applied sciences

MDPI

Article

On the Design of Regulation Controllers for Automation Systems with *RCPetri*

Carlos Alberto Anguiano-Gijón, Anibal Cid-Gaona, José Manuel Chávez-Delgado and Carlos Renato Vázquez *

Escuela de Ingenieria y Ciencias, Tecnologico de Monterrey, Av. Ramón Corona 2514, Zapopan 45201, Mexico; a00834072@itesm.mx (C.A.A.-G.); a01066506@tec.mx (A.C.-G.); a01375065@tec.mx (J.M.C.-D.)
* Correspondence: cr.vazquez@tec.mx

Abstract: Regulation control for Petri nets is a control framework that allows the design of sequence controllers for automation systems in a systematic and efficient way. In order to implement this control framework, the MATLAB® app *RCPetri* has been developed. In this work, the *RCPetri* tool functionalities are described, including model generation, automatic specification generation, automatic control design, model and control simulation, automatic translation to PLC code, and communication by Modbus TCP/IP and OPC UA. Furthermore, three examples are presented to illustrate the application of the tool and the regulation framework: an electro-pneumatic device, a process control system, and a robotic manufacturing cell under a decentralized control scheme.

Keywords: automation; industrial informatics; discrete event systems; Petri nets; regulation control

Citation: Anguiano-Gijón, C.A.; Cid-Gaona, A.; Chávez-Delgado, J.M.; Vázquez, C.R. On the Design of Regulation Controllers for Automation Systems with *RCPetri*. *Appl. Sci.* **2022**, *12*, 3294. https://doi.org/10.3390/app12073294

Academic Editor: Dimitris Mourtzis

Received: 2 February 2022
Accepted: 15 March 2022
Published: 24 March 2022

1. Introduction

Petri nets (*PNs*) is a family of mathematical models that have been extensively used for representing and analyzing systems that evolve through the occurrence of events named discrete event systems [1–3]. Applications include the analysis of traffic [4], risk assessment [5], biological systems [6], logistics [7,8], and manufacturing systems [9], among others.

One application of *PNs* that has received a lot of attention in the literature is the design of control algorithms for manufacturing systems. The most common approach consists of the design of places (monitors) that control the firing of transitions, in order to impose a requirement described as a linear inequality, named generalized mutual exclusion constraint (GMEC) [10–13]. These constraints are useful to enforce safety specifications, avoiding states when two or more processes use the same resource. This control technique has also been extensively investigated to avoid blocking states in particular classes of *PNs* (see, for instance, [14–17]). A recent survey on the control of *PNs* can be found in [18].

Regulation control [19–21] is a different control paradigm, the objective of which is to design controllers that impose sequences of sensors and actuators activation. In detail, the system to be controlled, named Plant, is modeled as an Interpreted Petri net (*IPN*), which is an extension of *PNs* with labels associated with places (representing sensors) and transitions (representing actuators). The required behavior is described as an *IPN* as well, named Specification, representing required sequences of sensors and actuators activation. Then, a controller is designed as an *IPN* that enforces the Plant to evolve according to the given Specification. Common problems in automation systems, where the engineer requires to program a routine so the Plant describes a required operations sequence, can be addressed by this approach. The advantage of using this scheme is that the engineer only specifies the required tasks in a high-level description; next, the controller can be automatically generated by using the algorithms that have been developed for this purpose [19–21]. Furthermore, once the *IPN* controller is designed, it can be automatically translated into programmable logic controller (PLC) code for implementation.

In order to perform a practical implementation of the regulation control approach, we have developed a MATLAB® app named *RCPetri* [22], which is described in this work.

The app was designed to help the user to design and implement regulation controllers by following a control design workflow. In a first step, the app allows us to define an *IPN* that represents the Plant in a graphical way. Next, the user can define the required behavior in high-level fashion by using standard Excel tables, which are automatically translated by the app to an *IPN* Specification. Later, the controller can be automatically generated. Furthermore, the closed-loop system can be simulated to verify the behavior of the Plant under control. Moreover, the *IPN* controller can be automatically translated into PLC code, for its final implementation. Furthermore, a communication module has been implemented, which allows synchronization with other software or hardware through protocols Modbus Internet Protocol Suite (Modbus TCP/IP) and object linking and embedding for process-control-unified architecture (OPC UA). In order to illustrate the use of the app for the synthesis of controllers and their application, three examples are described, as follows:

1. An electro-pneumatic arm that can perform four different tasks, whose controller is implemented in a PLC;
2. A process control system simulated in a supervisory control and data acquisition (SCADA) software synchronized to the app via Modbus TCP/IP;
3. A manufacturing system with two robotic cells, simulated in a dedicated software and synchronized to the app via an OPC UA server, under a decentralized control scheme.

Different research groups have developed software tools for modeling and analyzing different *PN* types. For instance, GreatSPN [23] and PIPE2 [24] are well-known tools for performance analysis in generalized stochastic *PNs*. TimeNET [25] allows the analysis of non-Markovian stochastic *PNs*. Similarly, GPenSIM [26] can be used for performance evaluation in event graphs. CPN Tools [27] and Access/CPN [28] were proposed for the modeling, simulation, and analysis of colored *PNs*. Continuous and hybrid *PNs* can be simulated in HYPENS [29] and SimHPN [30]. PNetLab [31] was developed to model and simulate timed/untimed *PNs* and colored *PNs*, allowing the possibility to simulate *PNs* under control, where the controller is a subroutine that provides a conflict resolution policy. In [32], a combination of the graphical editor Petri Net Editor and the toolbox Petri Net Engine is used to implement a *PN* conflict resolution policy into a microcontroller. All of these tools allow us to graphically define a *PN* system, simulate the system, and perform particular types of analysis (mainly performance analysis and structural analysis). Nevertheless, despite the large amount of *PN* software, none of the existing tools allow us to define *IPN* with place and transition labels, and to synthesize regulation controllers. Moreover, to the best of our knowledge, none of the tools allow us to synchronize *PN* simulations with external software/hardware via standard protocols, and to automatically generate PLC code. Furthermore, the implementation of *RCPetri* on MATLAB® will allow other researchers to use it for the analysis and application of other algorithms, such as GMEC control, *PN* observers, and *PN* diagnosers, among others.

This paper is organized as follows: Section 2 presents basic definitions on *IPN*. Section 3 explains the regulation control framework. Section 4 introduces the *RCPetri* app with its main features, and the algorithms implemented for the main functionalities. Section 5 presents the application of *RCPetri* in the three examples previously mentioned. Section 6 summarizes the conclusions.

2. Basic Concepts

Basic concepts and definitions concerning *PNs* and *IPN* are recalled in this section (for more details see, for instance [3,33]).

Definition 1. *A Petri net (PN) structure, $\mathcal{N}$, is a bipartite digraph represented by the 4-tuple $\mathcal{N} = \langle P, T, I, O \rangle$, where $P = \{p_1, p_2, ..., p_n\}$ is the finite set of places, $T = \{t_1, t_2, ..., t_m\}$ is the finite set of transitions, $I: P \times T \to \mathbb{N}_0$ is a function representing weighted arcs connecting places to transitions, and $O: P \times T \to \mathbb{N}_0$ is a function representing weighted arcs connecting transitions to places.*

The *incidence matrix* of $\mathcal{N}$ is defined as a $|P| \times |T|$ matrix $\mathbf{C}$, such that $\mathbf{C}[i,j] = O(p_i,t_j) - I(p_i,t_j)$, $\forall p_i \in P$, $\forall t_j \in T$. Graphically, places are represented by circles, transitions by rectangles, and arcs by arrows. Given a node $v \in P \cup T$, ${}^\bullet v$ (resp. $v^\bullet$) denotes the set of nodes directly connected to v (resp. nodes directly connected from v). A PN is said to be a *state machine* if each transition has only one input and one output place (i.e., $\forall t \in T$ $|{}^\bullet t| = |t^\bullet| = 1$) and the Petri net graph is strongly connected.

Definition 2. *An Interpreted Petri net (IPN) system is a 8-tuple* $Q = \langle \mathcal{N}, M_0, \Sigma_I, \Sigma_{Id}, \Sigma_O, \lambda_I, \lambda_{Id}, \varphi \rangle$*, where:*

- $\mathcal{N}$ *is a PN;*
- M_0 *is the initial marking, defined as a function,* $M_0 \colon P \to \mathbb{N}_0$*, describing the number of tokens (dots) initially residing inside each place;*
- Σ_I *is the input alphabet—each element of* Σ_I *is an input symbol;*
- Σ_{Id} *is the identity alphabet—each element of* Σ_{Id} *is an identity symbol;*
- Σ_O *is the output alphabet—each element of* Σ_O *is an output symbol;*
- $\lambda_I \colon T \to 2^{\Sigma_I}$ *is the input labeling function that associates transitions to input symbols. A transition can be associated with several input symbols. If* $\lambda_I(t_j) \neq \emptyset$*, then the transition* t_j *is said to be controllable, otherwise it is uncontrollable;*
- $\lambda_{Id} \colon T \to \Sigma_{Id} \cup \{\epsilon\}$ *is the identity labeling function, where two transitions cannot have the same identity label and each transition can have one identity label at most;* ϵ *represents that the transition has not identity label;*
- $\varphi : P \to 2^{\Sigma_O}$ *is the output labeling function that associates places to output symbols. A place can be associated with several output symbols.*

The marking distribution of an IPN describes the state of the represented dynamical system; the marking changes according to the following rules:

- Given a current marking distribution, $M_k \colon P \to \mathbb{N}_0$, an external observer reads the symbols associated with the marked places, whose set is denoted as $\varphi(M_k) = \{o \in \Sigma_O | o \in \varphi(p),\ M_k(p) > 0\}$. $\varphi(M_k)$ is said to be the current *output* of the IPN and the symbols of $\varphi(M_k)$ are said to be *indicated* at M_k.
- A transition t_j is said to be *marking-enabled* at marking M_k if $\forall p_i \in {}^\bullet t_j$, $M_k(p_i) \geq I(p_i,t_j)$.
- A transition t_j is said to be *label-enabled* at marking M_k if every symbol $a \in \lambda_I(t_j)$ is indicated by either the current marking M_k or an external agent. If $\lambda(t_j) = \emptyset$, then t_j is always label-enabled.
- If t_j is marking-enabled and label-enabled at M_k, then it is said to be *enabled*, and it can be fired (denoted as $M_k[t_j >$). Otherwise, it is said to be *disabled*.
- The firing of a transition t_j, enabled at M_k, leads to a new marking M_{k+1} (denoted as $M_k[t_j > M_{k+1}$), caused by removing $I(p_i,t_j)$ tokens to each place $p_i \in {}^\bullet t_j$ and adding $O(p_i,t_j)$ tokens to each place $p_i \in t_j{}^\bullet$.

The marking function M can be expressed as a column vector $\mathbf{M} \in \mathbb{N}_0^{|P|}$, such that $\mathbf{M}[i] = M(p_i)$, $\forall p_i \in P$. In this work, both M and $\mathbf{M}$ will be called marking, since both represent the same information. Thus, a marking M_{k+1}, reached from M_k after firing a transition t_j, can be computed as $\mathbf{M}_{k+1} = \mathbf{M}_k + \mathbf{C}\vec{t_j}$, where $\vec{t_j}$ is the j-th column of the identity matrix of dimension $|T|$. The function φ can be represented by a $|\Sigma_O| \times |P|$ matrix $\boldsymbol{\varphi}$, in which $\boldsymbol{\varphi}[i,j] = 1$, if p_j is labeled with the i-th output symbol and $\boldsymbol{\varphi}[i,j] = 0$ otherwise. The output vector of the IPN at M_k is defined as $\mathbf{y}_k = \boldsymbol{\varphi}\mathbf{M}_k$.

A *firing sequence* is a sequence of transitions $\sigma = t_i t_j \dots t_k$, such that $M_0[t_i > M_1[t_j > \cdots M_m[t_k > M_{m+1}$. The marking $\mathbf{M}'$ reached after the firing of σ from a marking $\mathbf{M}$ can be computed as $\mathbf{M}' = \mathbf{M} + \mathbf{C}\vec{\sigma}$, where $\vec{\sigma} \in \mathbb{N}_0^{|T|}$ is a column vector, named the *Parikh vector*, defined such that $\vec{\sigma}[j] = k$, if t_j is fired k times in σ. This is denoted as $M[\sigma > M'$, and M' is said to be *reachable* from M.

A *PN* system is said to be *bounded* if $\exists k \in \mathbb{N}$, such that $\forall p \in P$ and for any M reachable from M_0, it holds that $M(p) \leq k$. A *PN* system is said to be *safe* if it is bounded with $k = 1$. A *PN* system is said to be *live* if $\forall t_j \in T$ and for any M reachable from M_0 there exists a fireable sequence σ, such that $M[\sigma > M'$ and t_j is (marking) enabled at M'.

IPN models for complex systems can be built from *IPN* component models, by using the synchronous composition. Given two safe *IPN* Q^1 and Q^2, the synchronous composition results in another safe *IPN* $Q^3 = Q^1 \| Q^2$, defined as the union of Q^1 and Q^2 in which the transitions sharing the same identity label are merged, keeping all the transition and place labels. The synchronous composition is commutative and associative.

Example 1. *Consider the double-acting electro-pneumatic actuator driven by a 5/2 valve and its IPN model shown in Figure 1. The incidence matrix* $\boldsymbol{C}$ *and the output labeling matrix* $\boldsymbol{\varphi}$ *is given by the following:*

$$\boldsymbol{C} = \begin{bmatrix} -1 & 1 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & -1 & 1 & 1 & -1 \\ 0 & 0 & 0 & -1 & 0 & 1 \end{bmatrix}, \quad \boldsymbol{\varphi} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

The input alphabet is defined as $\Sigma_I = \{a, b\}$*, where each symbol represents a command signal that allows the change of the valve position (i.e., energizing a coil), which leads to an actuator movement. The output alphabet is defined as* $\Sigma_O = \{A, B\}$*, where each symbol represents the activation of a sensor witnessing the actuator position. Then, transitions* $\{t_1, t_2\}$ *are controllable, while transitions* $\{t_3, t_4, t_5, t_6\}$ *are uncontrollable. At the initial state (Figure 1a), the actuator is retracted, the initial marking is given by* $\mathbf{M}_0 = [0\,1\,1\,0\,0]^T$ *and the output vector is* $\mathbf{y}_0 = [1\,0]^T$*, meaning that sensor A is active. By indicating the symbol b, transition* t_2 *fires (i.e., coil b is energized), and eventually the sequence* $\sigma = t_2 t_5 t_6$ *occurs, leading to the marking* $\mathbf{M}_3 = [1\,0\,0\,0\,1]^T$*, where the actuator is extended (Figure 1b), resulting in the output vector* $\mathbf{y}_3 = [0\,1]^T$*, meaning that sensor B is active.*

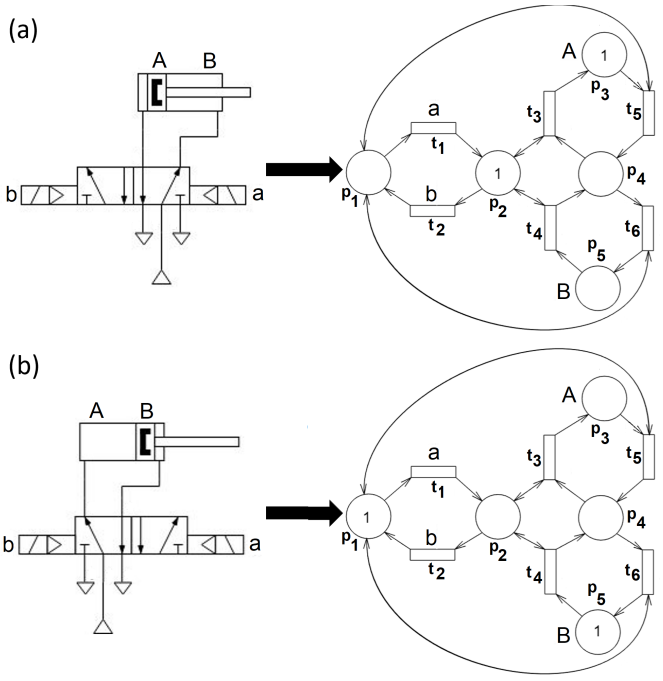

Figure 1. *IPN* model for an electro-pneumatic assembly: (**a**) the pneumatic actuator is retracted; (**b**) the actuator is extended.

3. Regulation Control for *IPN*

Regulation control is a paradigm, proposed for systems modeled by IPN, developed to induce sequences of output symbols on the system to be controlled, called the *Plant*. These sequences are specified in a high-level fashion by an IPN, called *Specification* (see, for instance [19–21]). Then, an IPN, named *Controller*, is designed for this purpose. Figure 2 shows a scheme describing the closed-loop system. In this, the controller indicates input symbols to the Plant transitions in order to make a selection of the Plant output ($\mathbf{\Omega} \cdot \mathbf{y}$) to be equal to the Specification output ($\check{\mathbf{y}}$), otherwise stated, to cause the output error $\mathbf{e} = \check{\mathbf{y}} - \mathbf{\Omega} \cdot \mathbf{y}$ to be null. For this, the controller has access to the complete marking of the Specification ($\check{\mathbf{M}}$), since the controller and the Specification are implemented in the same hardware, but it only has access to output signals from the Plant ($\mathbf{y}$), since only this information is available through the sensors connected to the controller hardware. In addition, the controller stops the Specification when the output error is not null. The Plant and Specification are formally defined as follows.

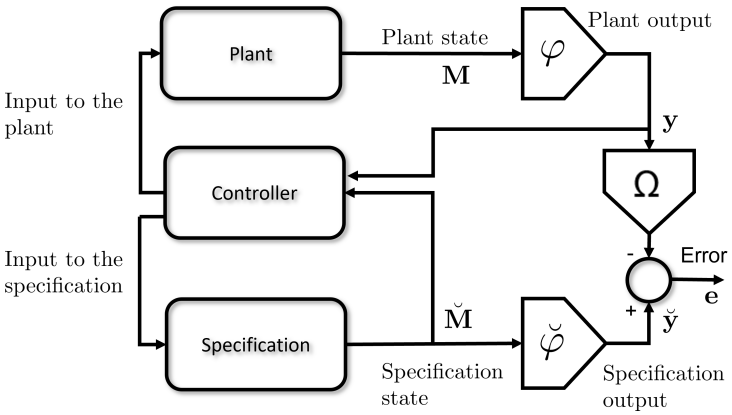

Figure 2. Regulation control scheme for IPN.

Definition 3. *The* Plant *is a safe live IPN,* $Q = \langle \mathcal{N}, M_0, \Sigma_I, \Sigma_{Id}, \Sigma_O, \lambda_I, \lambda_{Id}, \varphi \rangle$*, that models the discrete event system to be controlled. Two places cannot generate the same output symbol; moreover, input and output alphabets are disjointed.*

Definition 4. *A* Specification *is a safe, live-state machine IPN.* $\check{Q} = \langle \check{\mathcal{N}}, \check{M}_0, \check{\Sigma}_I, \check{\Sigma}_{Id}, \check{\Sigma}_O, \check{\lambda}_I, \check{\lambda}_{Id}, \check{\varphi} \rangle$*, together with a function* $\Omega \colon \Sigma_O \to \check{\Sigma}_O \cup \{\epsilon\}$ *that relates output symbols from the Plant to Specification output symbols. Each transition of the Specification has a unique identity symbol that is different from those of the Plant.*

Methodologies for modeling automated manufacturing systems as IPN have been proposed in [19,21]. Moreover, a methodology for designing IPN specifications has been presented in [34], in which the designer only provides a high-level description of what he wants to observe from the Plant.

Definition 5. *A* regulation controller *is a safe IPN,* $\hat{Q} = \langle \hat{\mathcal{N}}, \hat{M}_0, \hat{\Sigma}_I, \hat{\Sigma}_{Id}, \hat{\Sigma}_O, \hat{\lambda}_I, \hat{\lambda}_{Id}, \hat{\varphi} \rangle$*, fulfilling* $\hat{\Sigma}_{Id} \cap \Sigma_{Id} = \emptyset$ *(i.e., plant transitions do not share their identity symbols with controller transitions). The* closed-loop system *is defined as the IPN* $Q^{cl} = Q \| \check{Q} \| \hat{Q}$.

The regulation control problem consists of the synthesis of a controller that enforces the closed-loop system to exhibit the following behavior:

- if the output error is not null, then all the Specification transitions are disabled, and any fireable sequence drives the Plant to a marking where the output error is null;
- if the output error is null, then all the Plant transitions are disabled, but none of the Specification transition is disabled by the controller.

A synthesis method for computing such controller was presented in [20]. This method consists of the following three steps:

1. A marking mapping $\mathbf{\Pi}$ is computed, which relates each specification marking $\check{\mathbf{M}}_k$ to a Plant marking $\mathbf{M}_k$, such that the output error is null, i.e., $\check{\boldsymbol{\varphi}}\check{\mathbf{M}}_k - \boldsymbol{\Omega}\boldsymbol{\varphi}\mathbf{M}_k = \mathbf{0}$. Thus, if the Specification is at $\check{\mathbf{M}}_k$, the Controller should drive the Plant to $\mathbf{M}_k = \mathbf{\Pi}\check{\mathbf{M}}_k$. The mapping $\mathbf{\Pi}$ is computed by a linear integer programming problem (LIPP) that minimizes the number of firings between the plant markings (described by Parikh vectors $\vec{\sigma}_k$), subject to $\check{\boldsymbol{\varphi}} = \boldsymbol{\Omega}\boldsymbol{\varphi}\mathbf{\Pi}$, $\mathbf{\Pi}\check{\mathbf{M}}_0 = \mathbf{M}_0$, $\mathbf{\Pi}\check{\mathbf{C}}\vec{\check{t}}_k = \mathbf{C}\vec{\sigma}_k$ for all $\check{t}_k \in \check{T}$, and the entries of $\mathbf{\Pi}$ and vectors $\vec{\sigma}_k$ are non-negative integers.
2. For each specification transition $\check{t}_k$, such that $\check{\mathbf{M}}_k[\check{t}_k > \check{\mathbf{M}}_{k+1}$, for some reachable $\check{\mathbf{M}}_k$, a fireable controllable Plant sequence σ_k is computed, such that $\mathbf{\Pi}\check{\mathbf{M}}_k[\sigma_k > \mathbf{\Pi}\check{\mathbf{M}}_{k+1}$. Thus, if $\check{t}_k$ is fired in the Specification, reaching $\check{\mathbf{M}}_{k+1}$, then the Controller must induce the firing of σ_k to reach $\mathbf{\Pi}\check{\mathbf{M}}_{k+1}$, so the output error becomes null. This is performed by the A^* Algorithm 1, which computes a sequence with the minimum length that drives the Plant from $\mathbf{\Pi}\check{\mathbf{M}}_k$ to $\mathbf{\Pi}\check{\mathbf{M}}_{k+1}$. In the algorithm, the starting node is the empty sequence σ_ϵ; next, for each new node σ, the cost function is defined as $f(\sigma) = |\sigma| + ||\vec{\sigma} - \vec{\sigma}_k||_1$ (where $||\bullet||_1$ denotes the 1-norm), and the reached marking $\mathbf{M}_{reached}(\sigma)$ is computed (i.e., $\mathbf{\Pi}\check{\mathbf{M}}_k[\sigma > \mathbf{M}_{reached}(\sigma)$). Some conditions are included to ensure controllability (lines 7–10 and 12).
3. Finally, an IPN controller $\hat{Q}$ is built, which enforces the previously computed Plant sequences σ_k, through the indication of Plant input symbols, when the corresponding specification transitions $\check{t}_k$ have been fired.

Algorithm 1: Computation of a controllable sequence σ_k.

1 **Input:** IPN Plant structure Q, starting marking $\mathbf{\Pi}\check{\mathbf{M}}_k$, ending marking $\mathbf{\Pi}\check{\mathbf{M}}_{k+1}$, Parikh vector $\vec{\sigma}_k$.
2 **Output:** Controllable sequence σ_k.
3 Initialize $Open = \{\sigma_\epsilon\}$ and $Closed = \emptyset$. Set $\mathbf{M}_{reached}(\sigma_\epsilon) = \mathbf{\Pi}\check{\mathbf{M}}_k$. Set $f(\sigma_\epsilon) = ||\vec{\sigma}_k||_1$.
4 **while** $Open \neq \emptyset$ **do**
5 Find the sequence $\sigma \in Open$ with the lowest cost $f(\sigma)$
6 Compute the set of transitions that are marking-enabled at $\mathbf{M}_{reached}(\sigma)$ as $T_\mathbf{M} = \{t_j \in T | \forall p_i \in {}^\bullet t_j, \mathbf{M}_{reached}(\sigma)[i] \geq I(p_i, t_j)\}$
7 **if** $\exists t_i,\ t_j \in T_\mathbf{M}$, *such that* $\lambda_I(t_i) = \lambda_I(t_j) = \emptyset$ **then**
8 Update $T_\mathbf{M} = \emptyset$
9 **if** $\exists t_i \in T_\mathbf{M}$, *such that* $\lambda_I(t_i) = \emptyset$ **then**
10 Update $T_\mathbf{M} = \{t_i\}$
11 **for** *each* $t \in T_M$ **do**
12 **if** $\nexists t' \in T_M$, *such that* $\lambda_I(t') \subseteq \lambda_I(t)$ **then**
13 Define $\sigma' = \sigma t$
14 Define $\mathbf{M}_{reached}(\sigma') = \mathbf{M}_{reached}(\sigma) + \mathbf{C}\vec{t}$
15 Define $f(\sigma') = |\sigma'| + ||\vec{\sigma}' - \vec{\sigma}_k||_1$
16 **if** $\mathbf{M}_{reached}(\sigma') = \mathbf{\Pi}\check{\mathbf{M}}_{k+1}$ **then**
17 Define $\sigma_k = \sigma'$
18 **Stop** the algorithm.
19 **if** $\sigma' \notin Closed \cup Open$ **then**
20 Set $Open = Open \cup \{\sigma'\}$
21 Update $Closed = Closed \cup \{\sigma\}$, $Open = Open \setminus \{\sigma\}$

The closed-loop system under the synthesized regulation controller is safe (i.e., bounded), since the Controller only constraints the behavior of the Plant, which is already safe. Moreover, the closed-loop system is live, since the Controller induces the occurrence of each computed sequence, which is fireable and controllable by construction, when the corresponding specification transition fires, and the Specification is live by definition.

4. *RCPetri* MATLAB® App

Figure 3 shows the steps that a practitioner should follow to implement the regulation control framework. The *RCPetri* MATLAB® app was designed to aid engineers to perform each of these steps. The app's interface (shown in Figure 4) includes a main "Menu area", a "Visualization panel", a "Drawing area", and a "Tab panel". The app functionality is organized in Tabs, which can be accessed from the "Tab panel" and the "Tabs menu".

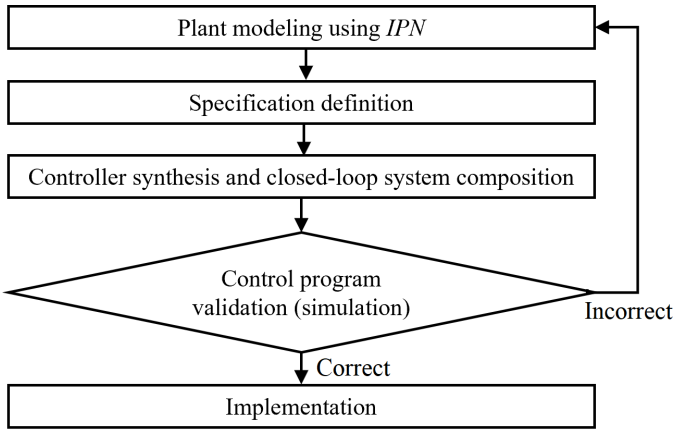

Figure 3. Workflow for the synthesis of controllers using regulation control framework.

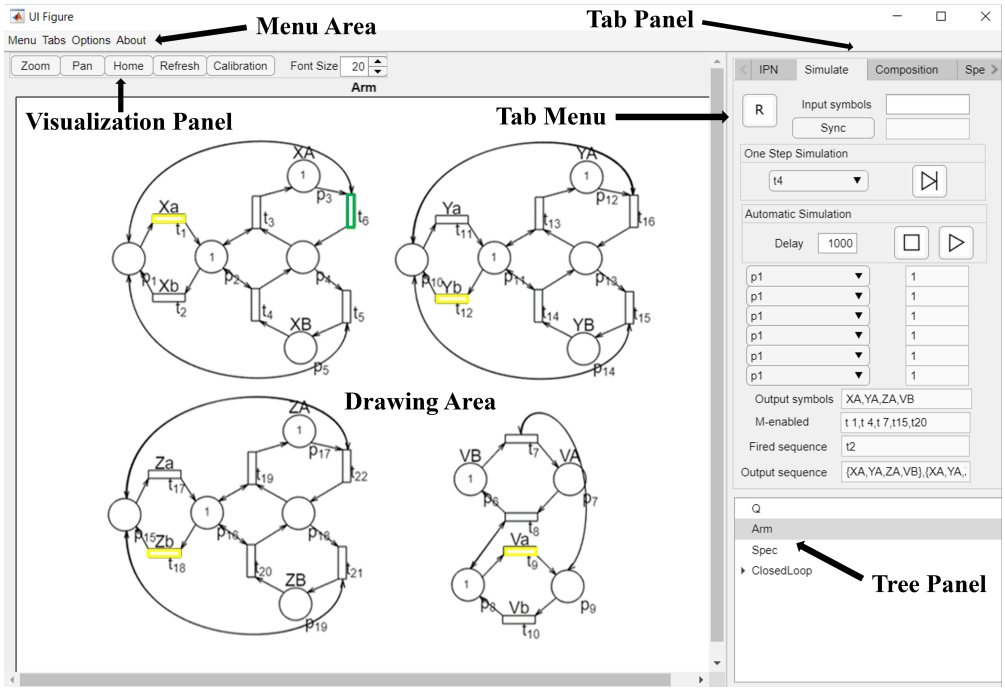

Figure 4. *RCPetri* app interface, showing the simulation of an *IPN* model of a 3-DoF pneumatic arm with a suction cup. The transition marked in green is enabled, while the transitions marked in yellow are marking-enabled. The current output symbols are $\{XA, YA, ZA, VB\}$.

4.1. Defining IPN Models

The first step is to define an IPN model in the "IPN Tab", by adding places and transitions to the "Drawing area". Later, arcs can be defined by clicking on the "Arc icon" at the "Arc subtab" and then selecting the nodes to be connected through the arc. Identity and input symbols can be associated with selected transitions on the "Transition subtab". Output symbols and tokens can be associated with selected places on the "Places subtab". Arcs are defined by a set of control points, new control points are added to a selected arc by clicking on the button ("o") (MATLAB® apps can be slow. MathWorks does not currently provide a solution to speed up apps. In the case of *RCPetri*, the speed depends on the number of nodes and labels to be drawn. It is advised to use the "Zoom tool" to maintain the number of drawn nodes as small as convenient. Additionally, the "Menu Options" provides the possibility to avoid writing labels. Another issue in MATLAB® apps is the mismatch between the cursor and axes coordinates. To correct this, press the "Calibration button" and next click at the center of the red cross).

Once an IPN is drawn, a name must be given and the "Generate button" must be pressed. This will compute all the IPN matrices as defined in Section 2, which are packed in a MATLAB® struct variable with the name of the IPN with the fields Draw (information for drawing the IPN), M0 ($\mathbf{M}_0$), I (I), O (O), AO (Σ_O), LP (φ), AI (Σ_I), LT (λ_I), AID (Σ_{Id}), and LTI (λ_{Id}). This variable is accessible and editable from the MATLAB® Workspace, and it is used for simulation, composition, and control synthesis. A struct variable containing a previously generated IPN can be uploaded to the app, by using the "Load from WS button".

Additionally, the "IPN Tab" allows us to design IPNs in a modular way, by composing IPNs previously generated. However, the app does not currently allow us to define neither subnets nor subsystems. Thus, once a set of nets are composed, they integrate one monolithic IPN system. The "Tree panel" at the right lower corner allows us to change the *current* IPN by clicking on the name of the desired IPN.

4.2. Simulation

The "Simulate Tab" is used to simulate the current IPN system. Before starting a simulation, the user must click on the reset button ("R"), in order to compute the initially enabled transitions. The simulation can be driven in three ways: by clicking on the enabled transitions on the drawing area, by selecting the transition to fire and pressing the "Step button", or by pressing the "Play button"; in the last case, a randomly selected enabled transition will fire after a number of milliseconds specified by the user ("Delay"). Marking-enabled transitions are drawn in yellow, while enabled transitions are drawn in green (see Figure 4). External input symbols can be specified by the user at any time in the "Input Symbols" text field, symbols must be separated by commas (e.g., a, b means the simultaneous indication of a and b). Drop down items allows us to select places, transitions and labels in order to visualize their values during the simulation.

4.3. Specification

The Specification IPN can be defined by drawing and generating an IPN (In this case, remember that each specification transition must have a unique identity symbol, which can be automatically generated in the subtab "IPN > Labels"). A more efficient option is to apply the specification generation algorithm of [34], implemented in the "Spec Gen Tab", which translates a set of standard tables (named "Task table", "Subtask table',' and "Shortcuts table") into an IPN. These tables can be provided in the Book Editor app (which can be open directly from the "Spec Gen Tab") or in an Excel file. To automatically compute the Specification IPN, the app applies two steps: first, the description of subtasks and shortcuts are substituted into the "Task table"; later, Algorithm 2 is applied to translate the "Task table" to an IPN.

Algorithm 2: Building the *IPN* Specification.

Input: Task table.
Output: *IPN* model $\breve{Q}$ representing the Specification.
Define the place $\breve{p}_0$ with one token. Label $\breve{p}_0$ with the symbols that define the first operation.
for *each* $task_k$ **do**
  Define a transition $\breve{t}_k^0$. Connect $\breve{p}_0$ to $\breve{t}_k^0$.
  Remove the first operation from $task_k$, and let r be the number of remaining operations.
  for $j = 1$ *to* r **do**
    Define a place $\breve{p}_k^j$. Connect $\breve{t}_k^{j-1}$ to $\breve{p}_k^j$. Denote as op_k^j the j-th operation of $task_k$. Label $\breve{p}_k^j$ with the symbols that define op_k^j.
    if *guard signals are defined for* op_k^j **then**
      Label $\breve{t}_k^{j-1}$ with the guard signals of op_k^j.
    Define a transition $\breve{t}_k^j$. Connect $\breve{p}_k^j$ to $\breve{t}_k^j$.
  Connect $\breve{t}_k^j$ to $\breve{p}_0$.

The "Task Table" must have three columns for the following:

1. The workstation name (the Plant can be split in workstations).
2. The task name (one workstation can perform several tasks).
3. The operation sequence (each task is defined as a sequence of operations).

Each operation is described by the output symbols that are active when it is completed, these must be separated by commas and enclosed by curly brackets; round brackets can be used to define a set of guard signals that must be active to perform the operation (e.g., $(start, a)\{A, B, C\}$ denotes that both signals *start* and *a* must be active to execute the operation $\{A, B, C\}$, characterized by the simultaneous activation of sensors A, B, and C). The first operation in all tasks must be the home position. For clarity, operations can be defined by using subtasksand shortcuts. Subtasks are defined in the "Subtasks table" as sequences of operations. Shortcuts are defined in the "Shortcuts table" as sets of output symbols. Figure 5 shows the tables in an Excel file that describe a specification for the *IPN* arm model of Figure 4, notice that tasks include subtasks (e.g., *HomeArm*, *TakeIn*, etc.), which are defined using shortcuts. In an Excel file, the Tasks, Subtasks, and Shorcuts tables must be located in the first, second, and third sheets, respectively.

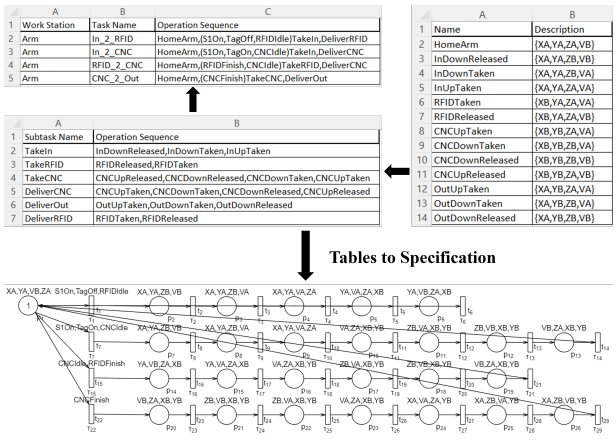

Figure 5. "Task table" (**upper left**), "Subtask table" (**lower left**) and "Shortcuts table" (**right**) in an Excel file describing the Specification of the 3-DoF pneumatic arm of Figure 4. Afterwards, the *IPN* Specification is automatically computed from the tables.

4.4. Control Synthesis

Once the Plant and the Specification are defined as IPN, a regulation controller can be computed in the "Controller Tab", which implements the methodology introduced in [20] and resumed in Section 3, synthesizing an IPN controller that enforces the occurrence of sequences σ_k in the Plant to make the output error to be null. For this, the app modifies the Specification output labels (character # is added to each symbol) in order to distinguish them from the Plant output labels, and later, the labels mapping $\mathbf{\Omega}$ is computed. The computation of the marking mapping $\mathbf{\Pi}$ is performed by the MATLAB® solver $intlinprog()$. Next, sequences σ_k are computed with Algorithm 1. Finally, Algorithm 3 is executed to build the IPN Controller. Moreover, the app generates the IPN representing the control program $(\breve{Q}||\hat{Q})$ and the closed-loop system $(Q||\breve{Q}||\hat{Q})$.

Algorithm 3: Building the IPN Controller.

Input: Plant Q. Specification $\breve{Q}$. Matrix $\mathbf{\Pi}$ and sequences σ_k.
Output: IPN model $\hat{Q}$ representing the Controller.
for *each* $\breve{t}_k \in \breve{T}$ **do**
  %Build a controller subnet $\hat{Q}^{\breve{t}_k}$ as follows:
  Denote the corresponding sequence $\sigma_k = t_k^1 t_k^2 ... t_k^r$, and let $\breve{M}_{k-1}[\breve{t}_k >$
  Define a new transition $\hat{t}_k^0$ with identity label $\breve{\lambda}_{Id}(\breve{t}_k)$. Set $s = 0$.
  for $q = 1$ *to* r **do**
    if t_k^q *is controllable* **then**
      Update $s = s + 1$.
      Define a new place $\hat{p}_k^s$ and connect $\hat{t}_k^{s-1}$ to $\hat{p}_k^s$. Add the symbols $\lambda_I(t_k^q)$ to $\hat{p}_k^s$.
      Define a new transition $\hat{t}_k^s$ and connect $\hat{p}_k^s$ to $\hat{t}_k^s$. Add the symbols $\varphi(M')$ to $\hat{t}_k^s$, where $\mathbf{\Pi}\breve{M}_{k-1}[t_k^1 ... t_k^q > M'$
  Define a new identity label δ_k and add it to $\hat{t}_k^s$.
for *each* $\breve{p}_j \in \breve{P}$ **do**
  Build a controller subnet $\hat{Q}^{\breve{p}_j}$ consisting of a place $\hat{p}_j$, an input transition $\hat{t}_k$ (identity-labeled as δ_k) for each $\breve{t}_k \in^\bullet \breve{p}_j$, and an output transition $\hat{t}_k$ (identity-labeled as $\lambda_{Id}(\breve{t}_k)$) for each $\breve{t}_k \in \breve{p}_j^\bullet$.
Then, the Controller IPN model is $\hat{Q} = \hat{Q}^{\breve{t}_1}||...||\hat{Q}^{\breve{t}_{|\breve{T}|}}||\hat{Q}^{\breve{p}_1}||...||\hat{Q}^{\breve{p}_{|\breve{P}|}}$

Example 2. *Figure 6 shows the automatically generated IPN control program (i.e., the synchronization of Specification and Controller) for the Plant of Figure 4 and the Specification of Figure 5. Let us explain the behavior of the closed-loop system. For this, keep in mind that the IPN Plant output symbols are read by the IPN control to enable control transitions, and vice versa. At the initial state, the Plant provides the sensor signals* $\{XA, YA, VB, ZA\}$ *(operation HomeArm at the Specification tables). The first task (In_2_RFID) starts when the guard signals* $\{S1On, TagOff, RFIDIdle\}$ *are simultaneously active, leading to the firing of the control transition* t_1*, removing tokens from* p_1 *and* p_{11} *and marking control places* p_2 *and* p_7*. The marking of the control place* p_7 *indicates the output symbol Zb, i.e., the controller commands the firing of a Plant transition labeled with Zb (*t_{18} *in Figure 4, which represents energizing coil b of the valve of actuator Z), allowing the Plant to reach the state characterized by the symbols* $\{XA, YA, VB, ZB\}$ *(i.e., the vertical actuator Z is down, corresponding to operation "InDownReleased"). These symbols coincide with those of the control transition* t_7*, becoming enabled and thus firing, moving a token from* p_7 *to* p_{12}*. Next, the control transition* t_2 *becomes enabled and it fires, removing tokens from* p_2 *and* p_{12} *and marking the control places* p_3 *and* p_8*. Place* p_8 *indicates to the Plant to fire the transition labeled with Va (*t_9 *in Figure 4, which represents energizing coil a of the valve of the suction cup), allowing us to reach the Plant state characterized by the following symbols:* $\{XA, YA, VA, ZB\}$ *(i.e., the suction cup V*

is active, corresponding to operation "InDownTaken"). These symbols coincide with those of the control transition t_8, becoming enabled and firing, moving a token from p_8 to p_{13}. Thus, the control transition t_3 becomes enabled and it can fire. This process is repeated until the first task is executed, and then the closed-loop system (both the Plant and the control program) returns to the initial state.

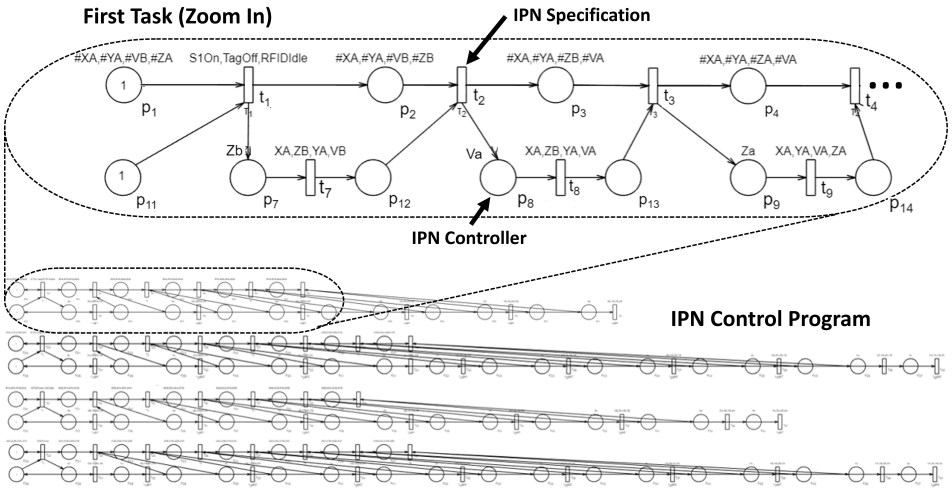

Figure 6. *IPN* control program generated for the 3-DoF pneumatic arm model of Figure 4 and the four tasks defined in the Specification of Figure 5. Due to the size of the *IPN*, it is shown in detail (zoomed in) with the first instructions for the first task.

4.5. PLC Code Generation

For implementation, the *IPN* control program $(\check{Q}||\hat{Q})$ can be translated to PLC code in the "PLC Code Tab". Two languages are supported, Instruction List (IL) and Structured Text (ST). In both cases, the application generates the code in a .txt file that is saved in the current MATLAB® folder. The user can specify the input/output PLC ports for each symbol by means of a "Symbols Table", which is provided by either the Book Editor app table or an Excel file (in the fourth sheet). The table format in both cases is equal, having four columns: input symbol, input port address, output symbol, and output port address.

Let us explain the translation algorithm. Each place p_i of the *IPN* program is associated with a memory bit 'M1x', where x represents the number i. First, a reset button is added, which sets the memories associated with the initially marked places to 1, and the other memories to 0. The second step codifies the dynamic behavior of the *IPN* program, each t_j is codified following the structure: *If* Precondition *Then* Postcondition, where the Precondition is such that all the memories associated with the input places of t_j must be 1 and all the input ports associated with the symbols in $\lambda_I(t_j)$ must be active, and the Postcondition is that all the memories associated with the input places of t_j are set to 0 and all the memories associated with the output places of t_j are set to 1. Finally, each output port associated with an output symbol $o_i \in \Sigma_O$ is activated when the memory associated with a place in $\{p|o_i \in \varphi(p)\}$ is 1.

4.6. Modbus TCP/IP and OPC UA

The app can synchronize input/output symbols with other software or hardware via Modbus TCP/IP or OPC UA. The app works in a similar way to Client. The "Modbus TCP/OPC-UA Client Tab" allows us to configure the connection. For this, a "Symbols table" must be provided, which relates the input/output symbols with the input/output ports (in the case of Modbus TCP/IP, the ports are numbered consecutively; in the case of OPC UA, the ports must be defined on the OPC UA server as Boolean variables). A Write/Read panel is included to help the user to check for the connection and the correct port assignation.

Once the app is linked, the "Simulate Tab" will allow us to simulate the *IPN* synchronized with the server. For this, the "Reset button" (R) must be pressed and later the

"Sync button" must be pushed. The input signals (input symbols) will be shown in a text field. Signals are read with a frequency specified at the "Refresh Delay text field" in the "Modbus/OPC-UA Client Tab" (delay between consecutive readings in *ms*). When an output symbol is generated during the simulation, this is immediately converted to a signal according to the symbols table and sent to the server.

5. Application Examples

In this section, three different examples are presented to illustrate different features of the *RCPetri* app, involving representative devices and automation systems: a pneumatic system, a process under logic control, and a robotic manufacturing system.

5.1. Pneumatic Arm

This example describes the application of the complete workflow of Figure 3 for the synthesis and implementation of a regulation controller for the electro-pneumatic arm with three degrees of freedom (3-DoF), as shown in Figure 7, which belongs to a small-scale manufacturing cell that assembles wood parts, named caps. According to the workflow, the first step is to obtain the Plant model. For this, the methodology of [19] was used, obtaining the IPN depicted in Figure 4, which consists of four submodels, one for each double-effect pneumatic actuator and another for the suction cup.

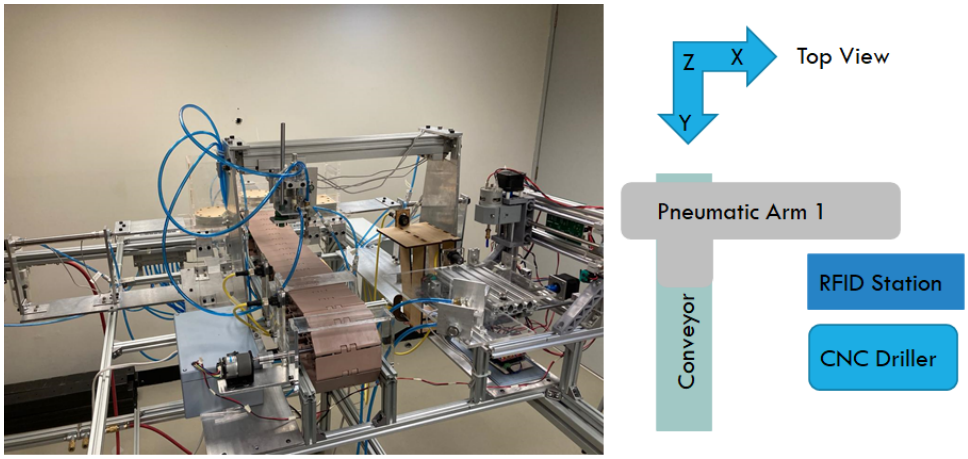

Figure 7. Small-scale manufacturing cell. The pneumatic arm has 3-DoF controlled by the same number of double-effect pneumatic actuators and a suction cup. This arm moves caps between the conveyor, the RFID station, and the CNC station (see the schematic from top view at the right).

The arm must perform four different tasks:

1. If there is a cap at the conveyor's In position without a radio-frequency identification (RFID) tag, then the arm moves it to the RFID station (to attach a RFID tag to it);
2. If the cap has a RFID tag, then the arm moves it to the computer numerical control (CNC) station (to drill 10 holes on it);
3. If the RFID station has collocated a tag on a cap, then the arm moves the cap to the CNC station;
4. If the CNC has drilled a cap, then the arm moves it to the conveyor's Out position.

This Specification was written in the tables depicted in Figure 5. For instance, the first task In_2_RFID starts from the $HomeArm$ state, characterized by $\{XA, YA, ZA, VB\}$. This task is enabled by the signals $S1On$ (the cap is at the conveyor's In position), $TagOff$ (the cap does not have a RFID tag) and $RFIDIdle$ (the RFID station is available). When these are active, the subtask $TakenIn$ must be executed, defined as the operation sequence $InDownReleased$, $InDownTaken$ and $InUpTaken$, characterized by $\{XA, YA, ZB, VB\}$ (i.e., at the In position, the suction cup is down and inactive), $\{XA, YA, ZB, VA\}$ (i.e., at the In position, the suction cup is down and active) and $\{XA, YA, ZA, VA\}$ (i.e., at the In position,

the suction cup is up and active), respectively. The *IPN* Specification depicted in Figure 5 was automatically computed with the app.

Next, the Controller, the control program, and the closed-loop behavior were computed with the "Controller tab". The obtained control program is depicted in Figure 6 and described at the end of Section 4.4, which was translated to an IL PLC code with the "PLC Code tab", obtaining 554 code lines. This code was used to implement the program in a PLC connected to the actual pneumatic arm, obtaining the desired closed-loop behavior, depicted in Figure 8. Let us remark that the engineer only provided the model of Figure 4 and the Specification tables of Figure 5, from this point, the app automatically transformed the tables to the *IPN* model, automatically computed the *IPN* control program of Figure 6, and automatically translated it to PLC code.

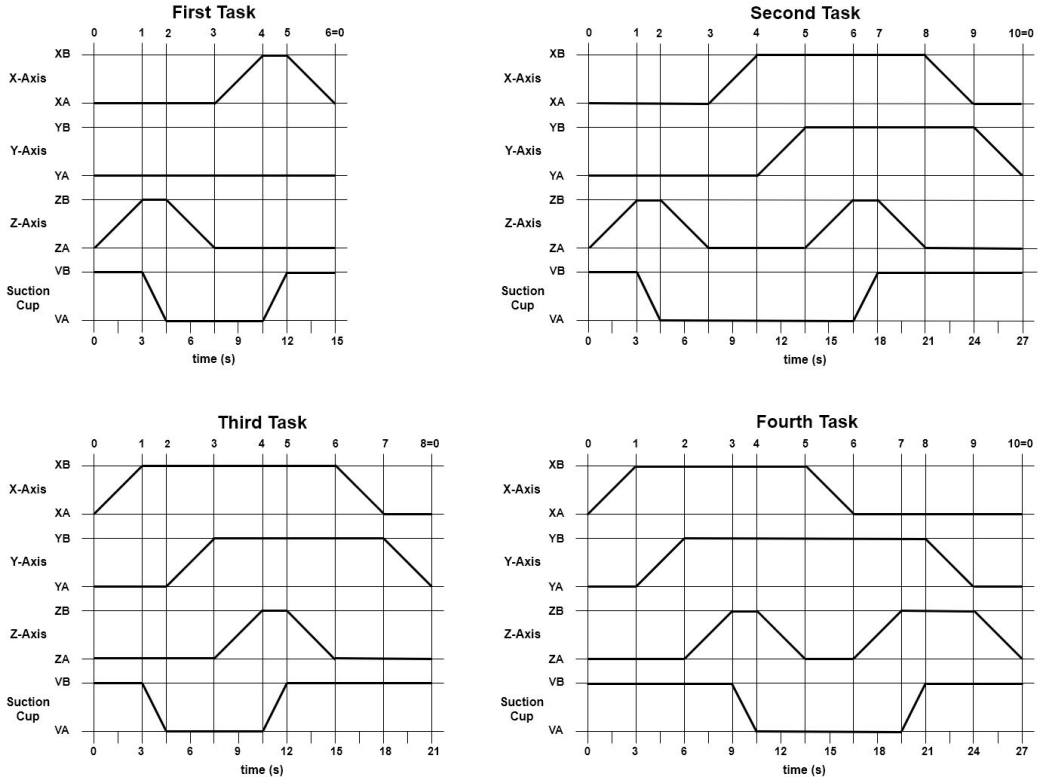

Figure 8. Displacement–time diagrams showing the sequences performed by each actuator during each of the four tasks defined in the Specification. Times are approximated, obtained by visual inspection.

5.2. Tank Filling Process

This example exhibits the use of *RCPetri* for the design and simulation of a logic controller for a tank filling process, simulated in the SCADA software LabVIEW®, in real-time through the Modbus TCP/IP protocol.

The tank filling process considered here (adapted from [35]) is described by the piping and instrumentation diagram (P&ID) of Figure 9. The Plant model was obtained by using the industrial process modeling methodology presented in [21]. In this case, the sensors and actuators were modeled as two-state devices. Tank levels were model as variables with three discrete states (empty, low, high), excepting the water tank that also includes the state medium. These states can be inferred by level sensors LT, for instance, for Soap Tank 101, both sensor LT-1 and sensor LT-2 are inactive when the tank is at the empty state, named T1empty; sensor LT-1 is active and LT-2 is inactive at low state, named T1low; both sensors LT-1 and LT-2 are active at the high state, named T1high. In order to ensure controllability, additional input symbols δ were added to transitions representing a level change, these

allow actuators' controllable transitions to preempt level transitions (δ means waiting until the level change).

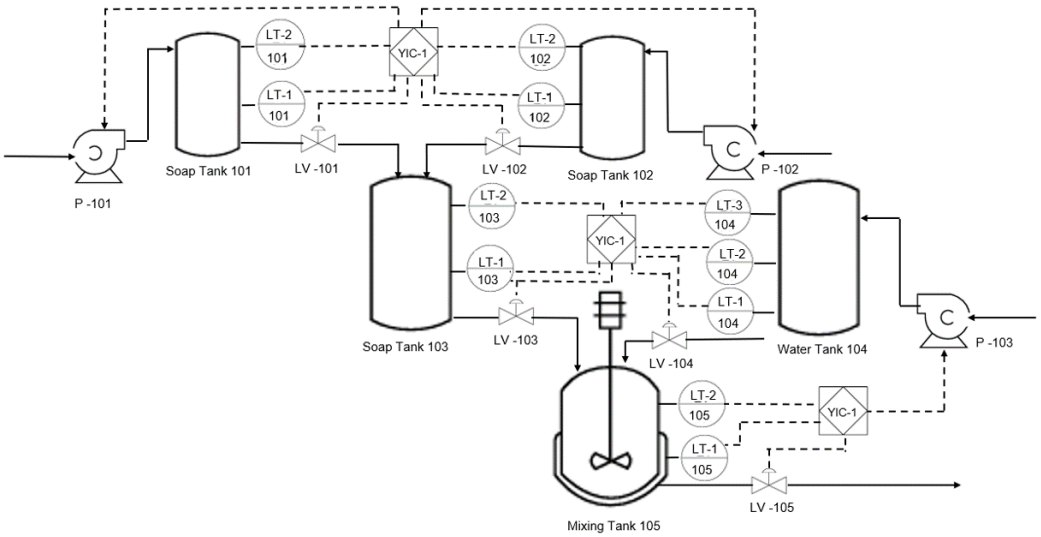

Figure 9. *P&ID* of a tank filling process (Plant for the tank filling process example).

For the Specification, two different filling sequences were considered. In the first one, called "Small", Soap Tank 103 is supplied by Soap Tank 101, the Water Tank 104 is supplied up to the middle level, next the fluids of Soap Tank 103 and Water Tank 104 are mixed in the Mixing Tank 105. In the second sequence, called "Large", Soap Tank 103 is filled by both Soap Tank 101 and Soap Tank 102, while Water Tank 104 is filled up to the high level, then the fluids are mixed in the Mixing Tank 105. The Specification is described in the tables in Figure 10.

Work Station	Task Name	Operation Sequence
Tanks	Small	EmptyTanks,(Small)OneSoapTank,WaterTankMed
Tanks	Large	EmptyTanks,(Large)TwoSoapTank,WaterTankHigh

Subtask Name	Operation Sequence
OneSoapTank	SoapTank-101H,SoapTank-103L
TwoSoapTank	SoapTank-101H,SoapTank-102H,SoapTank-103H
WaterTankMed	WaterTank-104M,MixingTank-105L
WaterTankHigh	WaterTank-104H,MixingTank-105H

Name	Description
EmptyTanks	{T1empty,T2empty,T3empty,T4empty,T5empty}
SoapTank-101H	{T1high,T2empty,T3empty,T4empty,T5empty}
SoapTank-103L	{T1empty,T2empty,T3low,T4empty,T5empty}
WaterTank-104M	{T1empty,T2empty,T3low,T4medium,T5empty}
MixingTank-5L	{T1empty,T2empty,T3empty,T4empty,T5low}
SoapTank-102H	{T1high,T2high,T3empty,T4empty,T5empty}
SoapTank-103H	{T1empty,T2empty,T3high,T4empty,T5empty}
WaterTank-104H	{T1empty,T2empty,T3high,T4medium,T5empty}
MixingTank-105H	{T1empty,T2empty,T3empty,T4empty,T5high}

Figure 10. Tasks, Subtasks, and Shortcuts tables describing the Specification for the Plant of Figure 9.

Once the *IPN* Specification was synthesized, the Controller was computed. The *IPN* closed-loop system is validated by simulation for implementation. The Plant was simulated on LabVIEW® as a hybrid dynamical system (the tank levels evolve with continuous dynamics while pumps, valves and sensors evolve as state machines). Figure 11 depicts the tank filling dynamics during the "Large" sequence, under the control program being simulated in the app and connected through Modbus TCP/IP. Note that at the end of the filling of Tank 101, the control program allows the Tank 102 to start filling immediately. The simulation showed the effectiveness of the control program, as well as the interaction between *RCPetri* and other software (in this case LabVIEW®) in real-time. Figure 12 shows the simulation of the Plant synchronized through Modbus TCP/IP with the *IPN* control program, when the Water tank is filling the Mixing Tank during the "Large" task sequence.

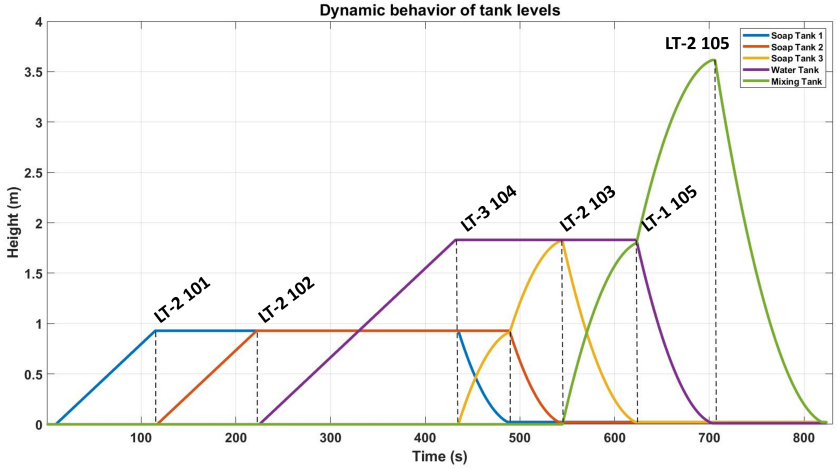

Figure 11. Dynamic behavior of the tanks levels through the "Large" sequence. First, Soap Tank 1 (blue) is filled until sensor LT-2 101 is active; next, Soap Tank 2 (orange) is filled until LT-2 102 is active; then, Water Tank (violet) is filled until LT-3 104 is active; then, both Soap Tank 1 and Soap Tank 2 are drained into Soap Tank 3 (yellow); after this, both Soap Tank 3 and water Tank are drained into Mixing Tank (green); finally the Mixing Tank is drained and the system returns to the initial state.

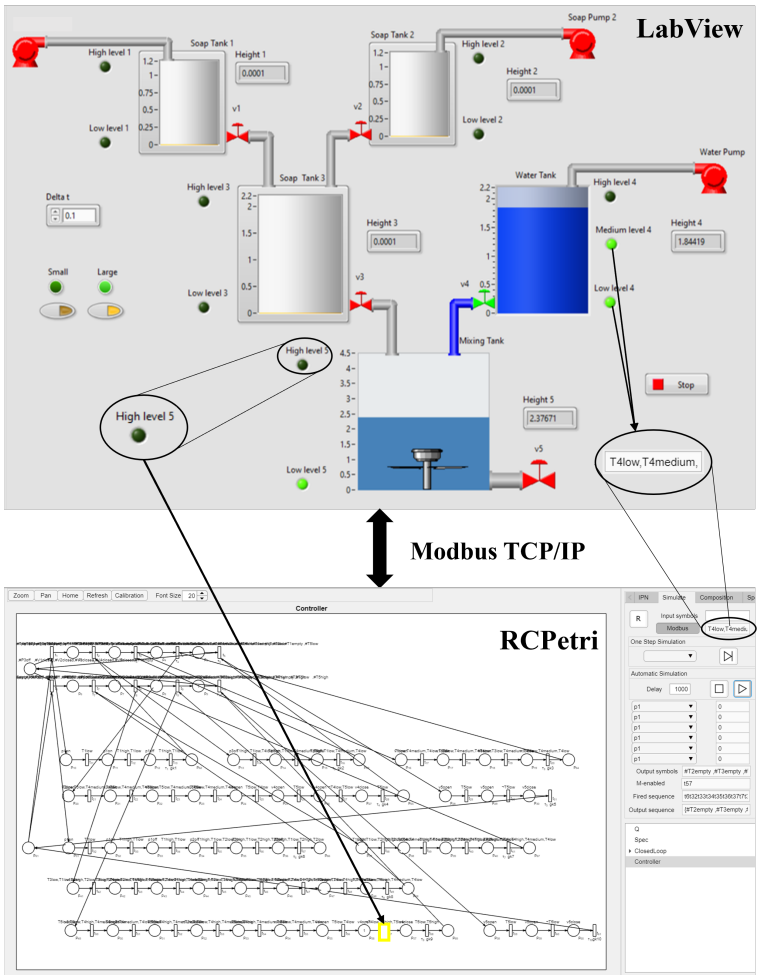

Figure 12. Simulation and control of the tank filling process. When the high-level sensor LT2-105 is activated, it will send a signal for the enabled transition to be fired. On the other hand, in the "Simulation Tab" the user can see the output symbols present in the plant.

5.3. Robotic Manufacturing System

In this example, a manufacturing system composed of two robotic cells is considered. The objective is to show the effectiveness of *RCPetri* for the implementation of a *decentralized control scheme* using the OPC UA protocol. For this, the Plant is simulated in the PLC training platform FACTORY I/O (see Figure 13), and two controllers are simulated in different app instances.

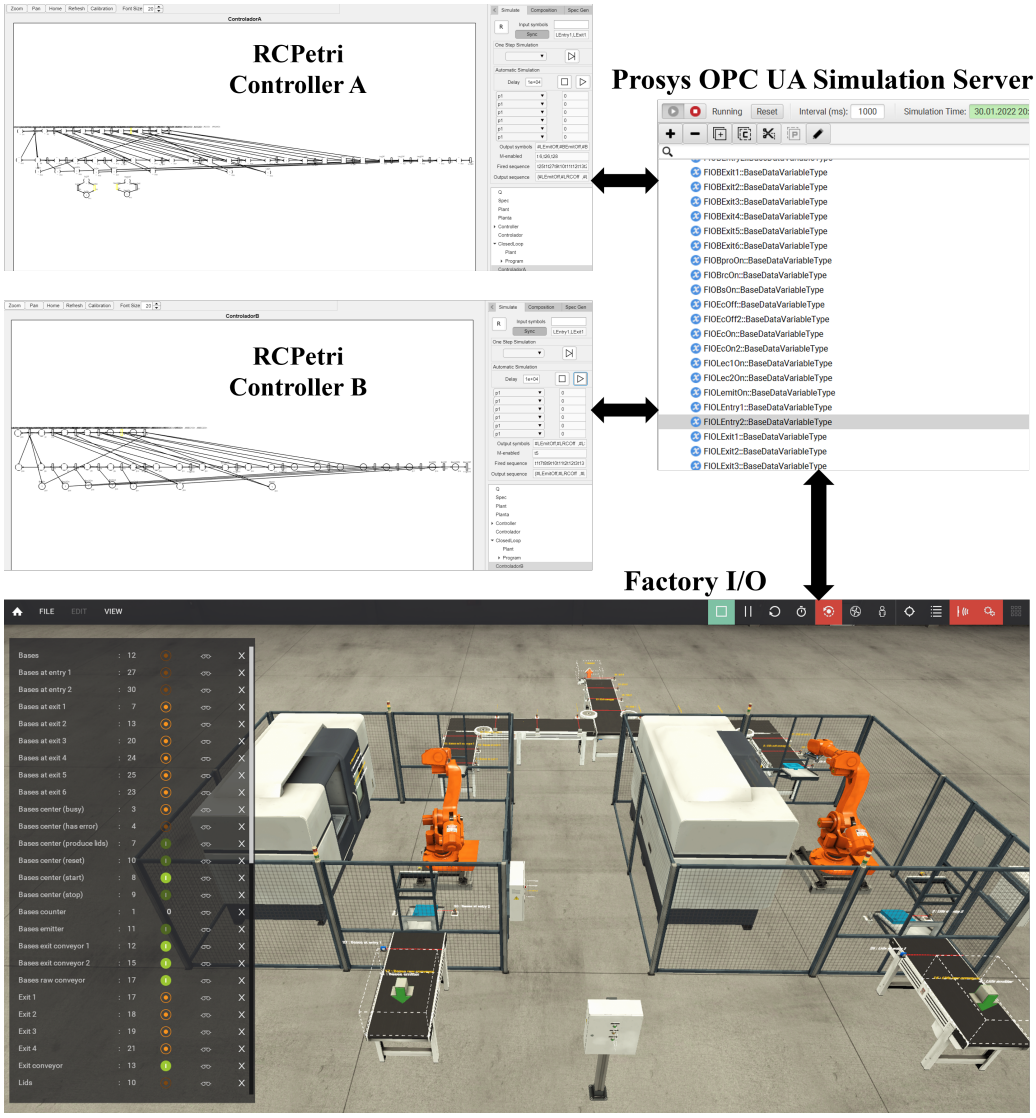

Figure 13. Simulation and control of the robotic manufacturing system. Controller A drives the lids production (left cell), while Controller B drives the bases production (right cell). Each control program is running in a different instance.

Each robotic cell is composed of a raw parts emitter (*Lemit* and *Bemit*), an input conveyor belt (*Lrc* and *Brc*), a robotic arm (*Ls* and *Bs*), a CNC machine (*Lpro* and *Bpro*), and two output conveyor belts (Lec_1, Lec_2, Bec_1 and Bec_2). In addition, both cells share an output conveyor belt *Ec* and a manufactured parts remover *Rem*. Each robotic arm has a predefined program, which consists of moving a part from the entry to the CNC machine, wait for processing, and then moving the part to the exit. Each CNC machine can manufacture either bases or lids. Four sensors, Len_1, Len_2, Ben_1, and Ben_2, indicate the presence of raw parts at different positions on the input conveyors, while the sensors $Lex_1, \ldots, Lex_6$, $Bex_1, \ldots, Bex_6$ indicate the presence of manufactured parts on the output conveyors (Some sensors send a 1 as a logic signal when no object is present (sending a 0

in the opposite case), this can be handled in the app by using negation symbols ¬ in the output labels (e.g., $\neg Lex_1$)).

The Plant was modeled according to the methodology of [19]. Output labels were added to places representing actuator states, describing virtual sensors that allow the computation of controllable sequences. The system was required to perform lids production in the left cell and bases production in the right cell. This desired behavior was translated into two different *IPN* Specifications, one Specification for the production of lids and the other for the production of bases. Later, each corresponding *IPN* Controller was synthesized, obtaining a *decentralized control scheme*, as explained in [19].

Figure 13 displays the control simulation. Here, the Plant under the two synthesized control programs is running in FACTORY I/O. Both control programs are executed simultaneously, each one running in a different *RCPetri* instance, synchronized through the OPC UA protocol. In this case, the Prosys OPC UA Simulation Server platform was used. This communication architecture made possible to effectively implement a *decentralized control scheme*.

6. Conclusions and Future Work

In this work, we have presented the main features of the *RCPetri* app, which allows us to efficiently and graphically define *IPN* models, simulate *IPN*s, automatically generate *IPN* specifications, automatically synthesize *IPN* controllers using the regulation control approach, translate *IPN* control programs into PLC code, and communicate with other software/hardware by using the Modbus TCP/IP and OPC UA protocols. In this way, the app can assist users to efficiently design and implement controllers through the complete control design workflow. In order to illustrate the features of the app, three examples were presented, being representative of common industrial automation systems: the design and PLC implementation of a controller for a pneumatic arm by applying the complete control design workflow; the design and simulation of a logic control of a filling tanks system, showing the synchronization of the *IPN* controller simulation in the app with the tanks simulation in LabVIEW® ; finally, the design and simulation of a system composed of two robotic cells, showing the possibility to synchronize two instances of the app, each one controlling a different part of the Plant, with the Plant simulation in the software FACTORY I/O via the OPC UA protocol, obtaining a decentralized control architecture.

There are some aspects that have been identified as limitations that will be addressed in future releases. In particular, MATLAB® apps tend to be slow when drawing several items, thus the drawing of large nets in the app becomes impractical. The current *IPN* model does not integrate time information, which becomes relevant in some practical applications. Moreover, even if the app can be used to define and simulate unsafe nets, the current control synthesis algorithm cannot deal with unsafe nets. In future releases, methods for dealing with large nets and verifying specifications and controllers will be also included.

Author Contributions: Methodology, C.A.A.-G.; software, A.C.-G.; validation, J.M.C.-D.; investigation, C.R.V. All authors have read and agreed to the published version of the manuscript.

Funding: The research leading to these results has received funding from the Conacyt Program for Education, project number 288470.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Murata, T. Petri Nets: Properties, Analysis and Applications. *Proc. IEEE* **1989**, *77*, 541–580. [CrossRef]
2. Silva, M. Introducing Petri nets. In *Practice of Petri Nets in Manufacturing*; Chapman & Hall: London, UK 1993; pp. 1–62.
3. David, R.; Alla, H. *Discrete, Continuous, and Hybrid Petri Nets*; Springer: Berlin/Heidelberg, Germany, 2010.
4. Ng, K.M.; Reaz, M.B.I.; Ali, M.A.M. A review on the applications of Petri nets in modeling, analysis, and control of urban traffic. *IEEE Trans. Intell. Transp. Syst.* **2013**, *14*, 858–870. [CrossRef]
5. Kabir, S.; Papadopoulos, Y. Applications of bayesian networks and Petri nets in safety, reliability, and risk assessments: A Review. *Saf. Sci.* **2019**, *115*, 154–175. [CrossRef]
6. Liu, F.; Heiner, M.; Gilbert, D. Fuzzy Petri nets for modeling of Uncertain Biological Systems. *Brief. Bioinform.* **2020**, *21*, 198–210.
7. Cavone, G.; Dotoli, M.; Seatzu, C. A Survey on Petri Net Models for Freight Logistics and Transportation Systems. *IEEE Trans. Intell. Transp. Syst.* **2018**, *19*, 1795–1813. [CrossRef]
8. Idel Mahjoub, Y.; Chakir El-Alaoui, E.h.; Nait-Sidi-Moh, A. Logistic Network Modeling and Optimization: An approach based on (MAX,+) algebra and coloured Petri Nets. *Comput. Ind. Eng.* **2021**, *158*, 107341. [CrossRef]
9. Campos, J.; Seatzu, C.; Xie, X. *Formal Methods in Manufacturing*; CRC Press: Boca Raton, FL, USA 2014.
10. Giua, A.; DiCesare, F. Supervisory design using Petri nets. In Proceedings of the 30th IEEE Conference on Decision and Control, Brighton, UK, 11–13 December 1991; pp. 92–97.
11. Giua, A.; DiCesare, F.; Silva, M. Generalized Mutual Exclusion Constraints on Nets with Uncontrollable Transitions. In Proceedings of the 1992 IEEE International Conference on Systems, Man, and Cybernetics, Chicago, IL, USA, 18–21 October 1992; pp. 974–979.
12. Moody, J.O.; Antsaklis, P.J. *Supervisory Control of Discrete Event Systems Using Petri Nets*; Kluwer Academic Publishers: Boston, USA 1998.
13. Basile, F.; Cordone, R.; Piroddi, L. Integrated design of optimal supervisors for the enforcement of static and behavioral specifications in Petri net models. *Automatica* **2013**, *49*, 3432–3439. [CrossRef]
14. Li, Z.; Zhou, M. On siphon computation for deadlock control in a class of Petri nets. *IEEE Trans. Syst. Man Cybern. Part A Syst. Hum.* **2008**, *38*, 667–679.
15. Chen, Y.; Li, Z.; Khalgui, M.; Mosbahi, O. Design of a maximally permissive liveness-enforcing Petri net supervisor for flexible manufacturing systems. *IEEE Trans. Autom. Sci. Eng.* **2011**, *8*, 374–393. [CrossRef]
16. Li, Z.; Wu, N.; Zhou, M. Deadlock control for automated manufacturing systems based on Petri nets. *IEEE Trans. Syst. Man Cybern. Part C Appl. Rev.* **2012**, *42*, 437–462.
17. Wang, S.; Wang, C.; Zhou, M.; Li, Z. A method to compute strict minimal siphons in a class of Petri net based on loop resource subsets. *IEEE Trans. Syst. Man Cybern. Part A Syst. Hum.* **2012**, *42*, 226–237. [CrossRef]
18. Giua, A.; Silva, M. Petri nets and Automatic Control: A historical perspective. *Annu. Rev. Control* **2018**, *45*, 223–239. [CrossRef]
19. Vázquez, C.; Gómez-Castellanos, J.; Ramírez-Treviño, A. Petri nets tracking control for electro-pneumatic systems automation. In *International Conference on Informatics in Control, Automation and Robotics*; Lecture Notes in Electrical Engineering; Gusikhin, O., Madani, K., Eds.; Springer: Cham, Swiztherland, 2019; Volume 613, pp. 503–525.
20. Guevara-Lozano, D.; Vázquez, C.; Ramírez-Treviño, A. Towards decentralized tracking control for Petri nets. In Proceedings of the 2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), Zaragoza, Spain, 10–13 September 2019; pp. 428–435.
21. Rozo-Ibañez, D.; Ruiz-León, J.; Guevara-Lozano, D.; Vázquez, C. Petri net modeling of industrial processes from a P&ID description. In Proceedings of the International Conference on Control, Decision and Information Technologies, Prague, Czech Republic, 29 June–2 July 2020; pp. 715–720.
22. Gaona, A.C.; Vazquez, C.R. RCPetri 1.2. 2022. Available online: https://github.com/CRVazquezT/RCPetri/find/main (accessed on 1 February 2022).
23. Chiola, G.; Franceschinis, G.; Gaeta, R.; Ribaudo, M. GreatSPN 1.7: Graphical Editor and Analyzer for Timed and Stochastic Petri Nets. *Perform. Eval. Spec. Issue Perform. Model. Tools* **1995**, *24*, 47–68. [CrossRef]
24. Dingle, J.; Knottenbelt, W.; Suto, T. PIPE2: A Tool for the Performance Evaluation of Generalised Stochastic Petri Nets. *ACM SIGMETRICS Perform. Eval. Rev.* **2009**, *36*, 34–39. [CrossRef]
25. Kelling, C.; German, R.; Zimmermann, A.; Hommel, G. TimeNET: Evaluation tool for non-Markovian stochastic Petri nets. In Proceedings of the IEEE International Computer Performance and Dependability Symposium, Urbana-Champaign, IL, USA, 4–6 September 1996; p. 62.
26. Davidrajuh, R.; Skolud, B.; Krenczyk, D. Performance evaluation of discrete event systems with GPenSIM. *Computers* **2018**, *7*, 8. [CrossRef]
27. Jensen, K.; Kristensen, L.M.; Wells, L. Coloured Petri nets and CPN Tools for modelling and validation of Concurrent Systems. *Int. J. Softw. Tools Technol. Transf.* **2007**, *9*, 213–254. [CrossRef]
28. Westergaard, M.; Kristensen, L.M. The Access/CPN framework: A tool for interacting with the CPN tools simulator. In *International Conference on Applications and Theory of Petri Nets*; Springer: Berlin/Heidelberg, Germany, 2009; p. 313–322.
29. Sessego, F.; Giua, A.; Seatzu, C. HYPENS: A Matlab tool for timed discrete, continuous and hybrid Petri nets. In *International Conference on Applications and Theory of Petri Nets*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2008; Volume 5062, pp. 419–428.

30. Júlvez, J.; Mahulea, C.; Vázquez, C. SimHPN: A MATLAB Toolbox for simulation, analysis and design with hybrid Petri nets. *Nonlinear Anal. Hybrid Syst.* **2012**, *6*, 806–817. [CrossRef]
31. Basile, F.; Carbone, C.; Chiacchio, P. Simulation and analysis of discrete-event control systems based on Petri nets using PNetLab. *Control Eng. Pract.* **2007**, *15*, 241–259. [CrossRef]
32. Kučera, E.; Haffner, O.; Drahoš, P.; Leskovský, R.; Cigánek, J. PetriNet Editor + PetriNet Engine: New Software Tool For Modelling and Control of Discrete Event Systems Using Petri Nets and Code Generation. *Appl. Sci.* **2020**, *10*, 7662. [CrossRef]
33. Ramírez-Treviño, A.; Rivera-Rangel, I.; López-Mellado, E. Observability of Discrete Event Systems Modeled by Interpreted Petri nets. *IEEE Trans. Robot. Autom.* **2003**, *19*, 557–565. [CrossRef]
34. Guevara-Lozano, D.; Vázquez, C.; Ramírez-Treviño, A. Automatic Specification Generation for Tracking Control in Interpreted Petri nets. In Proceedings of the International Conference on Control, Decision and Information Technologies, Prague, Czech Republic, 29 June–2 July 2020; pp. 341–346.
35. Gharte, M. Automation of soap windscreen washer filling machine with PLC and LabVIEW. In Proceedings of the 2016 International Conference on Automatic Control and Dynamic Optimization Techniques (ICACDOT), Pune, India, 9–10 September 2016; pp. 469–472.