 5.4), we obtain the results as presented in Figure 5.5. We see that the situation laid out by the storyboard occurs as proposed. It translates in a correct generation of input events at the desired time instants which given as input to the PLC program. The PLC program state evolution matches our expectations, showing a typical correct functioning of the alarm in Alarm mode.

In conclusion, the DES to PLC conversion toolchain functions as expected, properly deriving the event sequence from the storyboard and using it to guide a simulation of the alarm controller.

The second method allows a quicker change or fine-tuning to the sequence of events, but requires a higher knowledge of the modeled system IOPT, the Storyboard IOPT and the generation of a table U_t. Instead of the simpler representation of a story through an IOPT model, we can create a specific sequence of events directly through a custom made table U_t. This allows us to be more precise with the timings of the events, makes it easier to add or remove events and to create specific tests. Table U_t (5.3) tells the exact same story as the Storyboard from Figure 5.4, except now we can clearly see the exact timings of each event change and can quickly perform highly specific modifications.

$$U_t = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 5.0010 & 0 & 1 & 1 & 0 & 1 & 0 \\ 5.0030 & 0 & 1 & 1 & 0 & 1 & 0 \\ 45.003 & 0 & 1 & 1 & 0 & 0 & 1 \\ 45.005 & 0 & 1 & 1 & 0 & 0 & 1 \\ 47.005 & 0 & 1 & 1 & 0 & 1 & 0 \\ 47.007 & 0 & 1 & 1 & 0 & 1 & 0 \\ 50.007 & 0 & 1 & 1 & 0 & 1 & 0 \\ 50.009 & 0 & 1 & 1 & 0 & 1 & 0 \\ 55.009 & 1 & 0 & 1 & 0 & 1 & 0 \end{bmatrix} \tag{5.3}$$

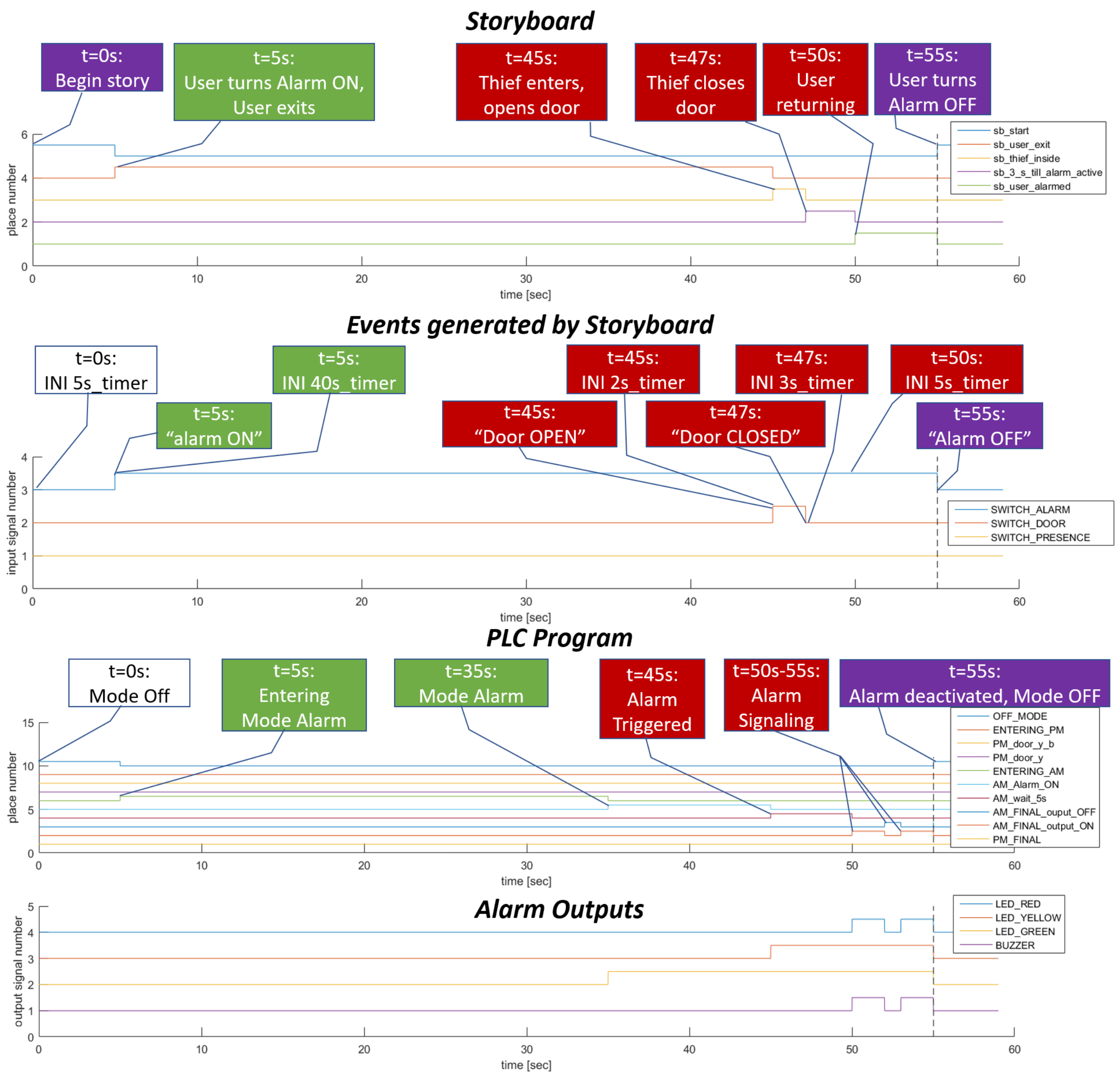


Figure 5.5: Alarm PLC program (Figure 3.5) driven by a storyboard (Figure 5.4) showing the typical correct functioning of the alarm in Alarm mode.

$$U_t = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 2 & 0 & 1 & 1 & 0 & 0 & 1 \\ 3 & 0 & 1 & 1 & 0 & 1 & 0 \\ 4 & 0 & 1 & 1 & 0 & 1 & 0 \\ 5 & 1 & 0 & 1 & 0 & 1 & 0 \end{bmatrix} \tag{5.4}$$

5.2.2 Validation using the PLC Development Tools

The `myterminal5` interface (Figure 5.6), made available for the Industrial Processes Automation course taught at Instituto Superior Técnico [11], recreates our use case alarm system and makes possible the connection to Unity Pro, a Schneider Electric software. This software allows both programming a real PLC and emulate a PLC, which enables the testing of an application without a physical connection to the PLC and other devices.

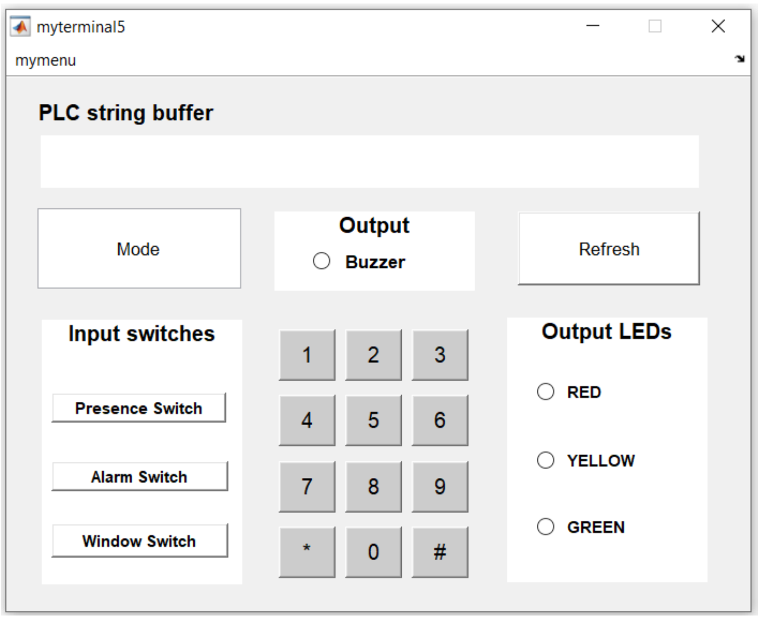


Figure 5.6: myterminal5 interface that emulates the alarm system.

The difference in relation to a real API is that there are no I/O and communications modules. However, Schneider Unity Pro can simulate the PLC with a MODBUS server and run the I/O digital interfaces, such as `myterminal5`, as a MODBUS client. In doing so we establish a communication system that allows sending information from the interface to the PLC and back. Clients can only write to Coils. Read operations can be performed on Coils or Inputs. In

Schneider Unity Pro, both Coils and Inputs are stored in internal Boolean memories ($\%M$) and share the same memory space. Also, in Schneider Unity pro, both Discrete Inputs/Outputs and Holding Registers are stored in internal Word memories ($\%MW$). In summary, we can simulate I/O modules to work with the PLC. This means we can implement the alarm supervisor IOPT Petri net without having to connect to a real PLC and have it communicate with the alarm terminal. Furthermore, this alarm terminal application can read a table of events and replay it, i.e. allows injection of inputs to the Unity Pro PLC simulator given a table U_t.

Given the enunciated conditions, we run the DES to PLC conversion toolchain ST compiler to convert the alarm supervisor IOPT Petri net to Structured Text code, which is placed in a Unity section. Then, we connect the PLC in simulation mode, transfer the project to PLC and run. Finally, we load table U_t (5.3) to `myterminal5` application and start the injection of inputs protocol to the Unity Pro PLC simulator.

To reduce the waiting times for the PLC simulation, two time-related changes were made, which do not change the general behaviour of the PLC-DES. The first change was done to the waiting time needed to fully activate the alarm: instead of 30 seconds we now wait 3 seconds. The second change was done to the storyboard itself: the thief now opens the door 3 seconds after the alarm is fully activated, instead of 10 seconds after.

The simulation plots the results in Figure 5.7. Firstly, we observe a correct injection of inputs and coil writing, following the desired storyboard. Secondly, we can observe a matching behaviour of the PLC-DES to the one observed in Figure 5.5. The output evolution corresponds to the Alarm mode state evolution. This means the PLC-DES simulated on a PLC shows a correct typical functioning of the alarm in Alarm mode, hence we validate the correct IOPT Petri net conversion to ST code.

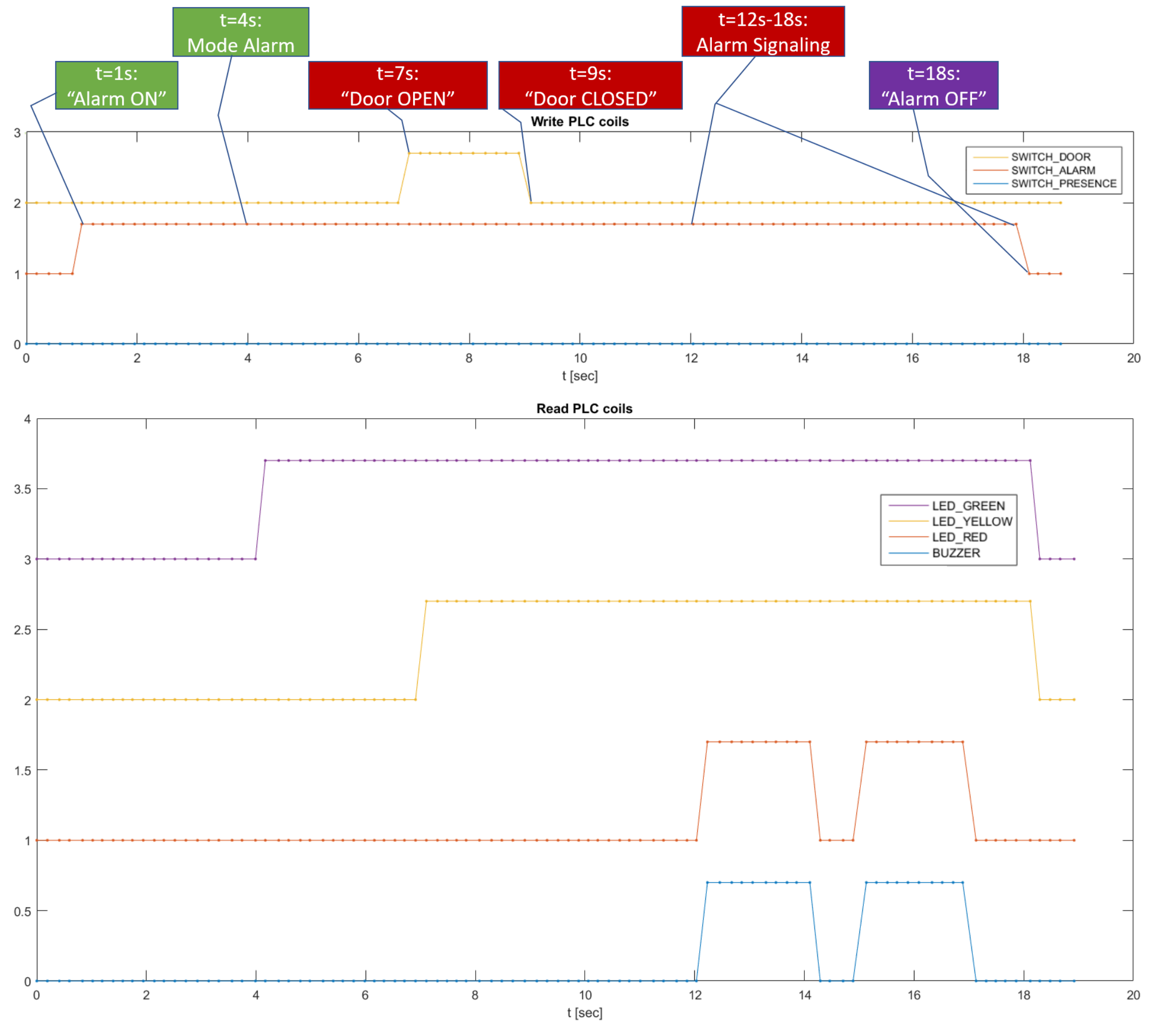


Figure 5.7: Alarm PLC-DES simulation on Unity Pro given a table U_t.

5.3 Use Case 3 - Effects of Hardware Constraints on Petri net Designs

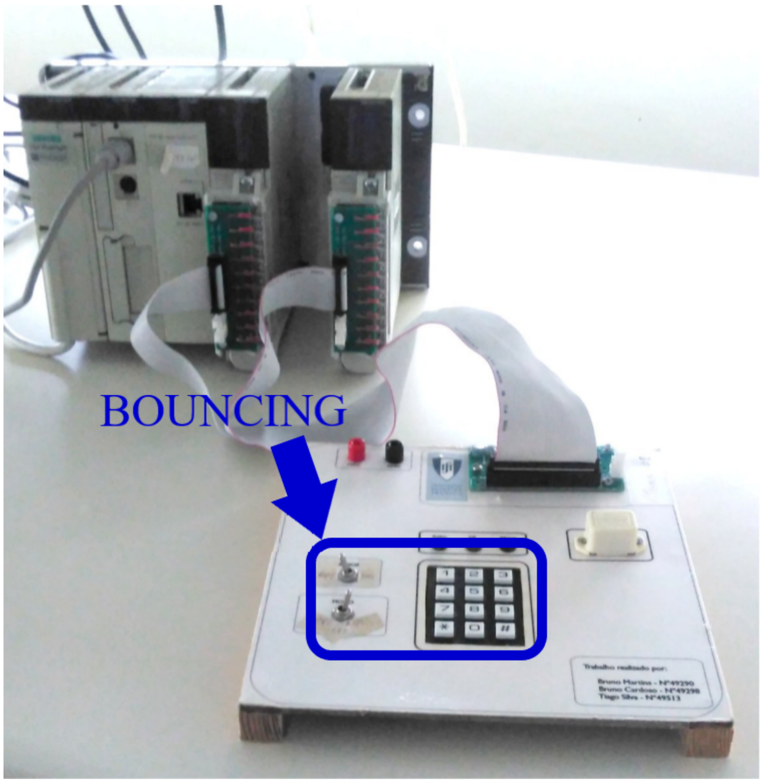


Figure 5.8: Alarm setup. Bouncing happens on the switches, but exists mostly in the push-buttons.

Hardware limitations arise when moving from simulation to hardware implementations. In many cases the software developer is not aware of hardware specific constraints and therefore does not take the limitations into account. Consequently, the implementations can fail validation tests.

A typical example in the PLC programs, which are in essence based on scan cycles, is the existence of inputs appearing as short pulses. Those fast inputs may not be recognized, depending on their duration and the state of the PLC scan cycle. On an opposite case, spikes associated to input bouncing, may be input by the PLCs if the spikes are long enough for being accepted. In another time scale, fast changes in the system states may lead to critical races. More common hardware problems are listed in Appendix C.

In this section we consider bouncing. The problem of bouncing may disrupt the correct functioning of the alarm supervisor implemented as the PLC-DES. The World-DES alarm interface

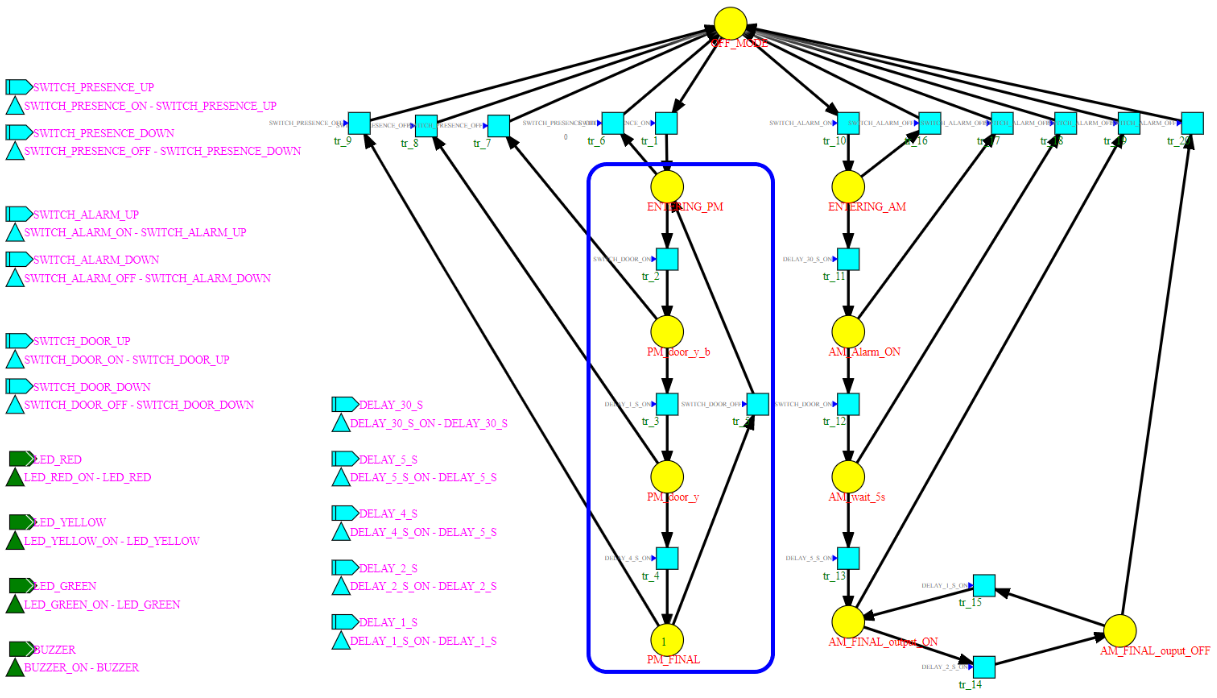


Figure 5.9: Area where bouncing may disrupt the expected functioning of the controller.

includes input switches and push-buttons, as seen in Chapter 3, therefore when modelling and simulating the alarm we should take into consideration the bouncing problem that may happen on the switches (Figure 5.8).

A simple method to look for cases where bouncing may result in unexpected behaviour is to search for a sequence of transitions containing the ON and OFF events of the same signal. For example, in Figure 5.9 and Figures 5.10 - 5.12, after the door was kept open for more than 5 seconds (the amount of time needed to finish detecting a presence), the door open switch turns OFF. The Petri net should transition from state `PM_FINAL` (Figure 5.10) to `ENTERING_PM` (Figure 5.11) by firing the enabled transition `tr_5`. However, this signal is controlled by a mechanical switch and bouncing may occur. The door open switch may be turned ON, enabling transition `tr_2`, and making the Petri net transition into state `PM_door_y_b` (Figure 5.12) right after. This is an undesirable behaviour, the user expects the alarm to be on stand-by for the door open switch to be activated, but the alarm is already working up to signal a detected presence. In other words, the alarm incorrectly detects a person entering the shop when the door is closed due to bouncing.

We propose table U_t (5.5) simulating the bouncing effect on the door switch signal. This

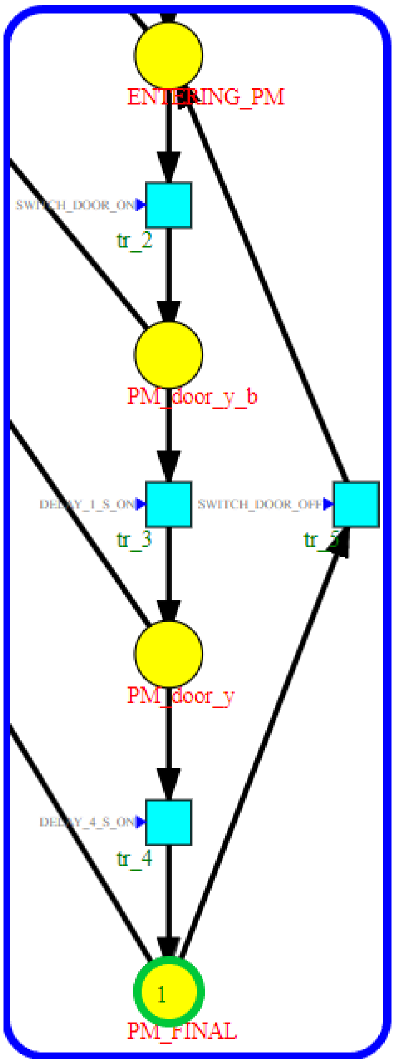


Figure 5.10: Petri net currently at state PM_FINAL.

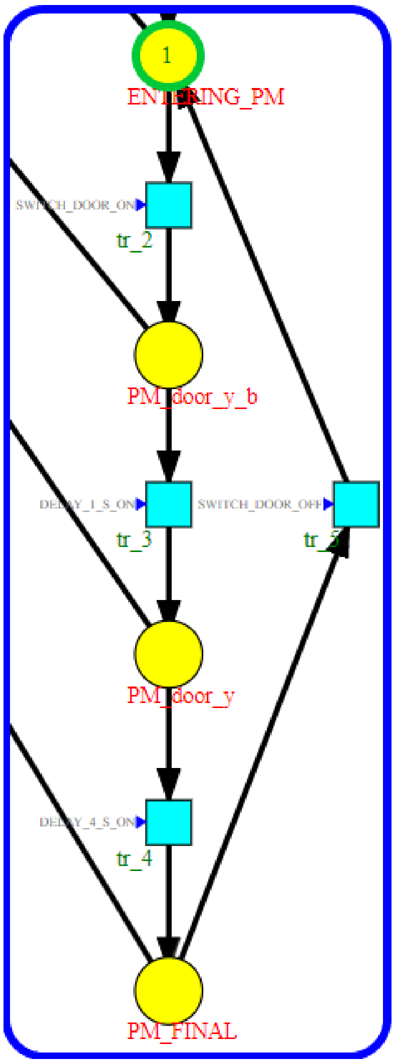


Figure 5.11: Evolution to state ENTERING_PM by turning OFF the door switch, enabling transition tr_5.

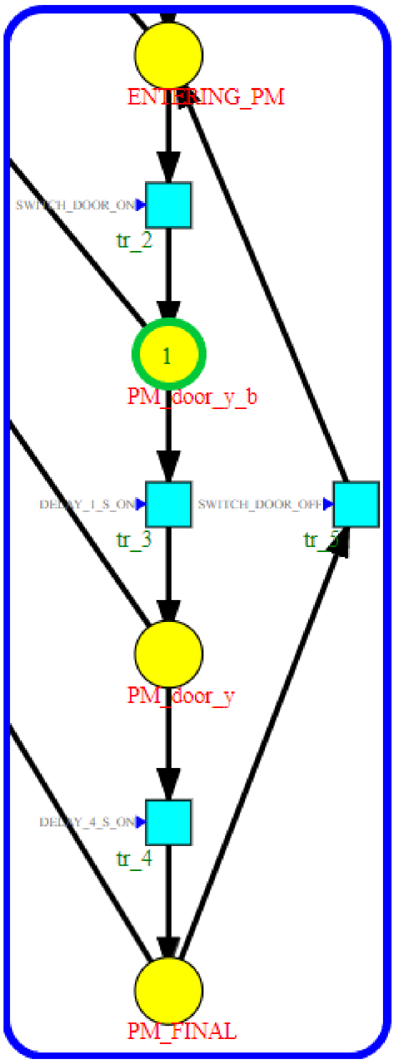


Figure 5.12: Evolution to state PM_door_y_b due to bouncing on the door switch, causing the signal to be ON and enabling transition tr_2.

signal OFF and ON part correspond to the last 2 columns, respectively. Note that the other input signals values should be set correctly matching the current state, so for example the presence switch should be at the ON position.

$$U_t = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0.1 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0.1001 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0.1002 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0.1004 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0.1005 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0.1006 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0.1007 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0.1008 & 1 & 0 & 0 & 1 & 1 & 0 \end{bmatrix} \quad (5.5)$$

The simulation results in Figure 5.13 confirm our suspicions. The first bounce of the signal is incorrectly accepted as an input, reflecting in an undesired change of state. Once again, it was meant for the Petri net to transition solely from `PM_FINAL` to `ENTERING_PM`, but it actually reaches the state `PM_door_y_b`, representing an undesirable functioning.

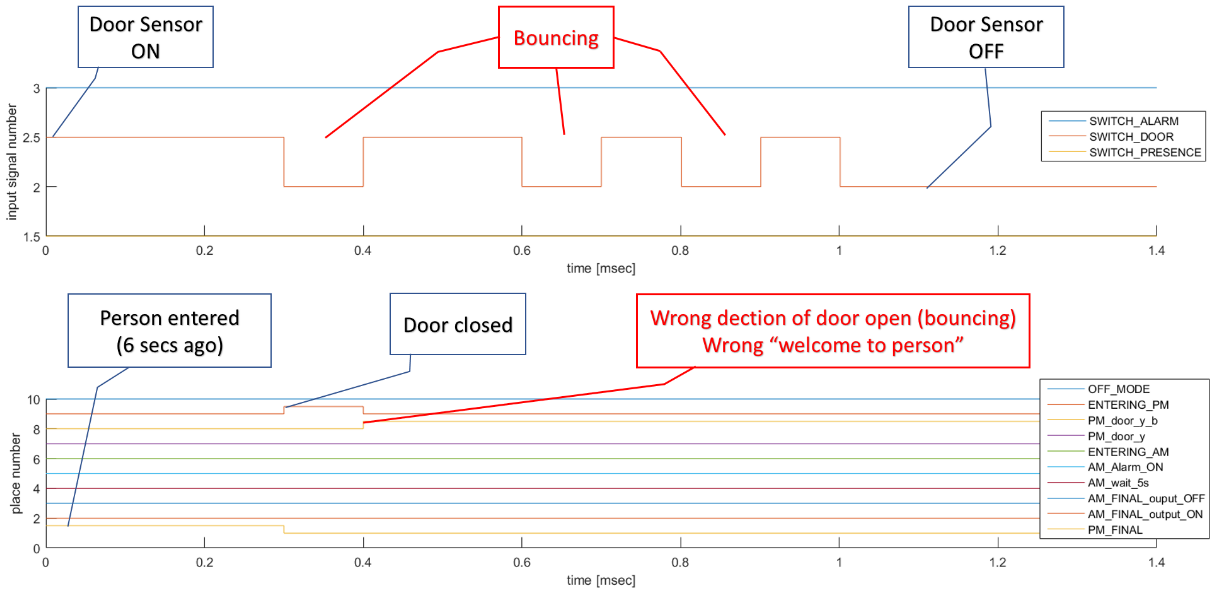


Figure 5.13: Behaviour of Petri net's region of Figure 5.9 given table U_t (5.5).

We propose a solution for debouncing directly in the IOPT Petri net. Debouncing can be done by connecting sequentially first place of the troublesome sequence to a transition `tr_dbnc_1`, connected to a place `debouncer`, connected to a transition `tr_dbnc_2`, connected back to the first place. The event associated with transition `tr_dbnc_1` must be the same as the entry transition to the troublesome place. The event associated with transition `tr_dbnc_2` must be the same as the exit transition to the troublesome place. Finally, a timer must be added to the exit transition of the first place that leads to the second place in the troublesome sequence. The associated time should be defined according to the type of input being dealt with. In our case 2ms are enough to filter the bouncing effect on the switch. Note that this solution intends for the mark to bounce around between this `debouncer` place and the troublesome one for the bouncing duration. This way it should reset the timer every time it exits and reenters the troublesome place, which is connected to the now timed transition.

In the case of our alarm, the troublesome situation happens when transitioning into state `ENTERING_PM` with bouncing on input signal of the door switch. We implement our solution as seen in Figure 5.14. This way, the Petri net only transitions to state `ENTERING_PM` when the door switch signal is stable at the OFF position. Note that with this solution, initially the mark is at place `Debounce_door` when door switch signal stabilizes at the ON position. In this case there are no outputs associated this state, but if there were a final touch to the solution is to associate to the `Debounce_door` place the same outputs as `PM_FINAL` so the output behaviour is not affected.

Using table U_t (5.5), we simulate the modified alarm. Observing Figure 5.15, we can clearly see the debouncing mechanism added to the Petri net model successfully handles the bouncing effect. Only after input signal given by the door switch stabilizes at the OFF position for 2ms does the Petri net evolve to state `ENTERING_PM` and stays there.

To be fair, debouncing should be applied to all inputs. Even if their bouncing may not affect the behaviour, bad contact may cause a quick signal change that may set off a transition. This quick signal change would generate a one-time bouncing, which would be contained by the debouncing mechanism.

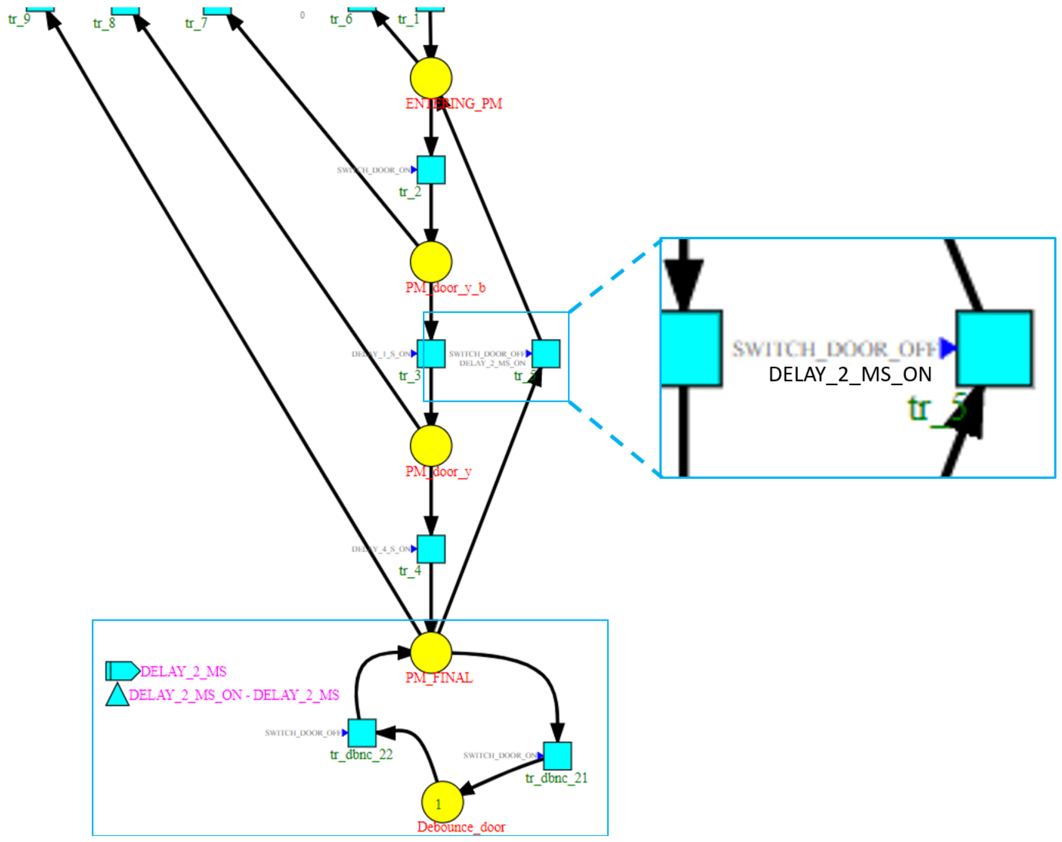


Figure 5.14: Petri net debouncing implementation for place PM_FINAL and input signal SWITCH_DOOR.

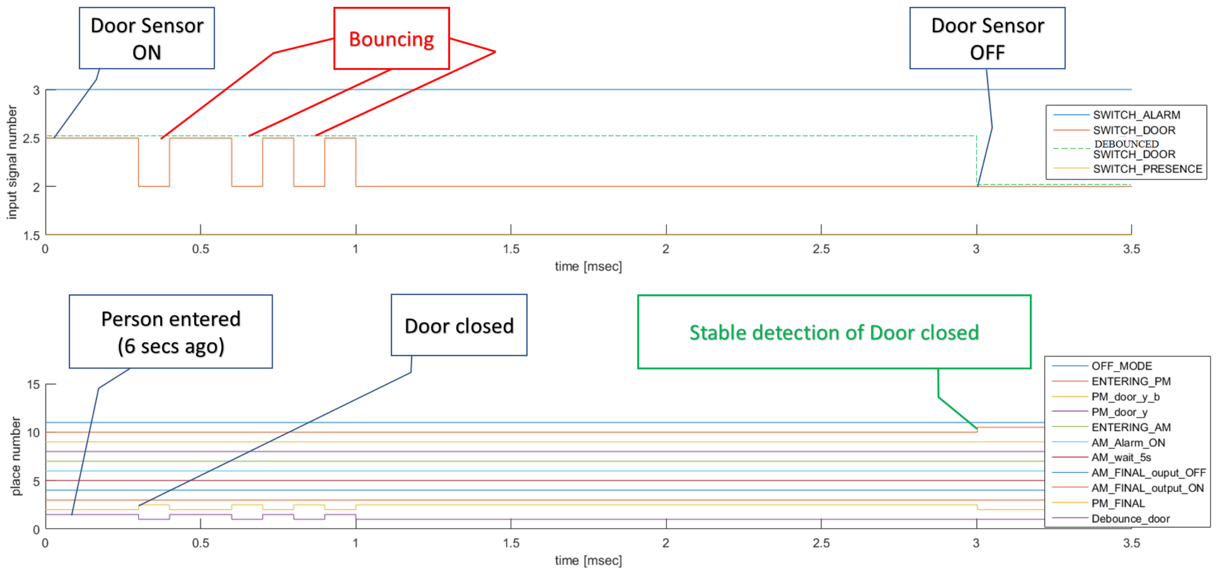


Figure 5.15: Behaviour of Petri net's region of Figure 5.14 given table U_t (5.5).

Chapter 6

Conclusion and Future Work

The work described in this thesis is focused on the production of industrial process supervisors implemented in PLCs. In particular approaches the aspect of assessing the produced PLC code. More in detail, a PLC code production toolchain is used and further developed, and are proposed tools that check whether or not IOPT Petri nets are correctly translated to PLC Structured Text programs.

One challenge of doing validation by reachability analysis is the so called state space explosion. Our verification and validation work starts by selecting bounded Petri nets based on the construction and analysis of the coverability tree. Given the objective of confirming if the Petri net generates the behaviour on the PLC as desired by the designer, an automatic generation of event sequences was proposed for driving the Petri net through all reachable states, using the toolchain simulation tool. In addition, the notion of time was introduced in the toolchain for making PLC/DES supervisor code. This allows creating specific tests covering expected PLC/DES uses.

The *IOPT tools* toolchain, created in Universidade Nova de Lisboa to develop embedded system controllers, was the basis used in this thesis to design Petri nets. The designed Petri nets were then used as the input of the PLC/DES controller maker toolchain, created in Instituto Superior Técnico for teaching automation, to obtain PLC code. Given the PLC code production toolchain, we applied tools based on the coverability tree to validate the PLC code, together with anticipating the detection of design issues and studying the effects of hardware constraints. Use cases were considered to test and demonstrate the use of the proposed tools.

An alarm system based on a PLC was used as a case study along the thesis. This alarm communicates with a PLC emulator tool, provided by the PLC manufacturer, which is used as the entry point for testing PLC implementations [1]. This framework allowed the validation testing both of the DES to PLC conversion toolchain simulation environment and the IOPT Petri net to PLC code conversion. Furthermore, it was addressed the problem of bouncing that arises from the connection of digital systems to noisy, transient-prone, *bouncing* inputs. A method to test the toolchain robustness to bouncing is proposed, as well as a debouncing solution to be implemented directly on the IOPT Petri net.

Future work, in the vein of producing DES to PLC programming tools, is the compilation of several components of the proposed toolchain into a *1-click application*. In other words, to assemble the toolchain into a single use program. A hardware approach for validation, to compare the validation testing on a simulated PLC with a validation testing on a physical PLC, will be considered in a *post-pandemic* setting with complete access to the hardware tools. Another aspect to be developed is a method to exploit a modelled DES to detect unconsidered state evolutions, i.e. finding unintended behaviours that are not contemplated on the initial specifications of the DES.

In a concepts point of view, formal verification tools and methodologies, used in programming and hardware development, may suggest complimentary or alternative directions to using Petri net formalism and tools. Studying state explosion by using symbolic model checking [10, 34], may open more ways for the verification and validation. More in detail, the *Cone of Influence* [5] may identify which part of a DES model is relevant for the evaluation of the given requirement and in that way augment to a larger class the Petri nets considered in this thesis that can be implemented in PLCs.

[1]The Covid19 pandemic prevented experiments with the real PLCs. Despite using the same development environment, provided by the PLC manufacturer, some more experiments may be required after the pandemic situation to assess whether the PLC emulation may have significant differences.

Appendix A

Implementation details of Use Case 1

In this appendix we provide a more detailed information on the implementation details on use case 1 presented in section 5.1.

The `mk_event_sequence.m` function converts sequence of operation cycles as extracted from the coverability tree to a table U_t, as seen for the alarm case taking the sequence (5.1) and converting to table U_t (5.2).

Adjustments were made to `PN_s2act.m` and `PN_t_fire.m` functions provided in [13] to work properly with the alarm and continuous work was done to generalize both functions to accept any proposed Petri net that complies with the restrictions and requirements.

At the end of this first stage, **`PN_s2act.m`** receives the current state `M` and the `pnml_struct` containing all information on the IOPT Petri net. One of the `pnml_struct` cells contains information on what places activate which outputs, which for the alarm example can be seen in (A.1).

$$\begin{bmatrix} [9] & '41' & 'o' \\ [3\ 4\ 7\ 8\ 9\ 10] & '42' & 'o' \\ [6\ 7\ 8\ 9] & '43' & 'o' \\ [4\ 9] & '44' & 'o' \end{bmatrix} \tag{A.1}$$

It check what outputs are to be set according to the marked places in `M`, storing each value on vector `act`, the output of the function. Consolidating with an example, consider the Petri net at state $[0; 0; 0; 0; 0; 0; 0; 1; 0; 0]$, meaning the 8th place is the only marked one. Checking against the information in (A.1), we can see the 2nd and 3rd rows contain that place, meaning the output events labeled '42' and '43' are to be set ON, while '41' and '44' are to be turned OFF. For this example we get `act`=$[0\ 1\ 1\ 0]$. To be noted that places may not be associated to any output, in such situation act is a zero vector.

Regarding **PN_t_fire.m**, it is given the current time instant t and the pnml_struct structure. It first checks the input value at time t according to table U_t. It proceeds to generate the possible firing vector qk, i.e., the function searches for the transitions whose inputs meet all required status, assigning 1 to those that do and 0 otherwise. Vector qk is the output of this function. As an example, Let the Petri net be at state $[0;1;0;0;0;0;0;0;0;0]$, meaning ENTERING_PM is the only marked place, and consider row entry [6 0 0 0 0 1 0 0 1 0 1] from table U_t as an example. This specific entry reads that at $t = 6s$ both Switch_Presence and SWITCH_DOOR are turned ON, whereas Switch_Alarm is turned OFF. One of the pnml_ struct cells contains information on transitions and the enabling inputs, which for the current example can be seen in (A.2)

According to the information, transitions 1 is possibly firable since it requires Switch_ Presence to be ON and transitions 2 and 12 are possibly firable on account of SWITCH_ DOOR being ON. For this example we set qk's 1st, 2nd and 12th entries as 1 and the others 0, qk=[1 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0].

The decision of which transitions are actually enabled is done by filter_possible_ firings.m, by checking if the possibly firable transitions meet their preconditions. For the input values presented on the second row of table (5.2) for the initial marking of the IOPT Petri net model in Figure 3.5 we can clearly state the only transition that can fire is transition 1. The Petri net mark then changes from OFF_MODE to ENTERING_PM. This first response can be observed in Figure 5.3.

$$\begin{bmatrix}
'100' & 'tr\_1' & 'SWITCH\_PRESENCE\_ON' & '1' & 1 & Inf \\
'103' & 'tr\_2' & 'SWITCH\_DOOR\_ON' & '5' & 2 & Inf \\
'107' & 'tr\_3' & 'DELAY\_1\_S\_ON' & '105' & 3 & Inf \\
'112' & 'tr\_4' & 'DELAY\_4\_S\_ON' & '102' & 4 & Inf \\
'115' & 'tr\_6' & 'SWITCH\_PRESENCE\_OFF' & '2' & 5 & Inf \\
'119' & 'tr\_10' & 'SWITCH\_ALARM\_ON' & '3' & 6 & Inf \\
'123' & 'tr\_11' & 'DELAY\_30\_S\_ON' & '101' & 7 & Inf \\
'127' & 'tr\_12' & 'SWITCH\_DOOR\_ON' & '5' & 8 & Inf \\
'133' & 'tr\_13' & 'DELAY\_5\_S\_ON' & '102' & 9 & Inf \\
'136' & 'tr\_8' & 'SWITCH\_PRESENCE\_OFF' & '2' & 10 & Inf \\
'140' & 'tr\_7' & 'SWITCH\_PRESENCE\_OFF' & '2' & 11 & Inf \\
'142' & 'tr\_16' & 'SWITCH\_ALARM\_OFF' & '4' & 12 & Inf \\
'143' & 'tr\_17' & 'SWITCH\_ALARM\_OFF' & '4' & 13 & Inf \\
'150' & 'tr\_15' & 'DELAY\_1\_S\_ON' & '105' & 14 & Inf \\
'151' & 'tr\_14' & 'DELAY\_2\_S\_ON' & '104' & 15 & Inf \\
'158' & 'tr\_18' & 'SWITCH\_ALARM\_OFF' & '4' & 16 & Inf \\
'159' & 'tr\_19' & 'SWITCH\_ALARM\_OFF' & '4' & 17 & Inf \\
'160' & 'tr\_20' & 'SWITCH\_ALARM\_OFF' & '4' & 18 & Inf \\
'170' & 'tr\_5' & 'SWITCH\_DOOR\_OFF' & '6' & 19 & Inf \\
'173' & 'tr\_9' & 'SWITCH\_PRESENCE\_OFF' & '2' & 20 & Inf
\end{bmatrix} \tag{A.2}$$

Appendix B

Implementation details of use case 2

In this appendix we provide a more detailed information on the implementation details on use case 2 presented in section 5.2.

B.1 Function Implementation

When it comes to the simulating environment, the main updates were done to `PN_s2act.m` and `PN_tfire.m`.

PN_s2act.m Besides setting the output for the current state, this function is now responsible for turning on and off the timers associated to each transition. It does so by verifying which transitions are enabled and which are not. Four situations may occur: (1) if the transition is enabled but its timer is stopped, then start the timer; (2) if the transition is enabled and the timer is started but not finished, do nothing; (3) if the transition is enabled and the timer is finished, then the transition's other inputs are checked; (4) if the transition is disabled, stop and set the timer to zero if started, else it prevails zeroed.

PN_tfire.m The updated version now takes into account the time factor when generating the vector of possibly firable transitions. In other words, it now also confirms if the associated timers are ready or not, if there is any.

set_Options.m New settings were added that the user must update according to the test one intends to perform. The user can now select between simulating a system to the full extent or testing a storyboard either given by an IOPT storyboard or a customized sequence of events given by a U_t table. The user may also decide if the simulation should take into consideration timed transitions and variable delays between events or if the events should happen sequentially with the same time interval between them, as seen in Use Case 1 (subsection 5.1).

B.2 Storyboard Implementation

This section contains only tables providing additional information on the IOPT Petri net components used to create the storyboard on Figure 5.4.

Name	**Mode**	**Description**
40 seconds Timer	Input	Used to delay the turning of SWITCH_DOOR on by 40 seconds, on transition sb_tr_23.
3 seconds Timer	Input	Used to decide the returning of the owner, on transition sb_tr_25.

Table B.1: New Inputs and Outputs necessary for the storytelling.

Condition Identifier	**Description**
sb_start	Start of the story.
sb_setup_alarm	User switches the alarm on.
sb_user_exit	The user exists the room. The room is now empty. A thief will open the door after 40 seconds.
sb_thief_inside	The thief enters, closing the door within 2 seconds.
sb_3_s_till_alarm_active	After closing the door, the thief stays inside the room for 3 seconds.
sb_user_returns	The user notices the alarm was triggered and immediately returns, catching the thief, and taking 5 seconds to switch off the alarm.

Table B.2: Identification of Storyboard Conditions (Places). The storyboard generates no outputs.

Event Identifier (sb_tr_)	**Description**
21	Fires after 5 seconds (timer controlled transition). Quiet time frame to set up the story.
22	Fires when the Alarm switch is on.
23	Fires after 40 seconds (timer controlled transition) if the door switch is on.
24	Fires after 2 seconds (timer controlled transition) if the door switch is off.
25	Fires after 3 seconds (timer controlled transition).
26	Fires after 5 seconds (timer controlled transition) if the Alarm switch is off.

Table B.3: Identification of Storyboard Events (Transitions).

Appendix C

Connecting Digital Systems to noisy Inputs

This appendix is complementary to use case 3 in section 5.3.

Manufacturers of DES have to deal with problems that arise from the connection of digital systems to noisy, transient-prone, "bouncing" inputs. They have to make systems more robust and reliable by simplifying design, reducing CPU time and overhead, and replacing multiple passive components. Some of the problems to take into consideration when using hardware are:

Race Condition - is an undesirable situation that occurs when a device or system attempts to perform two or more operations at the same time but, because of the nature of the device or system, the operations must be done in the proper sequence in order to be done correctly.

Transients and ESD - Voltage transient is a temporary unwanted voltage in an electrical circuit that range from a few volts to several thousand volts and last micro seconds up to a few milliseconds. Transient voltages are caused by the sudden release of stored energy due to incidents such as lightning strikes, unfiltered electrical equipment, contact bounce, arcing, capacitor bank or generators being switched ON and OFF. Electrostatic discharge (ESD) is a tiny version of lightning. As the current dissipates through an object, it's seeking a low impedance path to ground to equalize potentials. In most cases, ESD currents will travel to ground via the metal chassis frame of a device. However, it's well known that current will travel on every available path. In some cases, one path may be between the PN junctions on integrated circuits to reach ground. This current flow will burn holes visible to the naked eye in an integrated circuit, with evidence of heat damage to the surrounding area. One ESD event will not disrupt equipment operation. However, repeated events will degrade equipment's internal components over time.

Overvoltage - is caused by improper wiring, miscellaneous fault conditions, and power-supply sequencing (in which one box with power off is connected to another with power on, even temporarily). Electronic and electrical devices are designed to operate at a certain maximum supply voltage, and considerable damage can be caused by voltage that is higher than that for which the devices are rated. The amount of current in a circuit depends on the voltage supplied: if the voltage is too high, then the wire may melt and the light bulb would have "burned out real time". Similarly other electrical devices may stop working, or may even burst into flames if an overvoltage is delivered to the circuit.

Bouncing - When a contact is closed or opened, it will close and open many times before finally settling in a stable state.

Any dry contact/mechanical switch when they bang together, they rebound a bit before settling (bouncing). This erratic action can wreak havoc on data, because the exact number of counts does not necessarily repeat in the long term. Switch bounce is not consistent from unit to unit, lot to lot, or even over the life of an individual switch. Membrane switches and some other types don't appear to bounce when new, but all mechanical switches bounce sometimes. Nothing can ensure that another switch of the same type will act the same way, or that a particular switch will remain bounce-free as it ages. This is also true in small metal contact, when they open or close multiple electronic signals are generated. This behavior of a contact is interpreted as multiple false input signals and a digital circuit will respond to each of these on-off or off-on transitions. This problem has always been a very important problem when interfacing switches, relays, etc. to a digital control system.

Further explanation on the bouncing problem is given below, since it is the main focus in use case 3.

C.1 Handling input bouncing on PLCs

In the days of mechanical keyboard, when a key was pressed, the key would oscillate (bounce) against its contact several times before settling. And when it was released, it would oscillate again until it reverted back to its reset position. The hardware works faster than the bouncing, resulting in the hardware thinking you are pressing the key several times. Nowadays we have capacitive keyboards and we don't need the debouncing “delays”, we can type as fast as we like.

Likewise, in the world of process control, we do apply debouncing logic for the same reason. In addition to bounce, switches and digital systems have other annoying habits. Strange things happen, for example, when you run switch wiring in a noisy industrial environment. An open switch has high impedance by definition, so interfering signals have an easy load to work against. Any noise impulse that is capacitively or inductively coupled to the switch wiring can cause phantom switch closures. Debouncing logic will also reduce jittering of digital values as it comes close to its alarming state.

There is also the reality of having to deal with different varieties and alternatives of hardware, each one presenting their own personal limitations within the same problem. Having access to a simulation environment we can recreate the hardware problems and test how it affects the proposed system, as well as tweaking the simulation parameters to our desire when facing different setups.

Simple Counter to test for Debouncing

Note that all figure are presented side-by-side at the end of the appendix to facilitate comparison.

In use case 3, we focus on the bouncing problem and ways to identify it and to prevent it, both from the user's and the toolchain developer's point of view. Naturally, debouncing is the process of removing the bouncing effect.

In the experimental part we try for debouncing using the alarm PLC-DES to keep consistency. In this appendix we present a simpler test. We propose a simple counter IOPT Petri net as presented in Figure C.1 with the desire to test if the tool performs debouncing on input events. Its only function is to count how many times the button `B_1` is pressed, adding 1 mark to place `Counter` every time the button signal is set (firing of transition `tr_1`).

Typical PLC scan times can range from 1ms to 20ms. Considering that switch bouncing occurs several times within 1ms, we generate the bouncing effect through the sequence of inputs presented on table U_t (C.1). Analysing it, we see that the button signal starts in the OFF position and is changing every few microseconds, which is an abnormal pressing rate for any human,stabilizing at the ON position in the end.

$$U_t = \begin{bmatrix} 0 & 1 & 0 \\ 0.0001 & 0 & 1 \\ 0.0002 & 1 & 0 \\ 0.0003 & 0 & 1 \\ 0.0006 & 1 & 0 \\ 0.0009 & 0 & 1 \\ 0.0014 & 1 & 0 \\ 0.0015 & 0 & 1 \\ 0.0016 & 1 & 0 \\ 0.0017 & 0 & 1 \\ 0.0018 & 1 & 0 \\ 0.0019 & 0 & 1 \end{bmatrix} \tag{C.1}$$

For an input signal to be properly read, with no uncertainty, there is a rule of thumb that states the input should be ON for at least one scan cycle. With such in mind and knowing the input is changing at a higher rate than the lowest common PLC scan time, we direct our attention to the simulation results shown in Figure C.4. We can observe that every time the signal changes, no matter how long it stays set, the counter is incremented. We conclude that the DES to PLC conversion toolchain under evaluation does not apply debouncing.

In Figure C.3 we overlap a PLC scan cycle of approximately 1.3 ms, showing a possible situation where the input would be incorrectly read. The signal is changing from OFF to ON, so the intended value to be read should be ON. Instead, for that scan the input read values OFF.

Debouncing can be done by means of hardware or software. A common hardware debouncing mechanism used is a resistor and capacitor integrator, which gives the fastest response. In general, you need a pull-up resistor, a resistor and a capacitor in series, a resistor to the input of a Schmitt-trigger buffer, and (often) a diode to ensure that the capacitor charge doesn't force lots of current through the buffer's input-protection network during power-down. The resulting parts count can be unwieldy for multiple-input systems.

Debouncing by means of software is the primary method in use today. A good debouncing routine is actually real-time software that acts as a simple low-pass digital filter. Non-switch digital inputs are often routed through debounce filters as well. That technique can eliminate short transients at the input by ensuring a stable state before reporting the input open or closed. Every commercial PLC will have some level of filtering to cancel electrical noise. Some have an adjustable setting to set input filter times. Many other PLCs have one or two inputs that can be configured to take high frequency signals. They usually have adjustable filters to be used as normal inputs though.

Regarding software solutions, which is within the scope of this thesis, some approaches can be taken by the manufacturer of a toolchain and some by the user proposing a Petri net.

From the manufacturer's side, both within the simulation environment and within the ST compiler, one can add a subroutine that assigns an ON-DELAY timer (TON) to each input. This timer will monitor the input and will set a corresponding internal bit to HIGH once the input has been set for a defined time (e.g. 2ms). However, the longer the acceptance interval, the longer the delay in responding to a valid switch closure. Another software solution would be a routine that aggregates events across a limited period of time to produce one "confirmed event" once it adds over a certain threshold.

From the user's point of view, the solutions are limited to the Petri net modelling. The previously mentioned solution of associating a timer to the bouncing input can be translated to a Petri net version. The corresponding implementation is to replace the transition that may be affected incorrectly by bouncing with a timed transition. The associated time depends on which mechanical contact is used as input. For the example considered in this use case, the time should be 2ms. An immediate possible problem to this solution is that the timer may start counting as soon as the preconditions are met, while the bouncing is still happening, depending on how the toolchain starts and stops timers and if they are directly associated to the transition's other inputs or solely depending on if the entry place is marked.

To make the solution more robust, with the help of 2 additional transitions and 1 additional place, we can let the mark bounce around between this new place and the intended one for the bouncing duration. This way it should reset the timer every time it exits and reenters the place tied to the timed transition.

We implement this solution to the previous counter as shown in Figure C.2.

Using table U_t (C.1), we simulate the modified counter. Observing Figure C.5, we can clearly see the debouncing mechanism added to the Petri net model successfully contains the bouncing effect and the counter only recognizes the button press after the signal has been set for 2ms.

A valid question that may arise is what if the mark stays in the new place when the bouncing stops. This can never happen because in the end the mechanical contact will always fully transition to the final state.

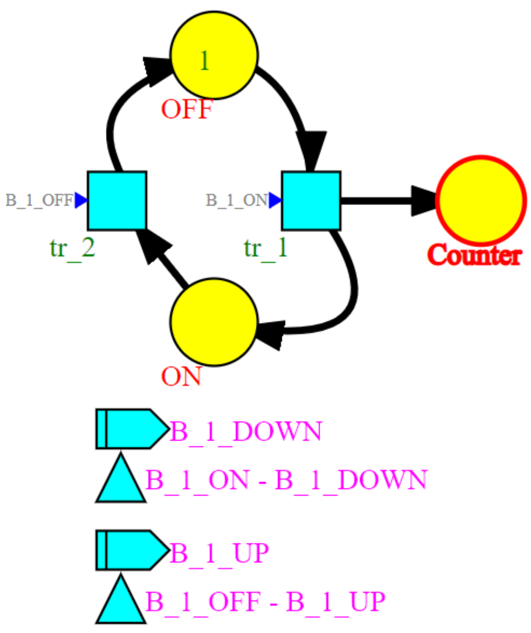


Figure C.1: A simple counter to detect bouncing.

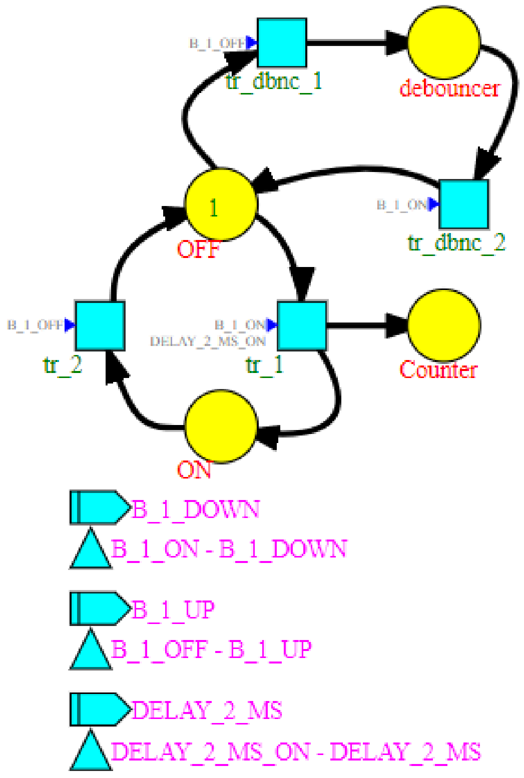


Figure C.2: A simple counter with debouncing mechanism.

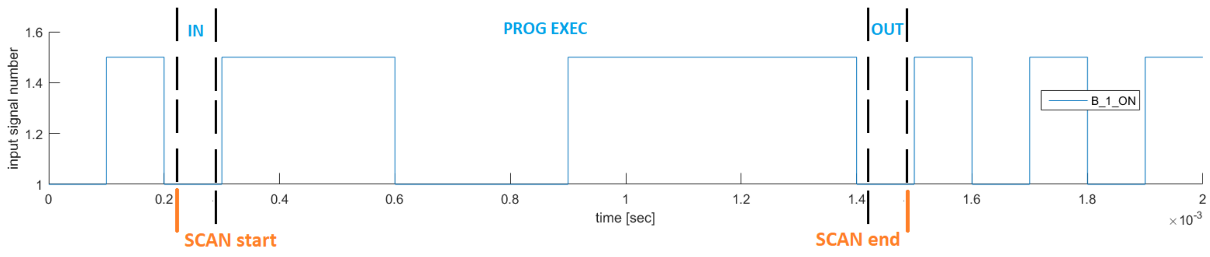


Figure C.3: Behaviour of IOPT Petri net in Figure C.1 given table U_t (C.1) overlapped with a possible scan cycle, showing an incorrect reading of the input signal due to bouncing.

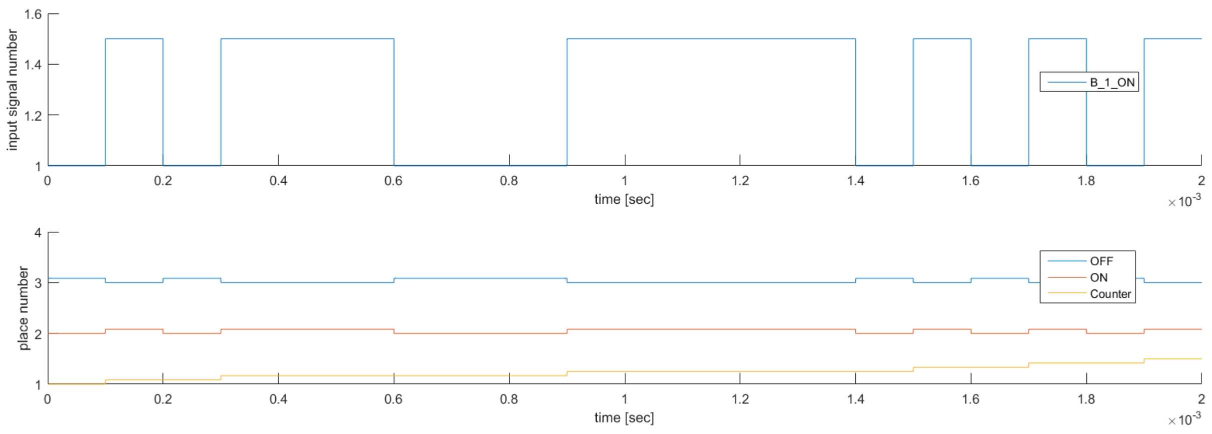


Figure C.4: Behaviour of Petri net in Figure C.1 given table U_t (C.1). The DES to PLC conversion toolchain does not apply debouncing.

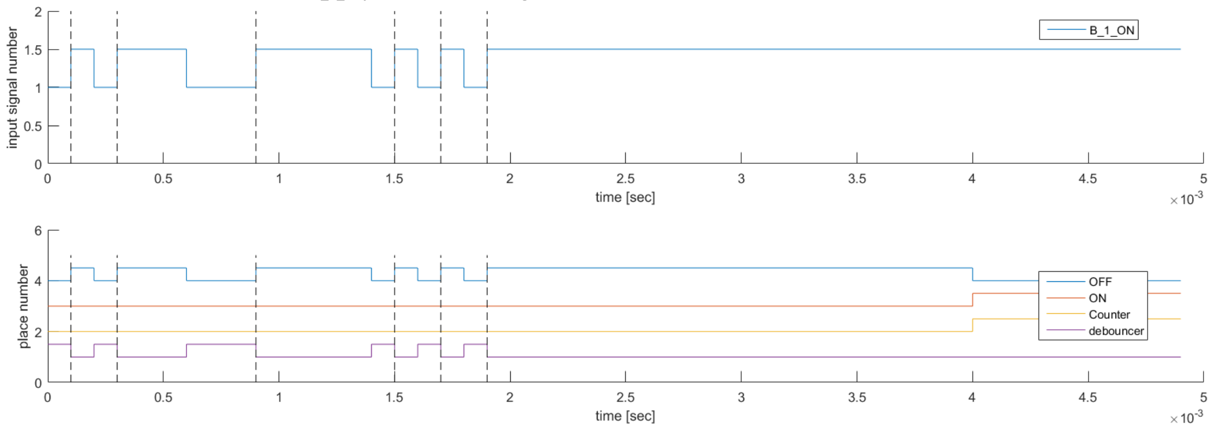


Figure C.5: Behaviour of Petri net in Figure C.2 given table U_t (C.1). Debouncing is now done on the IOPT Petri net.

Appendix D

Coverability Tree Construction Algorithm

The definition of the coverability tree is in section 4. Here are detailed the used function prototypes, starting from the more general function `Coverability_Tree.m`

Functions `transition_function.m` and `add_path.m` are used as subroutines.

Function	`[ RM, mat_path, track ] = Coverability_Tree(Pre, Post, M0)`
Input	**- Pre** : PxT : preconditions matrix **- Post** : PxT : postconditions matrix **- M0** : Px1 : intial marking vector
Output	**- RM** : PxM : Matrix of reachable markings (each column represents marking of one state) **- mat_path** : MxC : Cell matrix of paths to each reachable marking starting from the initial marking. Reachable markings with more than one path (which means rows with more than one non-empty column) means that marking has duplicate nodes in the tree) (**- track** : Mx4 : Cell matrix containing for each state (1st column) which is the following state (3rd column) depending on the transition fired (2nd column). Also contains the marking of the following state (4th column). This output is useful only for Graphviz)

Description	Generates the coverability tree, for the given Petri net and initial marking
Notation	**P**: #places , **T**: #transitions , **M**: #reachable markings , **C**: #different paths

Table D.1: `Coverability_Tree` function description.

Function	`[ next_RM ] = transition_function(RM, i, D, fx, k, mat_path)`
Input	**- RM** : PxM : Matrix of reachable markings (each column represents marking of one state) **- i** : Index of which RM is under evaluation **- D** : PxT : Incidence matrix **- fx** : 1xeT : Vector containing indexes of the enabled transitions **- k** : Index of which enabled transition is under evaluation **- mat_path** : MxC : Cell matrix of paths to each reachable marking starting from the initial marking. Reachable markings with more than one path (which means rows with more than one non-empty column) means that marking has duplicate nodes in the tree)
Output	**- next_RM** : Px1 : next marking reachable from RM(i) given fx(k)
Description	Generates the next reachable state according to the Petri net model of state transition, taking into account *node dominance* and the symbol ω

Notation	**P**: #places , **T**: #transitions , **eT**: #enabled transitions , **M**: #reachable markings , **C**: #different paths

Table D.2: `transition_function` function description.

Function	`[ mat_path ] = add_path(mat_path, i, j, fx, k, node_type)`
Input	**- mat_path** : MxC : Cell matrix of paths to each reachable marking starting from the initial marking. Reachable markings with more than one path (which means rows with more than one non-empty column) means that marking has duplicate nodes in the tree) **- i** : Index of which RM is under evaluation **- j** : Index of next_RM in mat_path **- fx** : 1xeT : Vector containing indexes of the enabled transitions **- k** : Index of which enabled transition is under evaluation **- node_type** : 1 if next_RM is duplicate node, 2 if next_RM is new node
Output	**- mat_path**
Description	**If Duplicate Node :** Add new path to existent duplicate node vector of paths in mat_path. Variable j is the index of the vector of paths on mat_path that lead to this existent duplicate node in mat_path **If New Node:** Expand mat_path by adding a new row storing the new path corresponding to the new node. This expansion is made by first copying the parent node's 1st path and then adding the transition fired from the parent node to reach the new node.

Notation	**eT**: #enabled transitions , **M**: #reachable markings , **C**: #different paths

Table D.3: `add_path` function description.

Bibliography

[1] Nanette Bauer, Ralf Huuck, Ben Lukoschus, and Sebastian Engell. A unifying semantics for sequential function charts. In *Integration of software specification techniques for applications in engineering*, pages 400–418. Springer, 2004.

[2] Christos G Cassandras and Stephane Lafortune. *Introduction to discrete event systems*. Springer Science & Business Media, 2008.

[3] C/S2ESC Software & Systems Engineering Standards Committee. 1012-2016 - ieee standard for system, software, and hardware verification and validation. `https://standards.ieee.org/standard/1012-2016.html`.

[4] Wojciech Czerwiński, Sławomir Lasota, Ranko Lazić, Jérôme Leroux, and Filip Mazowiecki. The reachability problem for petri nets is not elementary. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*, pages 24–33, 2019.

[5] Dániel Darvas, Borja Fernández Adiego, András Vörös, Tamás Bartha, Enrique Blanco Vinuela, and Víctor M González Suárez. Formal verification of complex properties on plc programs. In *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*, pages 284–299. Springer, 2014.

[6] Dániel Darvas, István Majzik, and Enrique Blanco Viñuela. Generic representation of plc programming languages for formal verification. In *Proc. of the 23rd PhD Mini-Symposium*, pages 6–9, 2016.

[7] Nachum Dershowitz and Zohar Manna. Proving termination with multiset orderings. *Communications of the ACM*, 22(8):465–476, 1979.

[8] John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Gordon Woodhull. Graphviz and dynagraph - static and dynamic graph drawing tools. In *GRAPH DRAWING SOFTWARE*, pages 127–148. Springer-Verlag, 2003.

[9] Javier Esparza and Mogens Nielsen. Decidability issues for petri nets. *Petri nets newsletter*, 94:5–23, 1994.

[10] Georg Frey and Florian Wagner. A toolbox for the development of logic controllers using petri nets. In *2006 8th International Workshop on Discrete Event Systems*, pages 473–474. IEEE, 2006.

[11] José Gaspar. Industrial processes automation (course ist/meec). `http://users.isr.ist.utl.pt/~jag/courses/api19b/api1920.html`. Accessed: 2020-01-30.

[12] José Gaspar. Petri net to plc converting. `http://users.isr.ist.utl.pt/~jag/course_utils/pn_to_plc/pn_to_plc.html`. Accessed: 2020-01-30.

[13] José Gaspar. Petri nets simulation. `http://users.isr.ist.utl.pt/~jag/course_utils/pn_sim/PN_sim.html`. Accessed: 2020-01-30.

[14] Luís Gomes, João Paulo Barros, Anikó Costa, and Ricardo Nunes. The input-output place-transition petri net class and associated tools. In *2007 5th IEEE International Conference on Industrial Informatics*, volume 1, pages 509–514. IEEE, 2007.

[15] Henrique Gonçalves and José Gaspar. Monitoring programmable logic controllers. Master's thesis, Departamento de Engenharia Eletrotécnica e de Computadores, Instituto Superior Técnico, 2015.

[16] GRES Research Group. Iopt tools. `http://gres.uninova.pt/IOPT-Tools/login.php`. Accessed: 2020-03-01.

[17] GRES Research Group. Iopt web tools - publications. `http://gres.uninova.pt/iopt_publications.html`. Accessed: 2020-08-31.

[18] Michel Henri Théodore Hack. Decision problems for petri nets and vector addition systems. MIT Project MAC, MAC-TM 59, Cambridge, MA, 1975.

[19] Michel Henri Théodore Hack. *Decidability questions for Petri Nets.* PhD thesis, Massachusetts Institute of Technology, 1976.

[20] Eric CR Hehner. A practical theory of programming. *Sci. Comput. Program.*, 14(2-3):133–158, 1990.

[21] C Michael Holloway. Why engineers should consider formal methods. In *16th DASC. AIAA/IEEE Digital Avionics Systems Conference. Reflections to the Future. Proceedings*, volume 1, pages 1–3. IEEE, 1997.

[22] International Standard 61131 IEC. Programmable logic controllers, part 3: Languages. 3rd Edition, Feb. 2013.

[23] Richard M Karp and Raymond E Miller. Parallel program schemata. *Journal of Computer and system Sciences*, 3(2):147–195, 1969.

[24] Robert M Keller. A fundamental theorem of asynchronous parallel computation. In *Sagamore Computer Conference*, pages 102–112. Springer, 1974.

[25] S. Rao Kosaraju. Decidability of reachability in vector addition systems (preliminary version). In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 267–281, 1982.

[26] Jean-Luc Lambert. A structure to decide reachability in petri nets. *Theoretical Computer Science*, 99(1):79–104, 1992.

[27] Jérôme Leroux. The general vector addition system reachability problem by presburger inductive invariants. *Logical Methods in Computer Science*, 6(3), 2010.

[28] Jérôme Leroux. Vector addition system reachability problem: A short self-contained proof. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 307–316, 2011.

[29] Jérôme Leroux. Vector addition systems reachability problem (a simpler solution). In *Turing-100 (EPiC Series in Computing)*, volume 10, pages 214–228. EasyChair, 2012.

[30] Jérôme Leroux and Sylvain Schmitz. Demystifying reachability in vector addition systems. In *2015 30th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 56–67. IEEE Computer Society, 2015.

[31] Jérôme Leroux and Sylvain Schmitz. Reachability in vector addition systems is primitive-recursive in fixed dimension. In *2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–13. IEEE, 2019.

[32] Richard Lipton. The reachability problem requires exponential space. *Department of Computer Science. Yale University*, 62, 1976.

[33] Ernst W Mayr. An algorithm for the general petri net reachability problem. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing*, pages 238–246. ACM, 1981.

[34] Mauro Mazzolini, Alessandro Brusaferri, and Emanuele Carpanzano. Model-checking based verification approach for advanced industrial automation solutions. In *2010 IEEE 15th Conference on Emerging Technologies & Factory Automation (ETFA 2010)*, pages 1–8. IEEE, 2010.

[35] José Meleiro and José Gaspar. Synthesis and identification of industrial processes. Master's thesis, Departamento de Engenharia Eletrotécnica e de Computadores, Instituto Superior Técnico, 2018.

[36] R. Pais, S. P. Barros, and L. Gomes. A tool for tailored code generation from petri net models. In *2005 IEEE Conference on Emerging Technologies and Factory Automation*, volume 1, pages 8 pp.–864, 2005.

[37] Fernando Pereira and Luís Gomes. Automatic synthesis of vhdl hardware components from iopt petri net models. In *IECON 2013-39th Annual Conference of the IEEE Industrial Electronics Society*, pages 2214–2219. IEEE, 2013.

[38] James Lyle Peterson. *Petri net theory and the modeling of systems*. Prentice Hall PTR, 1981.

[39] Carl Adam Petri. *Communication with automata*. PhD thesis, Darmstadt University of Technology, 1962.

[40] Rafael Rei and José Gaspar. Local and remote human machine interfaces for programmable logic controllers. Master's thesis, Departamento de Engenharia Eletrotécnica e de Computadores, Instituto Superior Técnico, 2019.

[41] Ricardo Reis and José Gaspar. Plc programming based on applied games. Master's thesis, Departamento de Engenharia Eletrotécnica e de Computadores, Instituto Superior Técnico, 2019.

[42] George S Sacerdote and Richard L Tenney. The decidability of the reachability problem for vector addition systems (preliminary version). In *Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 61–76, 1977.

[43] Manuel Silva. *Las Redes de Petri: en la Automática y la Informática*. Editorial Ac, 1985.

[44] Germany TGI group at the University of Hamburg. Petri nets tool database. `http://www.informatik.uni-hamburg.de/TGI/PetriNets/index.php`. Accessed: 2020-03-02.

[45] Jiacun Wang. *Timed Petri nets: Theory and application*, volume 9. Springer Science & Business Media, 2012.