applied sciences

MDPI

Article

# A Program Model of Fuzzy Interpreted Petri Net to Control Discrete Event Systems

**Michał Markiewicz * and Lesław Gniewek**

Department of Computer and Control Engineering, Rzeszow University of Technology, Rzeszow 35-959, Poland; lgniewek@prz.edu.pl
* Correspondence: mmarkiewicz@prz.edu.pl; Tel.: +48-178-651-536

Academic Editor: Elli Kartsakli
Received: 20 January 2017; Accepted: 16 April 2017; Published: 22 April 2017

**Abstract:** Using Petri nets (PNs) to control discrete event systems (DES) has many benefits, because of their graphical representations, the possibility of parallel process control, and their formal descriptions. Amongst the different PNs that are applied for this purpose, most have some limitations for visualization. For many of these PNs, another restriction is the length of time between the creation of the control algorithm in the form of a graph and its practical implementation. These two issues can be resolved with one solution called fuzzy interpreted PN (FIPN). This article proposes the use of a program model based on FIPN to control DES and the method for generation of this model using the graphical representation of the net. FIPN offers a better visualization in comparison to discrete PNs and it allows for the quick creation of program code through the application of a simulator called FIPN-SML. This computer tool implements a method that transforms the graphical form of FIPN into Structured Text (ST) language supported by the IEC 61131-3.

**Keywords:** Petri net; Petri net simulation; Petri net modelling; automatic program generation; programmable logic controller; discrete event systems

## 1. Introduction

Although a finite state machine and a finitely recursive process can be used to model discrete event systems (DES) [1,2], researchers began to use Petri nets (PNs). The reasons for this are the graphical representation of PNs, the possibility of parallel process control, the formal description of PNs, increasing complexity of DES, and greater expectations for the analysis and the modelling of DES [3–5]. This led to the creation of many classes of PNs [6]. Additionally, many software programs [7] that enable the analysis of various net properties were created. These tools can be applied to simulate the operations of systems based on PNs and to find their properties automatically. They usually permit the modelling of specific classes of nets. Research on the modelling, simulation, operation, and control of DES through the application of PNs can be divided into two trends. The first is related to the development of the formal methods that refer to general models. The second is also associated with the formal methods, but they refer to the specific programming languages.

Amongst the papers that consider the first trend, some are related to supervisory control that can be based on structural reasoning [8], applied with the use of the hybrid net [9], and used for a system with uncontrollable and unobservable transitions [10]. Others are associated with the validation of DES, which includes: diagnosis of an asynchronous system [11]; detecting and isolating fault events [12]; fault online detection [13]; fault diagnosis with unobservable transition [14]; model checking based on user specification [15]; formal verification with the use of structural reasoning and general unary hypothesis automaton (GUHA) methods [16]. Other research concerns the application of PNs to specific problems, e.g., to model flexible manufacturing systems (FMS) [17–22] or to model an emergency vehicle preemption system [23]. Different models and problems related to DES can be also found in [4–6,24–26].

The second trend refers mostly to programmable logic controllers (PLCs). For more than two decades, because of their many undeniable advantages, PLCs have become good platforms for the implementation of DES based on PNs [27]. One of the first solutions was Grafcet [28], which is currently available as the international standard IEC 60848:2013 [29]. Next, the Sequential Function Chart (SFC) was created based on Grafcet in IEC 1131-3:1992 (currently IEC 61131:2013 [30]). Many software tools of different manufacturers enable the use of Grafcet and SFC. Both nets allow the application of transitions, to which some two-state conditions can be assigned, which are usually linked to the binary sensors or the outputs of timers. Some areas of application and ways to use Grafcet and SFC are also presented in [6,31–33].

For industrial approaches, Ladder Diagram (LD) language is more often applied than the other languages supported by the IEC 61131-3. This tendency can also be seen in methods that combine PLC programming with PNs [34–43]. However, languages other than LD are also used. In [34], the conversion of automation PN into LD using the token passing logic methodology is described. Signal interpreted PN (SIPN), which can be implemented in LD or Instruction List (IL), is shown in [35–37]. For IL, the PLC program can be generated automatically using a computer tool for the graphical modelling of SIPN. In [38], the method to generate LD code based on control Petri net (CPN), and the specification for the creation of DES through the application of this net is presented. A summary of some solutions combining LD with PNs is presented in [39]. PNs can also be used to validate programs in LD [40] after some conversions from the LD metamodel to time Petri net. In turn, other methods that link PNs to function block diagram (FBD), IL, and LD are proposed in [41–43].

An important aspect related to the design of control systems, especially those that are complex, is performance evaluation. Within this area of science, PNs based on stochastic modelling (stochastic PNs) can be used. The application of Generalized Stochastic Petri Nets (GSPNs) to distributed systems, e.g., flexible manufacturing systems, is proposed in [44]. GSPNs enable performance evaluation by using simulation or numerical methods. A software tool based on GSPN called GreatSPN is presented in [45,46]. Another tool to model stochastic Petri nets is Mobius Framework [47,48]. This framework supports multiple modelling formalisms and modularity. It is based on the so-called atomic model which is composed of state variables, properties, and actions. Another solution supporting multi-formalism modelling and modularity is SIMTHESys [49], which enables the application of the product-form solution theory to multi-formalism compositional modelling techniques. Stochastic Preemptive Time Petri Net (SPTPN) is proposed in [50]. This net is used to validate and conduct a performance analysis of real-time systems, e.g., a digital control system. The formal model of SPTPN is constructed using step semantics.

All solutions presented that combine PLC programs with PNs can be very valuable. However, the discrete PNs are mainly applied. One of the few fuzzy PNs that can be used directly to control DES is fuzzy interpreted Petri net (FIPN) [51]. This net enables the use of analogue and binary signals of processes for diagnosis and control of these processes. It also allows quantitative changes of the resources to be modelled, and the natural interpretation of the fuzzy tokens' position to be maintained. In comparison to discrete PNs, which are usually used to create control systems, the graphic representation of FIPN more precisely shows the dynamics of the net through the possibility of using analogue sensors. To validate and analyse the properties of the net, e.g., liveness and deadlock, the coverability graph can be applied [52]. The algebraic representation [51] and the reachability graph can also be used to investigate the properties. When applying a reachability graph for a net with analogue sensors, it needs to be considered as a net with binary sensors to avoid a state-explosion problem. The computer tool called FIPN-SML facilitates the application of FIPN [53]. This simulator can be used to create a graph of FIPN and to generate program code for PLCs in ST language based on this graph.

As a continuation of [53], which lacked a formal description of the proposed solution, this article may be included in the second trend of dealing with DES. The previous work generally outlines the concept of this solution and gives an example of using it. Thus, this paper proposes a formal description of the PLC program and the method to create this program based on the graphical representation of FIPN. The aim is to describe how this program and method works by giving general formalisms and an illustrative example. The paper is organised as follows. Firstly, comparison to similar works is presented (Section 2). Secondly, the formal description and the conception of FIPN (Section 3) are shown. Next, the formal program model based on FIPN (Section 4) and the method which allows its creation based on the FIPN's diagram (Section 5.1) are described. Finally, the example of program generation in ST language through using FIPN-SML (Section 5.2) is discussed and a brief summary of the results and possible directions of future developments are given (Section 6).

## 2. Comparison to Related Works

In this section, the authors want to raise the issue related to executable specification [54–60]. A graphical representation based on FIPN created in FIPN-SML allows the presentation of the control system behaviour, validation of this system before it is implemented, and clarification of the requirements (that may be initially unclear). This solution enables the automatic construction of executable code and offers a higher abstraction of the designed system. It also reduces the costs and the time needed to develop control systems. All of these advantages are analogical to the cited executable specifications. Moreover, there are some similar approaches to the one proposed in this work [28–39,41]. Implementation of some of them can also be seen as a programming language. They all have unquestionable benefits. Nevertheless, the authors believe that their solution can be characterized by some unique advantages. Based on [51], in literature, fuzzy nets are mainly used to create expert systems, but few deal with the direct application to control [61–63]. These approaches do not have software tool support, do not permit modelling of resources using the net structure, nor do they generate executable code automatically.

When compared to other mentioned solutions [28–39,41], the main advantage of the program model based on FIPN is the possibility of using analogue sensors for the direct control of DES. Contrary to Grafcet and SFC [28–33], FIPN allows resources modelling in the structure of a net, because the weight of arcs and the places capacity can be greater than one, while Grafcet and SFC are based on binary net. Both Grafcet and SFC have similar features to FIPN. They both enable simulation of the created system before it is implemented in PLC and automatic executable code generation based on the simulation model (the graphical representation). However, Grafcet and SFC have some advantages compared to FIPN. Their main advantage is the use of a modularity/hierarchical structure. The authors are aware of this limitation of FIPN and their work related to this subject is currently under consideration by another journal.

Another solution is SIPN [35–37], a binary net, with a formal description. Transformation of the graphical representation to executable code is informal in the examples (IL and LD). As with FIPN, SIPN permits automatic generation of executable code. The main advantages in comparison with FIPN are the automatic investigation of properties through application of the SIPN editor and the possibility of using modularity. Apart from the three solutions mentioned, which seem complete because of software tools, there are some others which propose the conversion of PNs to LD [34,38,39,41]. They have many advantages and they do not take into consideration only binary nets, (e.g., [34]). However, they do not allow automatic generation of executable code based on the graphical representation of a net using a software tool.

To conclude this section, it can be observed that there is no other solution that combines PNs and PLC programming to create control systems and offer the possibility of using analogue sensors, software tool support, resources modelling by the structure of the net, and automatic generation of executable code. However, some new functionalities need to be implemented to see FIPN-SML as a complete solution.

## 3. The Formal Basis and the Conception of FIPN

Three definitions describe the formal basis of FIPN. The first shows the construction of the net.

**Definition 1.** *The fuzzy interpreted Petri net is the system [51]:*

$$FIPN = (P, T, \Omega, \Psi, R, \Delta, K, W, \Gamma, \Theta, M_0, e),$$

*where:* $P = P' \cup P''$*—is a nonempty finite set of places, where:* $P' = \{p'_1, p'_2, \ldots, p'_{a'}\}$*—is a set of places for processes modelling, and*

$P'' = \{p''_1, p''_2, \ldots, p''_{a''}\}$*—a set of places for resources modelling;*

$T = \{t_1, t_2, \ldots, t_b\}$*—is a nonempty finite set of transitions;*

$\Omega = \{\omega_1, \omega_2, \ldots, \omega_{a'+a''}\}$*—is a nonempty finite set of statements;*

$\Psi = \{\psi_1, \psi_2, \ldots, \psi_b\}$*—is a nonempty finite set of conditions;*

$P, T, \Omega, \Psi$*—where none of these sets have common elements;*

$R \subseteq (P \times T) \cup (T \times P)$*—is the incidence relation that assigns a place to each transition* $t_i \in T (1 \leq i \leq b)$,

*where there is the place* $p' \in P'$ *such that* $(p', t_i) \in R$ *or* $(t_i, p') \in R$;

$\Delta$: $P \rightarrow \Omega$*—is the function that assigns a statement to each place;*

$K : P' \rightarrow 1$ *and* $P'' \rightarrow \aleph\{1\}$*—is the function that assigns a capacity to each place,*

*where:* $\aleph = \{1, 2, \ldots\}$;

$\Gamma$: $T\ \Psi$*—is the function that assigns a condition to each transition;*

$\Theta$: $T \rightarrow [0, 1]$*—is the function that defines the degree to which the conditions corresponding to the transitions t are satisfied;*

$W$: $R \rightarrow \aleph$*—is the weight function that meets two conditions,*

*where:* $W(p, t) \leq K(p)$, *and*

$W(t, p) \leq K(p)$ *(p means p′ or p″);*

$M_0 : P' \rightarrow \{0, 1\}$ *and* $P'' \rightarrow W_+$*—is the initial marking function,*

*where:* $M_0(p''_j) = z_j / K(p''_j)$,

$z_j \in \aleph \cup \{0\}$,

$z_j \leq K(p''_j)$,

$j = 1, 2, \ldots, a''$, *and*

$W_+$*—is a set of non-negative rational numbers;*

*e—is an event that synchronizes the work of all transitions.*

FIPN can be represented as a bipartite graph. An exemplary net is shown in Figure 1. There are two types of places in the net: $p'$-type (for processes modelling) and $p''$-type (for resources modelling). They are drawn as circles. For both types, the marking is a real number from the range [0, 1] located inside the circle. However, the marking of $p''$-type places is presented as a fraction and can store a number of tokens greater than one. The capacity of these places $K(p) > 1$ arranged in the denominator is a normalization coefficient by which the marking value is bounded into the interval [0, 1]. Moreover, statements can be assigned to $p'$-type places to set a value of output variables. Transitions are represented by rectangles, and can be related to binary and analogue sensors. Additionally, some logic conditions can be assigned to synchronized transitions and the arcs which link places with transitions are labelled with weighting factors. The conception of using statements, sensors, and logic conditions is presented at the end of this section and in Section 4.2.

The transfer of markers from input to output places across the transition can begin when the conditions (1) and (2) of Definition 2 are satisfied and ends when conditions (3) and (4) are fulfilled.

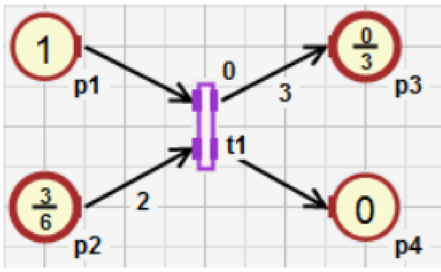


**Figure 1.** Two different input and output places of the transition $t_1$.

**Definition 2.** *The transition $t \in T$ with marking $M{:}P \to [0, 1]$ is enabled from the moment in which the degree to fulfil the condition $\Theta(t) = \vartheta$, which is assigned to the transition, is greater than zero and the following conditions are satisfied [51]:*

$$\forall p \in {}^{\bullet}t, M(p) \geq W(p,t)/K(p) \tag{1}$$

*and*

$$\forall p \in t^{\bullet}, M(p) \leq 1 - W(t,p)/K(p), \tag{2}$$

*to the moment at which:*

$$\exists p' \in {}^{\bullet}t, M(p') = 0 \tag{3}$$

*or*

$$\exists p' \in t^{\bullet}, M(p') = 1, \tag{4}$$

*where:* ${}^{\bullet}t = \{p \in P | (p,t) \in R\}$ *is the set of input places of the transition t, and*
$t^{\bullet} = \{p \in P | (t,p) \in R\}$ *is the set of its output places.*

In FIPN, an analogue signal can be assigned to a transition and its value is normalized into the range [0, 1]. The transfer of a marker across the fired transition is a process whose duration is longer than one clock cycle that synchronizes the net operation. This duration depends on the increment of the sensor value. Such work of transitions permits more precise observation of changes in the controlled system.

The change of the marking for places connected to the enabled transition depends on the increment of the degree to which the condition corresponding to the transition is satisfied. The method that calculates the new marking is described by Definition 3. The transition remains active until the markers are transferred from the input places to the output places of the transition.

**Definition 3.** *Let M be the marking for which the transition $t \in T$ is enabled. The degree $\Theta(t) = \vartheta \in [0,1]$ to which the condition corresponding to the enabled transition is satisfied will be changed by $\Delta\vartheta \geq 0$ and there will be an event e which synchronizes the work of all transitions. The new marking of the net $M'$ is computed by the following rule [51]:*

$$M'(p) = \begin{cases} M(p) - \dfrac{\Delta\vartheta \cdot W(p,t)}{K(p)} & \text{for} \quad p \in {}^{\bullet}t \setminus t^{\bullet}, \quad (5) \\ M(p) + \dfrac{\Delta\vartheta \cdot W(t,p)}{K(p)} & \text{for} \quad p \in t^{\bullet} \setminus {}^{\bullet}t, \quad (6) \\ M(p) - \dfrac{\Delta\vartheta \cdot [W(p,t) - W(t,p)]}{K(p)} & \text{for} \quad p \in {}^{\bullet}t \cap t^{\bullet}, \quad (7) \\ M(p) & \text{for} \quad p \notin {}^{\bullet}t \cup t^{\bullet}. \quad (8) \end{cases}$$

*The increment $\Delta\vartheta < 0$ does not introduce any changes in marking of the net.*

Figure 2 shows the exemplary change of the marking for places connected to the active transition $t_{1.}$ The increment of the degree to which the condition corresponding to the transition $t_1$ is satisfied has the value of $\Delta\vartheta = 0.2$ at the time when the synchronization signal arrives. Such a change of the marking gives a more precise visualization in comparison to discrete PNs and enables the monitoring of progress in the movement of tokens between the input and output places across the transition.

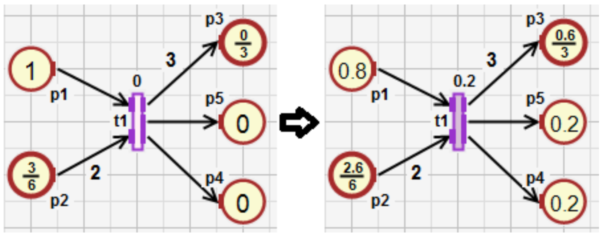


**Figure 2.** The change of the marking for the input and output places for $\Delta\vartheta = 0.2$.

At the end of this section, a simple example is shown to clarify the main advantage of FIPN in comparison to classic Petri nets. In Figure 3a–d, the tank $TK_1$ that can be filled using the valve $V_1$ is presented, and two different graphs as control systems to fill $TK_1$ are proposed. The first is created using SFC, and the second is based on FIPN. Different levels of liquid in $TK_1$ and corresponding to the states of both systems are also shown. In general, the systems operate in a similar manner. First, the filling of $TK_1$ is started by active step $S_1$/ the place $p_1$ which opens the valve $V_1$ (variable $V_1$ is set to *true*). Next, when the tank $TK_1$ is completely filled, active step $S_2$/place $p_2$ closes the valve (variable $V_1$ is set to *false*). However, both systems differ in one important aspect. While in the system based on SFC a binary sensor to monitor the level of liquid is used ($LLS_1$), the system based on FIPN allows the application of an analogue sensor (the same name $LLS_1$ is used to facilitate a comparison of both systems). First, the filling process of $TK_1$ begins, as shown in Figure 3a. Then, different levels of liquid in $TK_1$ during filling process are presented in Figure 3b. Finally, $TK_1$ is completely filled (see Figure 3d). The control system based on SFC enables only the display of two states: if the tank $TK_1$ is being filled (Figure 3a–c) or is completely filled (Figure 3d), whereas the system based on FIPN shows the actual level of liquid all the time. In addition, FIPN allows actions to be performed based on the actual level of liquid (the current marking of places), e.g., while a mixer is being filled with two liquids stored in tanks, and one of the tanks is being emptied too fast/slow compared to the other one, the mixing time can be increased. Hence, FIPN offers a more precise visualization and control of DES than classic Petri nets.

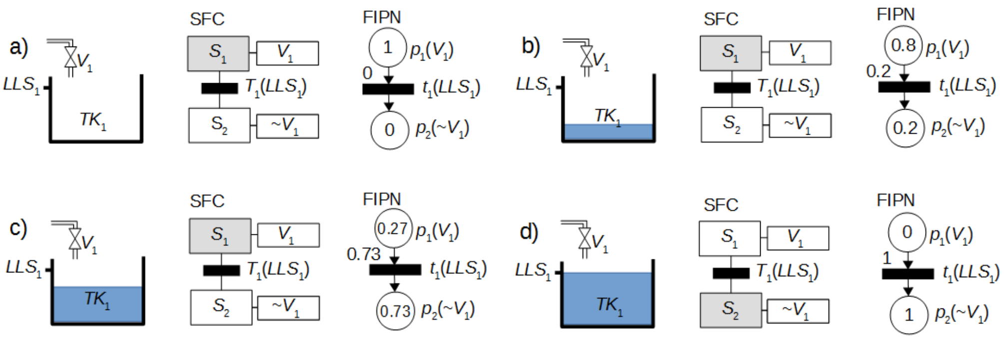


**Figure 3.** Different levels of liquid in the tank $TK_1$ and corresponding to the states of control systems based on Sequential Function Chart (SFC) and fuzzy interpreted PN (FIPN).

## 4. The Program Model of the FIPN

In this section, the program model based on FIPN (called $FIPN^P$) is described. Before it is discussed, the conception of using this model through FIPN-SML to control DES is shown (Figure 4). The graph of FIPN can be created in the simulator. Based on this graph, the program in ST language can be generated and applied to PLC. The important part of the conception is the method of the automatic code generation presented in the next section.

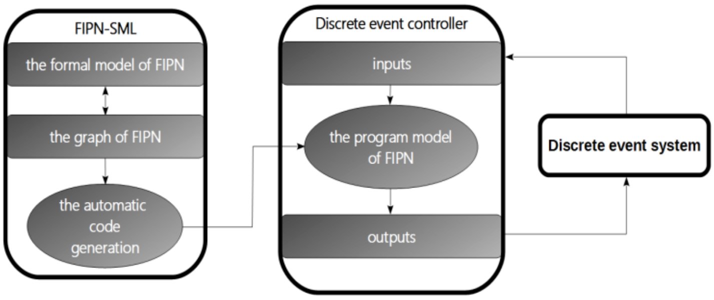


**Figure 4.** The conception of using FIPN to model discrete event systems (DES).

The model $FIPN^P$ is divided into two parts: a set of variables and the functions that operate on these variables. These parts are described below in the separate subsections.

### *4.1. The Program Model of the FIPN—Variables*

In this subsection, the variables set for the program model based on FIPN is presented. The definition of this set is as follows.

**Definition 4.** *The variables in the program model based on FIPN are the system*

$$V^P = \{M^P, Mn^P, K^P, A^P, Tet^P, Dtet^P, W^P, Out^P, In^P\},$$

*where:* $M^P = \{m_1, m_2, \ldots, m_{a'+a''}\}$*—the set of variables that* $\forall p \in P$ *stores the current marking* $M(p)$ *of the place;*

$Mn^P = \{mn_1, mn_2, \ldots, mn_{a'+a''}\}$*—the set of variables that* $\forall p \in P$ *stores the new marking* $M'(p)$ *of the place;*

$K^P = \{k_1, k_2, \ldots, k_{a'+a''}\}$*—the set of variables that* $\forall p \in P$ *stores the capacity* $K(p)$ *of the place;*

$A^P = \{a_1, a_2, \ldots, a_b\}$*—the set of variables that* $\forall t \in T$*stores the state of the transition activation;*

$Tet^P = \{tet_1, tet_2, \ldots, tet_b\}$*—the set of variables that* $\forall t \in T$ *stores the degrees*$\Theta(t) = \vartheta \in [0,1]$ *to which the conditions corresponding to the transitions are satisfied;*

$Dtet^P = \{dtet_1, dtet_2, \ldots, dtet_b\}$*—the set of variables that* $\forall t \in T$ *stores the increments of the degrees* $\Delta\vartheta$ *to which the conditions corresponding to the transitions are satisfied;*

$W^P$*—the set of variables that store the weights of arcs:* $\forall (p_i, t_k) \in R$ *the variable* $p_i\_t_k$ *is created with the value* $W(p_i, t_k)$*, while* $\forall (t_k, p_i) \in R$ *the variable* $t_k\_p_i$ *is created with the value* $W(t_k, p_i)$*, where* $i = 1, 2, \ldots, a' + a''$ *and* $k = 1, 2, \ldots, b$*;*

$Out^P$*—the set of output variables that are set through appropriate changes of the marking;*

$In^P$*—the set of input variables that represent sensors related to transitions.*

*All variables are of real type apart from sets* $A^P$ *and* $Out^P$*, which include Boolean variables.*

According to Definition 4, each place $p_i \in P$ $(i = 1, 2, \ldots, a' + a'')$ is represented by three variables in the program model based on FIPN which are: $m_i$, $mn_i$, and $k_i$ (the current marking, the new marking, and the capacity of the place, respectively). While each transition $t_k \in T$ $(k = 1, 2, \ldots, b)$ is represented by variables $a_k$, $tet_k$, $dtet_k$ that refer respectively to the state of the transition activation, the degree to which the condition corresponding to the transition is satisfied, and the increment of the degree $tet_k$. The set of variables $V^P$ from Definition 4 can be considered as the program implementation of Definition 1. Functions which operate on places and transitions from Definition 1 are replaced by appropriate variables from Definition 4.

### *4.2. The Program Model of the FIPN—Functions*

In this subsection, all functions that belong to the program model of the FIPN are presented. These functions are created based on Definitions 2 and 3. They specify some principles of model operation and can be divided into four groups:

- the functions that calculate the increment of the degree to which the condition corresponding to the transition is satisfied;
- the functions that check if the transitions are granted to be fired or lose the concession;
- the functions that calculate the new marking of places;
- the functions that set output variables based on the new marking of places.

All functions are more clearly described below.

#### 4.2.1. The Calculation of the Degree to which the Condition Corresponding to the Transition Is Satisfied

To limit the values from sensors, the function *Bound* is used. This function changes each value to the range [0, 1]. The limit of each input variable $in_i \in In^P$, that represents the value from the sensor corresponding to the transition $t_i \in T$ is as follows:

$$Bound(in_i) = \begin{cases} 1.0 & \text{for } in_i \geq 1.0 \\ 0 & \text{for } in_i \leq 0.0 \\ in_i & \text{in other case} \end{cases} \tag{9}$$

Based on (9), the function *CalcDtet* is implemented. This function calculates the increment $dtet_i$ of the degree to which the condition corresponding to the transition $t_i \in T$ is satisfied (in two subsequent cycles). It is implemented in the following way:

$$CalcDtet(tet_i, in_i) = \begin{cases} Bound(in_i) - tet_i & \text{for } Bound(in_i) \geq tet_i \\ 0 & \text{in other case} \end{cases} \tag{10}$$

where: $tet_i(tet_i \in Tet^P)$ denotes the variable that stores the degree to which the condition corresponding to the transition is satisfied.

#### 4.2.2. The Checking of the Conditions for the Activation or the Loss of a Concession by a Transition

To check the loss of activation by each transition of $t_i \in T$, the function *LAIn* or *LAOut* is applied. These functions detect the loss of a concession by the transition using one of its input or output $p'$-type places. If the transition has at least one input place of this type, the function *LAIn* is used pursuant to (3):

$$\begin{array}{l} LAIn(a_i, m'_{in(i)}, tet_i)\{ \\ \quad \text{if } a_i \wedge m'_{in(i)} = 0 \; \{ \\ \quad\quad tet_i := 0 \\ \quad\quad \text{return } false \\ \quad \} \text{ else return } true \\ \} \end{array} \tag{11}$$

where: $a_i(a_i \in A^P)$ is a variable that stores the state of the transition activation, $m'_{in(i)}(m'_{in(i)} \in M^P)$ denotes the variable that stores the current marking for the transition's input place that is $p'$-type.

On the other hand, if the transition $t_i$ does not have any input places of $p'$-type, it has at least one output place of this type according to the definition of incidence relation (Definition 1). In this case, the function *LAOut* is used and implemented pursuant to (4) in the following manner:

$$\begin{array}{l} LAOut(a_i, m'_{out(i)}, tet_i)\ \{ \\ \quad \text{if } a_i \wedge m'_{out(i)} = 1\ \{ \\ \quad\quad tet_i := 0 \\ \quad\quad \text{return } false \\ \quad \}\ \text{else return } true \\ \} \end{array} \tag{12}$$

where: $m'_{out(i)} (m'_{out(i)} \in M^P)$ denotes the variable that stores the current marking for the transition's output place that is $p'$-type.

Moreover, the functions (11) and (12) set the degree to which the condition corresponding to the transition is satisfied when the transition loses the concession. The variable $tet_i$ is set to zero.

To examine if the inactive transition $t_i$ can be activated through its $c_i$ ($c_i = card(^\bullet t_i)$) input places, the function $AIn_{c_i}$ is used. This function is implemented based on (1) as follows:

$$\begin{array}{l} AIn_{c_i}(a_i, m_{in(1)}, p_{in(1)}\_t_i, k_{in(1)}, m_{in(2)}, p_{in(2)}\_t_i, k_{in(2)}, \ldots, m_{in(c_i)}, p_{in(c_i)}\_t_i, k_{in(c_i)}) = \\ \begin{cases} true & \text{for } \neg a_i \ \wedge \ (\forall j \in \{1, 2, \ldots, c_i\} : m_{in(j)} \geq p_{in(j)}\_t^i / k_{in(j)}) \\ a_i & \text{in other case} \end{cases} \end{array} \tag{13}$$

where: $m_{in(1)}, m_{in(2)}, \ldots, m_{in(c_i)}$ ($m_{in(j)} \in M^P$) denote the variables that store the current marking for all input places of the transition $t_i$ ($^\bullet t_i$),

$k_{in(1)}, k_{in(2)}, \ldots, k_{in(c_i)}$ ($k_{in(j)} \in K^P$) denote the variables that store the capacity of $^\bullet t_i$,

$p_{in(1)}\_t_i, p_{in(2)}\_t_i, \ldots, p_{in(c_i)}\_t_i$ ($p_{in(j)}\_t_i \in W^P$) denote the variables that store the weights of all arcs from $^\bullet t_i$ to the transition $t_i$.

To check if the transition $t_i$ can be activated through its $d_i$ ($d_i = card(t_i{}^\bullet)$) output places, the function $ActOut_{d_i}$ is used. This function is implemented based on (2) as follows:

$$\begin{array}{l} AOut_{d_i}(a_i, m_{out(1)}, t_i\_p_{out(1)}, k_{out(1)}, m_{out(2)}, t_i\_p_{out(2)}, k_{out(2)}, \ldots, m_{out(d_i)}, t_i\_p_{out(d_i)}, k_{out(d_i)}) = \\ \begin{cases} true & \text{for } \neg a_i \ \wedge \ (\forall j \in \{1, 2, \ldots, d_i\} : m_{out(j)} \leq 1 - t^i\_p_{out(j)} / k_{out(j)}) \\ a_i & \text{in other case} \end{cases} \end{array} \tag{14}$$

where: $m_{out(1)}, m_{out(2)}, \ldots, m_{out(d_i)}$ ($m_{out(j)} \in M^P$) denote the variables that store the current marking for all output places of the transition $t_i$ ($t_i{}^\bullet$);

$k_{out(1)}, k_{out(2)}, \ldots, k_{out(d_i)}$ ($k_{out(j)} \in K^P$) denote the variables that store the capacity of $t_i{}^\bullet$;

$t_i\_p_{out(1)}, t_i\_p_{out(2)}, \ldots, t_i\_p_{out(d_i)}$ ($t_i\_p_{out(j)} \in W^P$) denote the variables that store the weights of all arcs from the transition $t_i$ to the $t_i{}^\bullet$.

If the activation of the transition $t_i$ requires an additional logic condition, the function *LC* is used to avoid conflicts between transitions in the net. This function is implemented in the following manner:

$$LC(a_i, lc_i) = \begin{cases} lc_i & \text{for } \neg a_i \\ true & \text{in other case} \end{cases} \tag{15}$$

where: $lc_i$ denotes the logic condition assigned to the transition $t_i$.

### 4.2.3. The Calculation of the New Marking

The new marking of the $c_i$ input places of each transition $t_i \in T$ is calculated using the function $InM_{c_i}$ implemented based on (5):

$$
\begin{array}{l}
InMn_{c_i}(a_i, dtet_i, mn_{in(1)}, p_{in(1)}\_t_i, k_{in(1)}, mn_{in(2)}, p_{in(2)}\_t_i, k_{in(2)}, \ldots, mn_{in(c_i)}, p_{in(c_i)}\_t_i, k_{in(c_i)})\ \{ \\
\quad \text{if } a_i\ \wedge\ dtet_i > 0\ \{ \\
\qquad mn_{in(j)} := mn_{in(j)} - \frac{dtet_i \cdot p_{in(j)}\_t_i}{k_{in(j)}},\ \forall j \in \{1, 2, \ldots, c_i\} \\
\quad \} \\
\}
\end{array}
\tag{16}
$$

where: $mn_{in(1)}, mn_{in(2)}, \ldots, mn_{in(c_i)}$ ($mn_{in(j)} \in Mn^P$) denote the variables that store the new marking for all input places of the transition $t_i$ ($^\bullet t_i$).

Whereas, the new marking of $d_i$ output places of the transition $t_i$ is computed using the function $OutMn_{d_i}$ created based on (6):

$$
\begin{array}{l}
OutMn_{d_i}(a_i, tet_i, dtet_i, mn_{out(1)}, t_i\_p_{out(1)}, k_{out(1)}, mn_{out(2)}, t_i\_p_{out(2)}, k_{out(2)}, \ldots, mn_{out(d_i)}, t_i\_p_{out(d_i)}, k_{out(d_i)})\ \{ \\
\quad \text{if } a_i\ \wedge\ dtet_i > 0\ \{ \\
\qquad tet_i := tet_i + dtet_i \\
\qquad mn_{out(j)} := mn_{out(j)} + \frac{dtet_i \cdot t_i\_p_{out(j)}}{k_{out(j)}},\ \forall j \in \{1, 2, \ldots, d_i\} \\
\quad \} \\
\}
\end{array}
\tag{17}
$$

where: $mn_{out(1)}, mn_{out(2)}, \ldots, mn_{out(d_i)}$ ($mn_{out(j)} \in Mn^P$) denote the variables that store the new marking for all output places of the transition $t_i$ ($t_i^\bullet$).

Moreover, in (17) the degree to which the condition corresponding to the transition is satisfied is updated $(tet_i)$.

### 4.2.4. The Setting of Output Variables

At the end of each PLC cycle, the output variables from the statement of each $p'$-type place $p_i \in P'$ are set to *true* or *false*, if the new marking of $p_i$ is updated to one. The function $MChg_{n_i}$ is used to set the output variables as follows:

$$
\begin{array}{l}
MChg_{n_i}(m_i, mn_i, out_{i(1)}, lv_{i(1)}, out_{i(2)}, lv_{i(2)}, \ldots, out_{i(n_i)}, lv_{i(n_i)})\ \{ \\
\quad \text{if } m_i \neq mn_i\ \{ \\
\qquad \text{if } mn_i = 1\ \{ \\
\qquad\quad out_{i(k)} := lv_{i(k)},\ \forall k \in \{1, 2, \ldots, n_i\} \\
\qquad \} \\
\qquad m_i := mn_i \\
\quad \} \\
\}
\end{array}
\tag{18}
$$

where: $out_{i(1)}, out_{i(2)}, \ldots, out_{i(n_i)}$ denote the output variables related to the place $p_i$,
$lv_{i(1)}, lv_{i(2)}, \ldots, lv_{i(n_i)}$ denote the logic values (*true* or *false*).

Moreover, the function $MChg_{n_i}()$ updates the variable of the current marking $m_i$ based on the variable of the new marking $mn_i$. The variables of the current marking for all $p'$-type places that are not linked to any output variables and for all $p''$-type places are updated by $MChg_0()$.

## 5. The Method of Automatic Program Generation Using ST Language

In this section, the method of program generation based on FIPN and the implementation of this method for an exemplary net are presented. In the first subsection, the algorithms that create the declaration of variables and the part of the program executed in the cycles are described. The second presents the use of these algorithms in FIPN-SML, which allows automatic generation of the most significant part of the program in ST language.

### *5.1. Algorithms of the Program Generation Based on FIPN*

The method of program generation based on FIPN is divided into two algorithms. The first creates the declaration of variables. At the beginning, an empty buffer is created. Next, for each of the place declarations of the current marking, the new marking and the capacity along with their initial values are added to the text buffer. Then, for each transition, declarations of the following variables are added to the text buffer: degree to which the condition corresponding to the transition is satisfied with an initial value of zero; the increment of this degree has an initial value of zero if the transition is synchronised by a sensor, otherwise it has an initial value of one; the activation of the transition has an initial value of *false*. Finally, the declarations of output variables related to places and input variables related to transitions are inserted into the text buffer. However, before the addition, it is checked to determine whether a declaration of an input or output variable has already been added (by another transition or place) to the text buffer to avoid duplication. Output variables should be initialized with *true* if the initial marking of the place is equal to one and this output variable is set to *true* in the statement assigned to the place. Otherwise it should be initialized with *false*. The first algorithm (Algorithm 1) is as follows:

**Algorithm 1. Create the declaration of variables**

Create empty text buffers $bt_1$, $bt_2$, $bt_3$, $bt_4$ and empty lists $l_1$, $l_2$.
**for each** place $p_i \in P$ **do**
  Add the declaration of the variable $m_i$ with the initial value $M_0(p_i)$ to $bt_2$
  Add the declaration of the variable $mn_i$ with the initial value $M_0(p_i)$ to $bt_3$.
  Add the declaration of the variable $k_i$ with the value $K(p_i)$ to $bt_4$.
**end for**
Add $bt_2$, $bt_3$, $bt_4$ to $bt_1$, and then clear $bt_2$, $bt_3$, $bt_4$.
**for each** transition $t_i \in T$ **do**
  Add the declaration of the variable $tet_i$ with the initial value zero to $bt_2$.
  **if** the condition $\psi_i = \Gamma(t_i)$ assigned to $t_i$ is related to a sensor variable $in_i$ **then**
    Add the declaration of the variable $dtet_i$ to $bt_3$ with the initial value zero.
  **else**
    Add the declaration of the variable $dtet_i$ to $bt_3$ with the initial value one.
  **end if**
  Add the declaration of the variable $a_i$ to $bt_4$ with the initial value *false.*
**end for**
Add $bt_2$, $bt_3$ and $bt_4$ to $bt_1$.
**for each** place $p_i \in P$ **do**
  **for each** output variable $out_k$ that is set using the statement $\omega_i = \Delta(p_i)$ **do**
    **if** the list $l_1$ does not contain the name of $out_k$ **then**
      Add the name of $out_k$ to $l_1$.
      **if** $M_0(p_i) > 0$ **and** $out_k$ is set to *true* in the statement $\omega_i$ **then**
        Add the declaration of $out_k$ to $bt_1$ with the initial value *true*.
      **else**
        Add the declaration of $out_k$ to $bt_1$ with the initial value *false*.
      **end if**
    **end if**
  **end for**
**end for**
**for each** transition $t_i \in T$ **do**
  **if** the condition $\psi_i = \Gamma(t_i)$ assigned to $t_i$ is related to a sensor variable $in_i$ **and** the list $l_2$ does not contain the name of $in_i$ **then**
    Add the name of $in_i$ to $l_2$.
    Add the declaration of $in_i$ to $bt_1$ with the initial value zero.
  **end if**
**end for**
Return the result of the algorithm: $bt_1$.

The second algorithm generates the part of the program that is executed repeatedly during the cycles. At first, an empty buffer is created. Next, for each transition synchronized by a sensor a line of code is added to this buffer. This line calculates the degree to which the condition corresponding to the transition is satisfied. Then, for each transition the next line of code is added that checks whether the transition loses activity or is fired. If the firing of the transition is related to a logic condition, an additional code is inserted to this line. After that, for each transition two lines are added to the text buffer. They compute the new marking of input and output places of the transition. Finally, for each place the line that updates the current marking (based on the new marking) of the place is inserted. Additionally, output variables are set if the current marking of the $p'$-type place is changed to one. The second algorithm (Algorithm 2) is as follows:

**Algorithm 2. Generate the part of the program that is executed repeatedly in the cycles of PLC**

Create an empty text buffer $bt_1$ to store the result of the algorithm.
**for each** transition $t_i \in T$ **do** *//compute the change of a sensor value*
  **if** the condition $\psi_i = \Gamma(t_i)$ assigned to $t_i$ is related to a sensor $in_i$ **then**
    Add to $bt_1$ the line of code that calculates the change of the value from the sensor $\Delta\vartheta_i$ based on (10): $dtet_i := CalcDtet(tet_i, in_i)$
  **end if**
**end for**
Create an empty text buffer $bt_2$ to store the single line of code.
**for each** transition $t_i \in T$ **do** *//check if a transition can be fired or loses the concession*
  **if** $t_i$ has at least one input $p$'-type place $p'_{in(i)}$ , such that $(p'_{in(i)}, t_i) \in R$ **then**
    Add to $bt_2$ the following fragment of code based on (11): $a_i := LAIn(a_i, m'_{in(i)}, tet_i)$
  **else**
    Find the output $p'$-type place $p'_{out(i)}$ of $t_i$, such that $(t_i, p'_{out(i)}) \in R$, and then add to $bt_2$ the following fragment of code based on (12): $a_i := LAOut(a_i, m'_{out(i)}, tet_i)$
  **end if**
  **if** card $({}^\bullet t_i) > 0$ **then**
    Add to $bt_2$ the following fragment of code based on (13): $\wedge AIn_{c_i}(a_i, m_{in(1)}, p_{in(1)}\_t_i, k_{in(1)}, m_{in(2)}, p_{in(2)}\_t_i, k_{in(2)}, \ldots, m_{in(c_i)}, p_{in(c_i)}\_t_i, k_{in(c_i)})$
  **end if**
  **if** card $(t_i{}^\bullet) > 0$ **then**
    Add to $bt_2$ the following fragment of code based on (14): $\wedge AOut_{d_i}(a_i, m_{out(1)}, t_i\_p_{out(1)}, k_{out(1)}, m_{out(2)}, t_i\_p_{out(2)}, k_{out(2)} \ldots, m_{out(d_i)}, t_i\_p_{out(d_i)}, k_{out(d_i)})$
  **end if**
  **if** the condition $\psi_i = \Gamma(t_i)$ assigned to $t_i$ includes a logic condition $lc_i$ **then**
    Add to $bt_2$ the following fragment of code based on (15): $\wedge LC(a_i, lc_i)$
  **end if**
  Add $bt_2$ to $bt_1$ and clear $bt_2$.
**end for**
**for each** transition $t_i \in T$ **do** *//compute the new marking of places connected to a transition*
  **if** $card({}^\bullet t_i) > 0$ **then** *//input places*
    Add to $bt_1$ the code based on (16): $InMn_{c_i}(a_i, dtet_i, mn_{in(1)}, p_{in(1)}\_t_i, k_{in(1)}, mn_{in(2)}, p_{in(2)}\_t_i, k_{in(2)}, \ldots, mn_{in(c_i)}, p_{in(c_i)}\_t_i, k_{in(c_i)})$
  **end if**
    **if** $card(t_i{}^\bullet) > 0$ **then** *//output places*
    Add to $bt_1$ the code based on (17): $OutMn_{d_i}(a_i, tet_i, dtet_i, mn_{out(1)}, t_i\_p_{out(1)}, k_{out(1)}, mn_{out(2)}, t_i\_p_{out(2)}, k_{out(2)}, mn_{out(d_i)}, t_i\_p_{out(d_i)}, k_{out(d_i)})$
  **end if**
**end for**
**for each** place$p_i \in P$ **do** *//update the current marking of places and set outputs variables*
    Take each variable $out_{i(k)}$ and its value $lv_{i(k)}$ from the statement $\omega_i = \Delta(p_i)$ and add to $bt_1$ the line based on (18): $MChg_{n_i}(m_i, mn_i, out_{i(1)}, lv_{i(1)}, out_{i(2)}, lv_{i(2)}, \ldots, out_{i(n_i)}, lv_{i(n_i)})$
**end for**
**return** the result of the algorithm: $bt_1$.

### 5.2. An Example of Program Generation in ST Language

In this subsection, the use of algorithms from the previous subsection is presented. The graphic diagram of FIPN is created in FIPN-SML (Figure 5). There are five places and two transitions in the diagram. To each place of $p'$-type the statement can be assigned that sets the output variables linked to the place, e.g., for the place $p_1$ the variable O1 is set to true and the variable O2 to false. Output variables are set when the value of the current marking is changed to one ($M(p') = 1$). Each transition can be synchronized by a sensor and activated by a logic condition. The transition $t_1$ is synchronized by the sensor $IN_1$ and the activation of the transition $t_2$ is subject to the logic condition $M(p_1) < 1$ (to avoid conflict with the transition $t_1$). From the moment the program is generated in FIPN-SML, it can be copied to PLC software. The basic assumption of the method is that the special library with the implemented functions (9)–(18) was prepared earlier and added to the PLC software. The designer of the net must copy the generated program code and combine the physical inputs and outputs of a controller with the program variables.

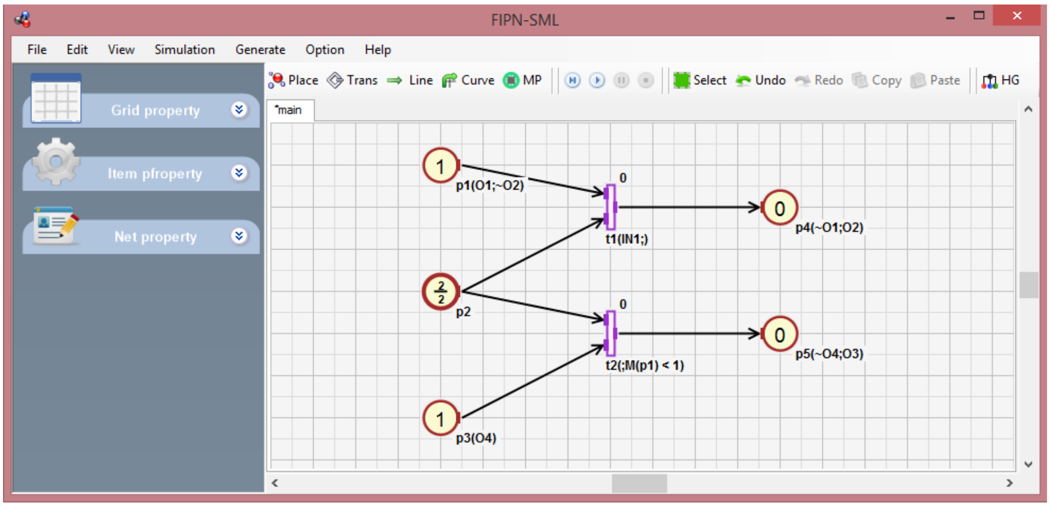


**Figure 5.** View of exemplary FIPN.

Appendix A presents the part of the program created as a declaration of variables, and Appendix B shows the part of the program that is executed repeatedly in the cycles of PLC. Both parts are created based on the net from Figure 5.

## 6. Conclusions

In this paper, the formal description of the PLC program based on FIPN and the method to create this program are proposed. Additionally, the formal basis and conception of using FIPN, the creation of an exemplary diagram based on FIPN in FIPN-SML and the automatic code generation for this diagram are shown. The code generated is in the ST language and can be applied to PLC at low cost and with little effort. The presented approach also allows for a significant reduction of the program implementation time to control DES. The work of the control system can be better visualized using analogue sensors in comparison to the discrete PNs. The application of FIPN allows resources to be modelled by the structure of the net, offers software tool support, and provides automatic generation of executable code. An additional advantage of the formal description presented in this paper is its generality. The proposed formalism does not limit the solution only to the ST language or other languages from IEC 61131:2013, but it provides the opportunity to use it beyond the PLC area.

However, some functionalities still need improvement. Apart from modularity, which is the subject of an author's article considered in another journal, the authors will: extend automatic code generation to other languages supported by the IEC 61131-3; add to the FIPN-SML a component that automatically investigates properties of FIPN; and extend the generated PLC program in regard to the diagnostic module based on [52].

**Author Contributions:** Lesław Gniewek and Michał Markiewicz prepared the conception and the formal description of the proposed solution; they both made the literature review; Michał Markiewicz implemented the proposed solution in FIPN-SML.

**Conflicts of Interest:** The author declares there to be no conflict of interest.

## Appendix A

The part of the program created as a declaration of variables which is generated based on the net from Figure 5:

```
PROGRAM MAIN
VAR
(*Marking*)
      m1:REAL:=1;
      m2:REAL:=1;
      m3:REAL:=1;
      m4:REAL:=0;
      m5:REAL:=0;
(*New marking*)
      mn1:REAL:=1;
      mn2:REAL:=1;
      mn3:REAL:=1;
      mn4:REAL:=0;
      mn5:REAL:=0;
(*Capacity*)
      k1:REAL:=1;
      k2:REAL:=2;
      k3:REAL:=1;
      k4:REAL:=1;
      k5:REAL:=1;
(*Line weight*)
      p1_t1:REAL:=1;
      p2_t1:REAL:=1;
      p3_t2:REAL:=1;
      t2_p5:REAL:=1;
      t1_p4:REAL:=1;
      p2_t2:REAL:=1;
(*Active transition*)
      a1:BOOL:=FALSE;
      a2:BOOL:=FALSE;
(*Tet*)
      tet1:REAL:=0.0;
      tet2:REAL:=0.0;
(*Dtet*)
      dtet1:REAL:=0.0;
      dtet2:REAL:=1.0;
(*Output signal*)
      O1:BOOL:=TRUE;
      O2:BOOL:=FALSE;
      O3:BOOL:=FALSE;
```

```
     O4:BOOL:=TRUE;
(*Input signal*)
     IN1:REAL;
END_VAR
```

## Appendix B

The part of the program that is executed repeatedly in the cycles of PLC and generated based on the net from Figure 5:

```
(*Checking sensors*)
dtet1:=CalcDtet (tet1, IN1);
(*Checking active transition*)
a1:=LAIn (a1, m1, tet1) AND AIn2 (a1, m1, p1_t1, k1, m2, p2_t1, k2) AND AOut1 (a1, m4,
t1_p4, k4);
a2:=LAIn (a2, m3, tet2) AND AIn2 (a2, m3, p3_t2, k3, m2, p2_t2, k2) AND AOut1 (a2, m5,
t2_p5, k5) AND LC(a2, (m1<1));
(*New marking*)
InMn2 (a1, dtet1, mn1, p1_t1, k1, mn2, p2_t1, k2);
OutMn1 (a1, tet1, dtet1, mn4, t1_p4, k4);
InMn2 (a2, dtet2, mn3, p3_t2, k3, mn2, p2_t2, k2);
OutMn1 (a2, tet2, dtet2, mn5, t2_p5, k5);
(*On Marking Changed*)
MChg2 (m1, mn1, O1, TRUE, O2, FALSE);
MChg0 (m2, mn2);
MChg1 (m3, mn3, O4, TRUE);
MChg2 (m4, mn4, O1, FALSE, O2, TRUE);
MChg2 (m5, mn5, O4, FALSE, O3, TRUE);
```

## References


1. Inan, K.; Varaiya, P. Finitely recursive process models for discrete event systems. *IEEE Trans. Autom. Control* **1988**, *33*, 626–639. [CrossRef]
2. Ramadge, P.J.G.; Wonham, W.M. The control of discrete event systems. *Proc. IEEE* **1989**, *77*, 81–98. [CrossRef]
3. Ichikawa, A.; Hiraishi, K. Analysis and control of discrete event systems represented by Petri nets. In *Discrete Event Systems: Models and Applications*; Springer: Heidelberg, Germany, 1988; pp. 115–134.
4. Zhou, M.; Dicesare, F. *Petri Net Synthesis for Discrete Event Control of Manufacturing Systems*; Springer: New York, NY, USA, 2012.
5. Cassandras, C.G.; Lafortune, S. *Introduction to Discrete Event Systems*; Springer: New York, NY, USA, 2009.
6. David, R.; Alla, H. *Discrete, Continuous, and Hybrid Petri Nets*; Springer: New York, NY, USA, 2010.
7. Thong, W.J.; Ameedeen, M.A. A survey of Petri net tools. In *Advanced Computer and Communication Engineering Technology*; Springer: Cham, Switzerland, 2015; pp. 537–551.
8. Barkaoui, K.; Chaoui, A.; Zouari, B. Supervisory control of discrete event systems based on structure theory of Petri nets. In Proceedings of the Computational Cybernetics and Simulation 1997 IEEE International Conference on Systems, Man, and Cybernetics, Orlando, FL, USA, 12–15 October 1997; pp. 3750–3755.
9. Demongodin, I.; Koussoulas, N.T. Representing continuous systems in a discrete-event world. *IEEE Trans. Autom. Control* **1998**, *43*, 573–579. [CrossRef]
10. Moody, J.O.; Antsaklis, P.J. Petri net supervisors for DES with uncontrollable and unobservable transitions. *IEEE Trans. Autom. Control* **2000**, *45*, 462–476. [CrossRef]
11. Benveniste, A.; Fabre, E.; Haar, S.; Jard, C. Diagnosis of asynchronous discrete-event systems: A net unfolding approach. *IEEE Trans. Autom. Control* **2003**, *48*, 714–727. [CrossRef]
12. Genc, S.; Lafortune, S. Distributed diagnosis of discrete-event systems using Petri nets. In *SpringerLink*; Springer: Berlin/Heidelberg, Germany, 2003; pp. 316–336.

13. Ramirez-Trevino, A.; Ruiz-Beltran, E.; Rivera-Rangel, I.; Lopez-Mellado, E. Online fault diagnosis of discrete event systems. A Petri net-based approach. *IEEE Trans. Autom. Sci. Eng.* **2007**, *4*, 31–39. [CrossRef]
14. Cabasino, M.P.; Giua, A.; Seatzu, C. Fault detection for discrete event systems using Petri nets with unobservable transitions. *Automatica* **2010**, *46*, 1531–1539. [CrossRef]
15. Grobelna, I.; Adamski, M. Model checking of control interpreted Petri nets. In Proceedings of the 18th International Conference Mixed Design of Integrated Circuits and Systems–MIXDES 2011, Gliwice, Poland, 16–18 June 2011; pp. 621–626.
16. Lobov, A. *Formal Validation of Discrete Automation Systems Applying Structural Reasoning and General Unary Hypothesis Automaton Methods*; Tampere University of Technology: Tampere, Finland, 2008.
17. Wu, N. Necessary and sufficient conditions for deadlock-free operation in flexible manufacturing systems using a colored Petri net model. *IEEE Trans. Syst. Man Cybern. C* **1999**, *29*, 192–204.
18. Huang, Y.S.; Jeng, M.; Xie, X.; Chung, D.H. Siphon-based deadlock prevention policy for flexible manufacturing systems. *IEEE Trans. Syst. Man Cybern. A* **2006**, *36*, 1248–1256. [CrossRef]
19. Huang, Y.-S. Design of deadlock prevention supervisors using Petri nets. *Int. J. Adv. Manuf. Technol.* **2007**, *35*, 349–362. [CrossRef]
20. Li, Z.W.; Hu, H.S.; Wang, A.R. Design of Liveness-Enforcing Supervisors for Flexible Manufacturing Systems Using Petri Nets. *IEEE Trans. Syst. Man Cybern. C* **2007**, *37*, 517–526. [CrossRef]
21. Li, Z.; Zhou, M.; Wu, N. A Survey and comparison of Petri net-based deadlock prevention policies for flexible manufacturing systems. *IEEE Trans. Syst. Man Cybern. C* **2008**, *38*, 173–188.
22. Wu, N.; Zhou, M.; Li, Z. Resource-Oriented Petri Net for Deadlock Avoidance in Flexible Assembly Systems. *IEEE Trans. Syst. Man Cybern. A* **2008**, *38*, 56–69.
23. Huang, Y.S.; Weng, Y.S.; Zhou, M. Design of traffic safety control systems for emergency vehicle preemption using timed Petri nets. *IEEE Trans. Intell. Transp. Syst.* **2015**, *16*, 2113–2120. [CrossRef]
24. Holloway, L.E.; Krogh, B.H.; Giua, A. A survey of Petri net methods for controlled discrete event systems. *Discret. Event Dyn. Syst.* **1997**, *7*, 151–190. [CrossRef]
25. Giua, A.; DiCesare, F.; Silva, M. Generalized mutual exclusion contraints 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.
26. Ma, Z.; Li, Z.; Giua, A. Design of optimal Petri net controllers for disjunctive generalized mutual exclusion constraints. *IEEE Trans. Autom. Control* **2015**, *60*, 1774–1785. [CrossRef]
27. Frey, G.; Litz, L. Formal methods in PLC programming. In Proceedings of the 2000 IEEE International Conference on Systems, Man, and Cybernetics, Nashville, TN, USA, 8–11 October 2000; pp. 2431–2436.
28. David, R.; Alla, H. *Petri Nets and Grafcet: Tools for Modelling Discrete Event Systems*; Prentice Hall: Upper Saddle River, NJ, USA, 1992.
29. *International Standard IEC 60848:2013 "GRAFCET Specification Language for Sequential Function Charts"*; The International Electrotechnical Commission: Geneva, Switzerland, 2013.
30. *International Standard IEC 61131-3:2013 "Programmable Controllers—Part 3: Programming Languages"*; The International Electrotechnical Commission: Geneva, Switzerland, 2013.
31. David, R. Grafcet: A powerful tool for specification of logic controllers. *IEEE Trans. Control Syst. Technol.* **1995**, *3*, 253–268. [CrossRef]
32. Fujino, K.; Imafuku, K.; Yuh, Y.; Hirokazu, N. Design and verification of the SFC program for sequential control. *Comput. Chem. Eng.* **2000**, *24*, 303–308. [CrossRef]
33. Li, L.; Tang, N.; Mu, X.; Shi, F. Implementation of traffic lights control based on Petri nets. In Proceedings of the 2003 IEEE International Conference on Intelligent Transportation Systems, Washington, DC, USA, 12–15 October 2003; Volume 2, pp. 1087–1090.
34. Uzam, M.; Jones, A.H. Discrete event control system design using automation Petri nets and their ladder diagram implementation. *Int. J. Adv. Manuf. Technol.* **1998**, *14*, 716–728. [CrossRef]
35. Frey, G. Automatic implementation of Petri net based control algorithms on PLC. In Proceedings of the 2000 American Control Conference. ACC (IEEE Cat. No.00CH36334), Chicago, IL, USA, 28–30 June 2000; Volume 4, pp. 2819–2823.
36. Minas, M.; Frey, G. Visual PLC-programming using signal interpreted Petri nets. In Proceedings of the 2002 American Control Conference (IEEE Cat. No.CH37301), Anchorage, AK, USA, 8–10 May 2002; Volume 6, pp. 5019–5024.

37. Klein, S.; Frey, G.; Minas, M. PLC Programming with Signal Interpreted Petri Nets. In *Applications and Theory of Petri Nets 2003*; Springer: Berlin/Heidelberg, Germany, 2003; pp. 440–449.
38. Lee, G.B.; Zandong, H.; Lee, J.S. Automatic generation of ladder diagram with control Petri net. *J. Intell. Manuf.* **2004**, *15*, 245–252. [CrossRef]
39. Peng, S.S.; Zhou, M.C. Ladder diagram and Petri-net-based discrete-event control design methods. *IEEE Trans. Syst. Man Cybern. C* **2004**, *34*, 523–531. [CrossRef]
40. Bender, D.F.; Combemale, B.; Crégut, X.; Farines, J.M.; Berthomieu, B.; Vernadat, F. Ladder Metamodeling and PLC Program Validation through Time Petri Nets. In *Model Driven Architecture—Foundations and Applications*; Schieferdecker, I., Hartman, A., Eds.; Springer: Berlin/Heidelberg, Germany, 2008; Volume 5095, pp. 121–136.
41. Korotkin, S.; Zaidner, G.; Cohen, B.; Ellenbogen, A.; Arad, M.; Cohen, Y. A Petri Net formal design methodology for discrete-event control of industrial automated systems. In Proceedings of the 2010 IEEE 26th Convention of Electrical and Electronics Engineers, Eilat, Israel, 19–21 October 2010; pp. 431–435.
42. Heiner, M.; Menzel, T. A Petri net semantics for the PLC language Instruction List. In Proceedings of IEE Workshop on Discrete Event Systems (WODES'98), Cagliari, Italy, 26–28 August 1998; pp. 161–165.
43. Da Silva, L.D.; de Assis Barbosa, L.P.; Gorgônio, K.; Perkusich, A.; Lima, A.M.N. On the automatic generation of timed automata models from function block diagrams for safety instrumented systems. In Proceedings of the 34th Annual Conference of IEEE Industrial Electronics (IECON 2008), Orlando, FL, USA, 10–13 November 2008; pp. 291–296.
44. Marsan, M.A.; Balbo, G.; Conte, G.; Donatelli, S.; Franceschinis, G. *Modelling with Generalized Stochastic Petri Nets*; John Wiley & Sons: Chichester, UK, 1994.
45. Chiola, G.; Franceschinis, G.; Gaeta, R.; Ribaudo, M. GreatSPN 1.7: Graphical editor and analyzer for timed and stochastic Petri nets. *Perform. Eval.* **1995**, *24*, 47–68. [CrossRef]
46. GreatSPN Home Page—Dipartimento di Informatica—Università di Torino. Available online: http://www.di.unito.it/~greatspn/index.html (accessed on 5 March 2017).
47. Deavours, D.D.; Clark, G.; Courtney, T.; Daly, D.; Derisavi, S.; Doyle, J.M.; Sanders, W.H.; Webster, P.G. The Mobius framework and its implementation. *IEEE Trans. Softw. Eng.* **2002**, *28*, 956–969. [CrossRef]
48. The Möbius Tool. Available online: https://www.mobius.illinois.edu/ (accessed on 5 March 2017).
49. Barbierato, E.; Dei Rossi, G.-L.; Gribaudo, M.; Iacono, M.; Marin, A. Exploiting product forms solution techniques in multiformalism modeling. *Electro. Notes Theor. Comput. Sci.* **2013**, *296*, 61–77. [CrossRef]
50. Bucci, G.; Sassoli, L.; Vicario, E. Correctness verification and performance analysis of real-time systems using stochastic preemptive time Petri nets. *IEEE Tran. Softw. Eng.* **2005**, *31*, 913–927. [CrossRef]
51. Gniewek, L. Sequential Control Algorithm in the Form of Fuzzy Interpreted Petri Net. *IEEE Trans. Syst. Man Cybern. Syst.* **2013**, *43*, 451–459. [CrossRef]
52. Gniewek, L. Coverability Graph of Fuzzy Interpreted Petri Net. *IEEE Trans. Syst. Man Cybern. Syst.* **2014**, *44*, 1272–1277. [CrossRef]
53. Markiewicz, M.; Surdej, Ł.; Gniewek, L. Transformation of a fuzzy interpreted Petri net diagram into structured text code. In *Proceedings of the 2016 21st International Conference on Methods and Models in Automation and Robotics (MMAR), 29 August–1 September 1*; IEEE: Piscataway, NJ, USA, 2016; pp. 94–99.
54. Zave, P.; Schell, W. Salient features of an executable specification language and its environment. *IEEE Trans. Softw. Eng.* **1986**, *SE-12*, 312–325. [CrossRef]
55. Harel, D. Statecharts: A visual formalism for complex systems. *Sci. Comput. Progr.* **1987**, *8*, 231–274. [CrossRef]
56. Fuchs, N.E. Specifications are (preferably) executable. *Softw. Eng. J.* **1992**, *7*, 323–334. [CrossRef]
57. Gajski, D.D.; Vahid, F.; Narayan, S. A system-design methodology: Executable-specification refinement. In Proceedings of European Design and Test Conference, 1994. EDAC, The European Conference on Design Automation. ETC European Test Conference. EUROASIC, The European Event in ASIC Design, Proceedings, Paris, France, 28 February–3 March 1994; pp. 458–463.
58. Van der Aalst, W.M.P.; de Crom, P.J.N.; Goverde, R.R.H.M.J.; van Hee, K.M.; Hofman, W.J.; Reijers, H.A.; van der Toorn, R.A. ExSpect 6.4 An Executable specification tool for hierarchical colored Petri Nets. In *SpringerLink*; Springer: Berlin/Heidelberg, Germany, 2000; pp. 455–464.
59. Lämmel, R. Google's MapReduce programming model—Revisited. *Sci. Comput. Progr.* **2008**, *70*, 1–30. [CrossRef]

60. Akhtar Khan, N.; Ahmad, F.; Khan, S.A. Formal and Executable specification of random waypoint mobility model using timed coloured Petri Nets for WMN. *Abstr. Appl. Anal.* **2014**, *2014*, e798927. [CrossRef]
61. Andreu, D.; Pascal, J.-C.; Valette, R. Fuzzy Petri net-based programmable logic controller. *IEEE Trans. Syst. Man Cybern. B* **1997**, *27*, 952–961. [CrossRef] [PubMed]
62. Gniewek, L.; Kluska, J. Hardware implementation of fuzzy Petri net as a controller. *IEEE Trans. Syst. Man Cybern. B* **2004**, *34*, 1315–1324. [CrossRef]
63. Venkateswaran, P.R.; Bhat, J. Fuzzy Petri net algorithm for flexible manufacturing systems. *ACSE J.* **2006**, *6*, 1–5.