Validation of Industrial Processes Implemented on PLCs based on Petri Nets

Hugo Conde Barroso, José Gaspar
Instituto Superior Técnico / UTL, Lisbon, Portugal
hugo.barroso@tecnico.ulisboa.pt, jag@isr.tecnico.ulisboa.pt

***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 conversion (code production) process. The Petri nets representation allows defining for each process the reachable set of states, which can have infinite size. We approach the problem of infinite size by building the coverability tree. We use the coverability tree to consider just the cases of finite reachable sets. In those cases we find operation cycles, avoiding therefore deadlock cases, and use the sequences of transitions to assess whether the process code effectively reaches all possible states.

Experiments have been carried with tools to create IOPT Petri nets, then converting to PLC code and assessing the implementations on a PLC based alarm system. Results indicate the tools allow testing reachability of all possible states, testing typical system usages and studying the effects of hardware constraints.

I. 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 (PLC) are the most common devices for integrating and controlling industrial processes as *Discrete Event System* (DES). Despite the widespread usage based on standard programming languages, it is still time consuming their direct programming. One convenient way to create PLC program is to use a higher level programming language, such as IOPT Petri nets to model the DES, followed by the translation to the PLC languages.

Features, properties and other information may be lost in translation, causing the PLC implementation to be incorrect. This can result in lost productivity and dangerous conditions. Testing the project with simulation tools 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 tested before the system is activated, increasing its quality. A validation process must be imposed to check that the software system meets the specifications and fulfills its intended purpose.

Simulation tools also need to be verified and validated to assure their correctness so it can best translate both the DES Petri net model and its relation to inputs and outputs. Focusing on the Input-Output Place-Transition Petri net (IOPT) subclass, we validate the simulation processes of a DES to PLC conversion toolchain . We propose an automatic generation of event sequences 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 toolchain behaves correctly.

II. Background

A. 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.

In software project management, engineering and testing, the terms Verification and Validation consist 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 [16].

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 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 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.

B. Petri Nets

A Petri net is weighted bipartite graph constituted by Places (circles), which are related to specific states of the system and are linked to Transitions (bars) which identify changes in the system. Arcs connect places to transitions (and vice-versa) which cannot directly be connected. Tokens inside places describe the Petri net Marking and specific conditions to the current active state.

From [2], a Marked Petri net, C is formally defined as a five-tuple

$$C = (P, T, A, w, \mu_0) \tag{1}$$

where P is the finite set of places defined as $P = (p_1, p_2, ..., p_n), n \in \mathbb{N}$, T is the finite set of transitions defined as $T = (t_1, t_2, ..., t_m), m \in \mathbb{N}$, (A, w) represent the arcs and 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 $\mu = [\mu(p_1), \mu(p_2), ..., \mu(p_n)] \in \mathbb{N}^n$. About the arcs, (A, w), A denote the finite set of arcs from places to transitions and from transitions to places in the graph defined as $A \subseteq (P\text{x}T) \cup (T\text{x}P)$, and w is the weight function on the arcs defined as $A \rightarrow 1, 2, 3, ...$; 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 $D^+,\ D^- \in \mathbb{R}^{n\text{x}m}$.

For simplicity, a marked Petri net shall henceforth be referred to as just a Petri net. The Petri net algebraic representation is done in matrix form by the *Incidence Matrix* $D \in \mathbb{R}^{n\text{x}m}$, which in turn is obtained from the weight function w as $D_{ij} = w(t_j, p_i) - w(p_i, t_j)$, where $i = 1, ..., n$ and $j = 1, ..., m$.

The dynamic of Petri nets is based on two concepts, *Enabled Transition* and *Firing Rule*.

a) 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., $\mu(p_i) \geq w(p_i, t_j), \forall (p_i, t_j) \in w$.

b) 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, also known as state transition mechanism, is based on the equation $\mu' = \mu + D \cdot q(j)$ where q is the firing vector and $q(j)$ a vector representing the firing of transition t_j.

C. IOPT Petri Nets

Petri nets capture the structural information of a system. Given we want to verify a tool that converts a DES to a PLC program, to be uploaded to a PLC allowing the DES to interact with real systems, we want to be able to test how the proposed DES responds to external inputs and acts on external outputs. We consider a class of Petri net that complements our common Petri net with input-output interactions.

IOPT Petri nets [26] 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. It allows the association of input events to transitions and output events to places, simulating 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. 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 Petri net dynamics. If the marked places are associated with an output event, these events will be triggered and the corresponding output signal value will change.

D. Reachability

The set of all states reachable by a Petri net from the initial state μ is called reachable set $\mathscr{R}$. The graphic representation of a Petri net reachable set is called a reachability tree.

Depending 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. When the reachable set is infinite the reachable tree becomes infinite. There exists a way of representing an infinite reachability tree in a finite form by introducing the infinity symbol ω, which is presented in section II-E1.

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*.

The *reachability problem* is a decision problem. For Petri nets it consists of deciding, given a Petri net C with initial marking μ_0 and a possible marking μ of C, if μ can be reached from μ_0. This problem was first proposed by Karp and Miller [17] within the scope of Vector Addiction Systems, but left unsolved.

Hack [15] and Keller [18] 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.

Esparza's research on decidability issues for Petri nets [7] states that, after an incomplete proof by Sacerdote and Tenney [30], decidability of the problem was established by Mayr in his seminal STOC 1981 work [23]. The algorithm uses a structure called *regular constraint graphs* and is based on conditions given by Presburger's Arithmetic.

The proof was then simplified by Kosaraju [19] 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 [20], where he completely suppressed the use of Presburger's Arithmetic.

1) Lambert's proof outline: Lambert 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

$$L(R, \mu_i, \mu_f) = L(\mathscr{U}_0, \varphi_0) \tag{2}$$

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

$$L(R, \mu_i, \mu_f) = \bigcup_{(\mathscr{U}, \varphi) \in \Gamma} L(\mathscr{U}, \varphi) \quad . \tag{3}$$

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 [6] 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.

2) Complexity: Regarding the complexity of the problem, Lipton showed in 1976 that the reachability problem lower bound is EXPSPACE-hard [22]. In 2019, Leroux and Schmitz [21] published a new upper bound to be non-primitive recursive Ackermannian. Later in the same year, Czerwinski [3] established a non-elementary lower bound, i.e. that the reachability problem needs a tower of exponentials of time and space.

E. Coverability Tree

In 1969, R. M. Karp and R. E. Miller [17] 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 [27] 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 [27] who based the proof on [14] and [17].

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.

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

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

- *Root node*. Corresponds to the initial state of the Petri net. Has no parent.
- *Terminal node*. Leaf node. Corresponds to a state with no enabled transitions (e.g., deadlock). Has no child nodes.
- *Duplicate node*. Corresponds to a state that is identical to a state already present in the tree. A node is considered duplicate when the already existing identical node is in the path from the root node to the node under

consideration. No successors of a duplicate node need to be considered.

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

$$\omega \pm a = \omega$$
$$a < \omega$$
$$\omega \leq \omega$$

In the created coverability tree, the symbol ω will appear as a marking of any unbounded place of a state that *dominates* another.

c) *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) $\exists i : \mu(p_i) > \mu'(p_i)$, 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.

2) Algorithm Steps Outline: The algorithm builds a coverability tree for a given Petri net preconditions matrix, post-conditions matrix and initial state. With both matrices one can obtain the incidence matrix needed for the state evolution.

The tree starts out with the initial state, which is the first "new state", and checks for all enabled transitions.

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

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

III. Implementation of Industrial Processes on PLCs using Petri Nets

A PLC program implemented by a Petri net with I/O interacts with a system, i.e. supervises a system. A Petri net typically has inputs at the transitions and outputs at the places. To run a Petri net it is required a simulator of the supervised system that can handle the inputs and the outputs that make the interaction possible.

A. DES To PLC Conversion Toolchain

We make usage of a DES to PLC conversion toolchain that provides a simulation environment. This technique was first presented and studied by H. Gonçalves [12]. It is based on the tools [10] and [11] presented in the Industrial Processes Automation course taught at Instituto Superior Técnico. In the years that followed, this technique was improved by J. Meleiro [25], R. Rei [28] and R. Reis [29]. This tool main objective is to convert a Discrete Event System modeled by a Petri net into a PLC Structured Text program, granting access to a controlled simulation environment.

We assume that the modeling of the DES is the user's responsibility and is done with IOPT Tools [13].

The general steps of the toolchain are: IOPT Petri net model parsing, coverability tree and event sequence computation, model simulation, ST compiler and test run on PLC.

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. 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 [5], assuming that the ambiguities of the standard are first resolved [1].

B. Case Study - Alarm System

In order to design a Petri net with I/O 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.

In addition to an OFF mode, the alarm system has two operating modes, (i) presence detection, where the alarm just informs a door has be open by a person entering the room, and (ii) intrusion detection, where whenever a door is open, or other supposed to be resting sensors become active, the alarm must be sound.

C. Proposed IOPT Petri net model

To model the alarm system, we created an IOPT Petri net, shown on Figure 1, based on the work of R. Rei [28].

The marking of each place is given by an integer inside each place, if there is no integer then the marking is 0. There are 10 places represented by yellow circles and 20 transitions by blue squares. 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

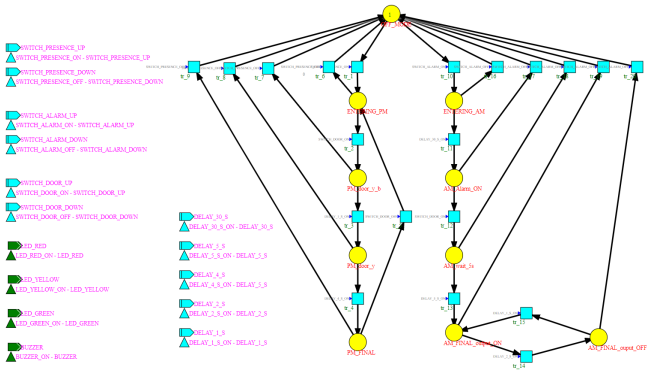

Figure 1. Petri net edited with IOPT Tools [13] to control the alarm.

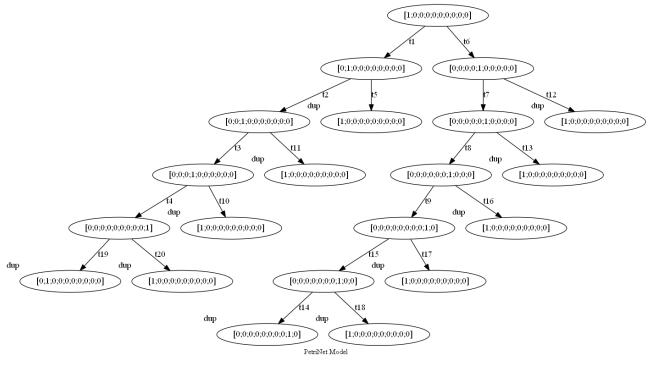

Figure 2. Coverability Tree for the proposed Alarm Petri net (Duplicate states are marked "dup" on the top left corner).

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 enable transitions by providing changes on the corresponding signal.

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.

D. Properties Study

The coverability tree in Figure 2 is obtained by the coverability tree algorithm previously proposed. 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. **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.

IV. Validation of Industrial Processes based on Petri Nets

A. Coverability Tree Adapted for Validation Testing

The tree data structure is a good choice to graphically represent the reachable states of a Petri net since one can easily see the path from the initial state to any other 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 I and implemented in MATLAB. The two modifications made to Cassandras and Lafortune's algorithm take place in **step 2.2** and its **substeps**.

B. Coverability Tree Construction

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, $\boldsymbol{M_P}$ is formed by two parts, $\boldsymbol{P}$ and $\boldsymbol{S}$.

$\boldsymbol{P}$ 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.

The implementation of the algorithm is summarized as pseudo-code in tables I and II.

C. Generation of Event Sequences

The **testing event sequence** is generated by extracting 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.

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. Each transition in the sequence

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 $\boldsymbol{MP}$.

Step 1 - Initialize $R = \{\mu_0\}$, $P = \emptyset$, $S = \emptyset$, $M = \emptyset$.

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

$x : (\mu_i \geq D_t^- \rightarrow 1) \wedge (\neg(\mu_i \geq D_t^-) \rightarrow 0)$.

Step 2.1 - If $x(t) = 0, \forall t$, then μ_i is terminal node.

Step 2.2 - Else for each $t : x(t) \neq 0$ create a new node μ' mapped by transition function f defined in Table II.

Let p_n be the set of sequences of transitions to μ_n and p_{n_j} be the sequence j. Let s_n be set of sequences of states indexes in $\boldsymbol{R}$ to μ_n and s_{n_j} be the sequence j. All sequences are initially empty.

Step 2.2.1 - If $\exists \mu_n \in R : \mu_n = \mu'$, then μ' is duplicate node. Do:

$p_n \leftarrow p_{i_1} \cup \{t\}$,

$s_n \leftarrow s_{i_1} \cup \{i\}$,

$M \leftarrow M \cup \{(\mu_i, t, \mu_n, "duplicate")\}$.

Step 2.2.2 - Else μ' 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 $S \leftarrow S \cup \{s_{N+1}\}$,

$M \leftarrow M \cup \{(\mu_i, t, \mu_{N+1}, , "new")\}$,

$R \leftarrow R \cup \{\mu_{N+1}\}$,

Step 3 - If all new nodes have been marked as either terminal or duplicate nodes, then stop.

Table I
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 $\boldsymbol{k}$.

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 II-E1 in section II-E (MATLAB solves the ω operations on its own, else an additional step would be needed to correct the propagation).

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 II-E1 in section II-E by doing:

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 $\mu'(a) = \omega$.

Step 3 Otherwise, μ' is as obtained in **Step 1**.

Table II
TRANSITION FUNCTION IMPLEMENTATION

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. Unless stated otherwise, 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.

D. Extension of Petri net model 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 [31], 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 to include a timer representation associated to each transition. It represents the timestamp at which the associated timer finishes. These parameters are updated within the simulation 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.

V. EXPERIMENTS

In this section are combined the tools for creating DES supervisors running on PLCs with the tools for 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. 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.

A. Use Case 1 - Reach All Possible States

The considered toolchain allows converting a high level DES design (PN) to an implementation running on a PLC or just as a simulation. In this use case we are assessing whether the conversion toolchain performed well, or not, the conversion.

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 in simulation 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. If anything is different than expected it is solely due to the toolchain. Otherwise, we can validate the success of the conversion.

Figure 3 illustrates the test we are performing, we feed the simulated inputs to the IOPT Petri net and evaluate

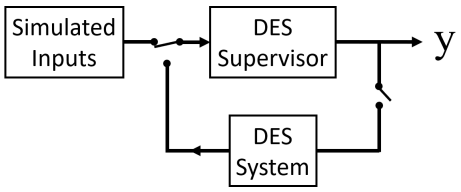

Figure 3. Illustration of use case 1 testing. The IOPT Petri net reacts only to the given input sequences.

the resulting state *y*. All interactions with the system to be controlled are ignored.

General considerations on the creation of input sequence:

- all timed transitions are converted to untimed transitions;
- inputs are reset to their starting values once the supervisor returns to its initial state, creating a soft reset.

The alarm supervisor IOPT Petri net produces the coverability tree shown in Figure 2. 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. We confirm that the IOPT Petri net evolution is correctly simulated and all reachable states are reached.

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.

B. Use Case 2 - PLC-DES interaction with World-DES

Before describing the experiment, we introduce two terms: PLC-DES and World-DES. The former refers to the DES implemented on the PLC, i.e. the controller/supervisor. The latter names the DES modelling a real system, i.e. the system to be controlled/supervised. Consider the alarm supervisor IOPT Petri net as the PLC-DES and the alarm system as the World-DES.

In the second experiment, the PLC-DES is run together with a World-DES. In particular are simulated timings of the real world. Are considered Petri nets extended with timed transitions, both for modelling the supervisors and the real world systems.

Validating the PLC-DES 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, representing the World-DES, to cover the expected usages.

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 sequence of events in the form a table U_t.

The first way shows the flow of events by following the laid out path of the storyboard. Consider the storyboard to be the typical correct functioning of the alarm in Alarm mode.

The story starts with a shop owner inside the shop monitored by the alarm. After an initial wait of 5 seconds, the owner turns the alarm switch to the ON position 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 (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. We 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 PLC-DES with the World-DES storyboard we obtain the results as presented in Figure 4. The toolchain correctly generates the input events at the desired time instants and 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 both DESs and the generation of a table U_t. We can create a specific sequence of events directly through a 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 (4) tells the exact same story as the storyboard, except now we can clearly see the exact timings of each event change.

$$U_t = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 5.0010 & 0 & 1 & 1 & 0 & 1 & 0 \\ 5.0030 & 0 & 1 & 1 & 0 & 1 & 0 \\ 45.003 & 0 & 1 & 1 & 0 & 0 & 1 \\ 45.005 & 0 & 1 & 1 & 0 & 0 & 1 \\ 47.005 & 0 & 1 & 1 & 0 & 1 & 0 \\ 47.007 & 0 & 1 & 1 & 0 & 1 & 0 \\ 50.007 & 0 & 1 & 1 & 0 & 1 & 0 \\ 50.009 & 0 & 1 & 1 & 0 & 1 & 0 \\ 55.009 & 1 & 0 & 1 & 0 & 1 & 0 \end{bmatrix} \quad (4)$$

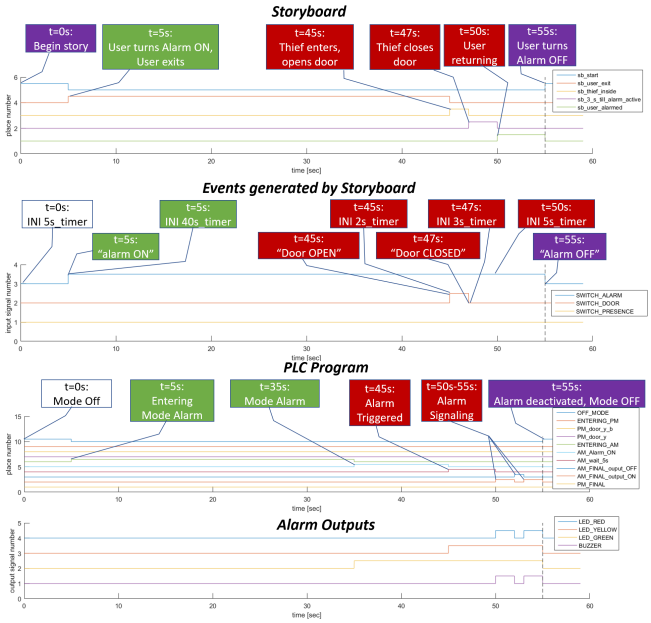

Figure 4. Alarm PLC program (Figure 1) driven by a storyboard showing the typical correct functioning of the alarm in Alarm mode.

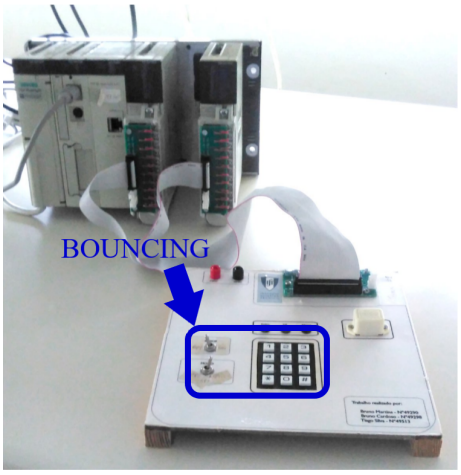

Figure 5. Alarm setup. Bouncing happens on the switches, but exists mostly in the push-buttons.

C. Validation using the PLC Development Tools

The `myterminal5` interface, made available for the Industrial Processes Automation course taught at Instituto Superior Técnico [9], 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.

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. This means we can implement the PLC-DES without having to connect to a real PLC and have it communicate with the World-DES. 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.

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 (4) to `myterminal5` application and start the injection of inputs protocol to the Unity Pro PLC simulator.

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 4. 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 to ST code.

D. Use Case 3 - Effects of Hardware Constraints on PN Designs

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. Other hardware problems such as transients and overvoltage may disrupt the DES behaviour.

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 includes input switches and push-buttons, therefore when modelling and simulating the alarm we should take into consideration the bouncing problem that may happen on said switches (Figure 5).

A 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, consider the Petri net in state `PM_FINAL`, after the door was kept open for more than 5 seconds (the amount of time needed to finish detecting a presence), and the door open switch turns OFF. The Petri net should transition from state `PM_FINAL` to `ENTERING_PM` 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

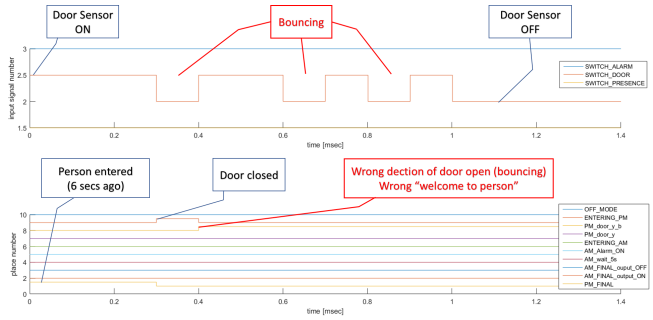

Figure 6. Behaviour of Petri net given table U_t (5) simulating bouncing for the example stated in this use case 2.

net transition into state `PM_door_y_b` 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) simulating the bouncing effect on the door switch signal. This signal OFF and ON part correspond to the last 2 columns, respectively.

$$U_t = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0.1 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0.1001 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0.1002 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0.1004 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0.1005 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0.1006 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0.1007 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0.1008 & 1 & 0 & 0 & 1 & 1 & 0 \end{bmatrix} \quad (5)$$

In Figure 6, the first bounce of the signal is incorrectly accepted as an input, reflecting in an undesired change of state. 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.

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. 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

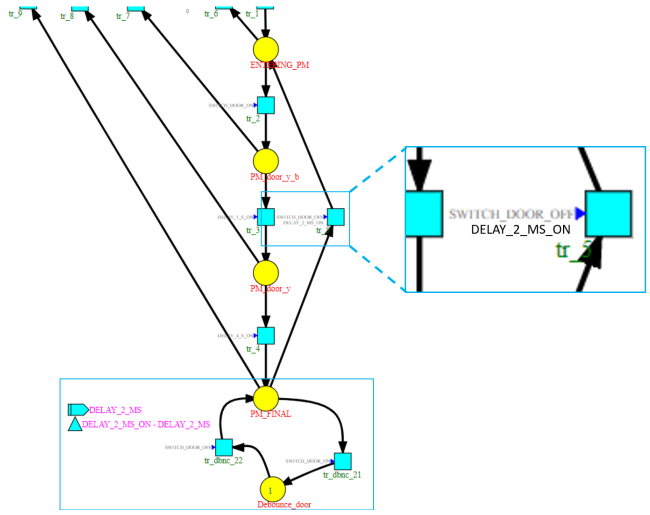

Figure 7. Petri net debouncing implementation for place PM_FINAL and input signal SWITCH_DOOR.

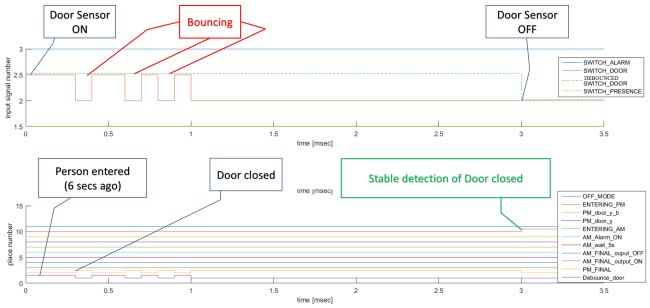

Figure 8. Behaviour of Petri net's region of Figure 7 given table U_t (5).

as seen in Figure 7. 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.

Observing Figure 8, we can 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.

VI. 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 verify and 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 may build on studying state explosion by using symbolic model checking [8], [24], opening the way for the verification and validation tools to accept also unbounded Petri nets. Using the *Cone of Influence* [4] to identify which part of the model is relevant for the evaluation of the given requirement, the unnecessary parts of the model can be removed without affecting the result, an unbounded Petri net may be reduced to a bounded one, thus easily implemented on a PLC.

[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.

REFERENCES

[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] 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.

[4] 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.

[5] 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.

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

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

[8] 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.

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

[10] 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.

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

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

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

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

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

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

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

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

[19] 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.

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

[21] 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.

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

[23] 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.

[24] 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.

[25] 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.

[26] 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.

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

[28] 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.

[29] 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.

[30] 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.

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