Simulation of Discrete-Event Systems in MATLAB

Raul Campos-Rodriguez,
Mildreth Alcaraz-Mejia and Uriel Sanchez-Ramirez

Additional information is available at the end of the chapter

http://dx.doi.org/10.5772/63230

Abstract

The discrete-event systems (DES) are systems guided by asynchronous events rather than by the passage of the time as in traditional systems. There exists a wide set of systems that could be considered within this class, such as communication protocols, computer and microcontroller operating systems, flexible manufacturing systems, communication drivers for embedded applications and logistic systems, among others. Their proper study is a requirement for a suitable implementation of embedded hardware and software, for example. The aim of this chapter is to approach the simulation of this class of systems within the MATLAB/SIMULINK framework. A suitable simulation process, allowing the injection of input signals to the system and observing its output response, is a first step in the analysis of this class of systems, which may lead to more elaborated analysis such as reachability and deadlock avoidance. The advantage of the approach and techniques proposed in this chapter is the application of the set of tools, algorithms and visualization instruments present in the MATLAB/SIMULINK to the simulation of Discrete-Event Systems, which allows a simple integration of various DES by utilizing the matrices that define them. The concluding section of the chapter provides a link for downloading all the code for the examples developed here.

Keywords: discrete-event systems, analysis, modelling, simulation, MATLAB/SIMULINK

1. Introduction

The discrete-event systems (DES) are systems guided by asynchronous events rather than by the passage of time as in the traditional framework of Control Theory, for example [1]. There exists a wide set of systems that could be considered in the class of DES, such as operating systems of microprocessors and embedded microcontrollers, communication protocols such as IPv4/IPv6, complex software architectures such as database management systems, production systems and flexible manufacturing systems (FMSs), delivering and logistic systems, among others. Their proper study is a requirement for the fulfillment of performance and safety requirements, for example. The traceability of requirements and its satisfaction is simplified by using a model that is suitable for a rigorous simulation process [2].

The aim of this chapter is to approach the simulation of DES within the MATLAB/SIMULINK framework. Analysis such as the application of random inputs to a DES and the visualization of system's output response are intended to be covered in this chapter. The overall goal is to enable the application of the set of tools, algorithms and visualization instruments present in the MATLAB/SIMULINK to the analysis of DES. There exist several approaches for the analysis of this class of systems. On the one hand, for example, empirical practices are used for addressing the problems that arise in the DES field. Most of these practices are based on experience and good knowledge among engineers in the daily execution of a system. On the other hand, in the formal point of view, scientists and engineers typically use mathematical tools based on automata theory, Petri nets (PN), Markov chains and Queue theory for addressing main aspects in the design and implementation of DES. The aspects most studied in the analysis of DES are the reachability and deadlock analysis, fault tolerance, control and observability schemes, to mention a few [3].

In recent years, the simulation methods have taken great relevance in the design and implementation of big systems. These methods allow engineers and scientists the study of complex behaviours by simulating in the lab different real-world scenarios. Intensive workload conditions, parametric variations, environmental changes and fault scenarios are possible to investigate by simulation methods. Statistical information, performance curves, and parameter optimization are some of the possible results obtained by a simulation process.

2. Discrete-event systems (DES)

As mentioned in the introduction, within a DES the state evolution depends on the occurrence of events that are asynchronous in time. An event is an instantaneous action occurred in the context of the DES that is relevant for the understanding of the system. An occurrence of an event may cause an immediate change in the system state. For example, an event could be a package arriving by the network connection, a button pressed by the user at a control panel, a timer's overflow within an embedded device driver, a change in a Boolean flag within an Interrupt Service Routine, etc. By convention, it is supposed that no time is elapsed between the event occurrence and the change of the state in a DES.

Some examples of DES's include communication protocols, supply chains, queue systems, task schedulers, logistic systems, device drivers, memory managers, landing and take-off systems of airplanes, urban rail systems and subway, and line of manufacturing and production systems, among others. For a wide list of examples of DES, see [4].

The study of a DES is important for several reasons, including safety and economic issues, for example. There exist several approaches in the study of this class of systems. For example, there exist empirical practices for addressing the problems that arise in the DES. Most of these are based on experience and good knowledge among engineers in the daily execution of the DES. In the formal point of view, scientists and engineers use automata theory, PN, Markov chains and Queue theory for addressing main aspects in the design and implementation of DES, such as the modelling, reachability and deadlock analysis, fault tolerance, and control schemes, among other interesting properties [5, 6].

In recent years, the simulation methods have taken great relevance in the design and implementation of big systems. These methods allow engineers and scientists the study of complex behaviours by simulating in the lab different real-world scenarios. Intensive workload conditions, parametric variations, environmental changes and fault scenarios are possible to investigate by simulation methods. Statistical information, performance curves, and parameter optimization are some of the possible results obtained by a simulation process.

2.1. Modelling DES with finite state machines

The finite state machines (FSM) are one of the first and most used mathematical models for the representation of the dynamics of a DES. A FSM is an extension of the concept of the automaton [7]. The states and events are basic concepts in the construction of a FSM. It is supposed that at every time, the FSM is in one of a finite number of states and that an incoming event causes an immediate change in the state of the FSM. Formally, a FSM is defined by $G = (Q, \Sigma, \delta, q_0)$ where [8]:

- Q is a finite set of states,
- Σ is a finite set of input symbols called events,
- $\delta : Q \times \Sigma \rightarrow Q$ is a partial relation called the state-transition function,
- q_0 is the initial state and is in Q.

As a graphical representation, the states are depicted as circles or ovals, while the events are represented as labelled arrows from one "source" state to other "destination" state. The initial state q_0 is designated by an incoming arrow, usually thicker than the other, with no source state.

Alternatively, the definition of a FSM may include a set of "marked states" designated as Q_m which represents the "acceptable" or "suitable" states of a DES. Moreover, an extension to the state-transition function may include subsets of Q as its range, allowing the representation of a non-deterministic FSM. For simplicity of the code implemented in this work, the deterministic definition of a FSM with no marked states is considered. The modification of the code here developed for the inclusion of those cases is not hard to achieve.

Figure 1 depicts a FSM model of the basic functionality of a typical microwave oven adapted from [9]. The initial state is Idle, as denoted by the thicker incoming arrow to s_1, where the oven performs no activity, and it is waiting for the buttons pressed by the user. The events that the user may execute in the system are denoted by the labels over the arrows. For example, at the "Idle" state the user may press the "Full Power" button (e_1) that causes a change to the state s_2 "Full Power on."

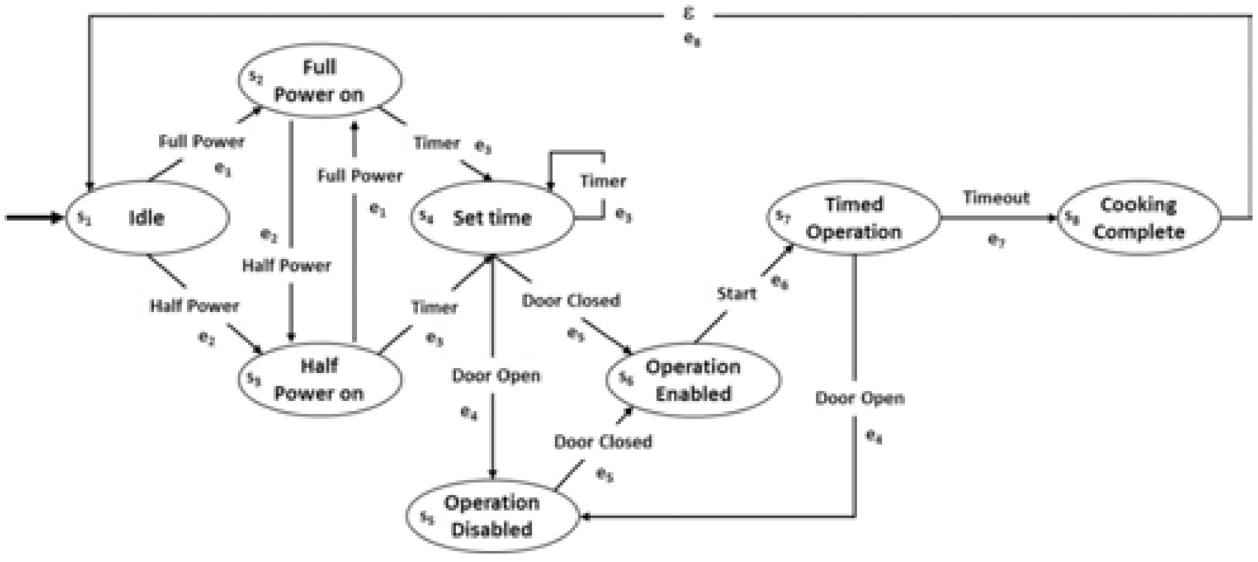


Figure 1. A FSM model of a microwave oven. The initial state is determined by the bold incoming arrow to s_1 which corresponds to Idle. The system events are labelled over the arrows and designated as e_i for $i = 1 \ldots 8$ for an easy implementation. Similarly, the states are designated as s_j for $j = 1 \ldots 8$.

Then the user may set the time for the cooking process at the "Set Time" state. For security reasons, if the door is opened, the operation of the oven is disabled, otherwise it is enabled. At the "Operation Enabled" state, i.e. s_6, if the user presses the "Start" button, the cooking process begins. Any opening of the oven's door immediately disables its operation. After a timeout event, the cooking process is completed and the FSM is restarted to its initial state ready for the next operations.

For simplicity in the construction of the state-event matrix, the labels s_i for $i = 1, \ldots, 8$ and e_j for $j = 1, \ldots, 8$, have been added to the FSM model representing the state and event, respectively. The initial state of the FSM in **Figure 1** is $q_0 = e_1$ while the state-event matrix is the following:

$$\begin{array}{c} \\ s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \\ s_6 \\ s_7 \\ s_8 \end{array} \begin{array}{c} \begin{array}{cccccccc} e_1 & e_2 & e_3 & e_4 & e_5 & e_6 & e_7 & e_8 \end{array} \\ \begin{bmatrix} s_2 & s_3 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & s_3 & s_4 & 0 & 0 & 0 & 0 & 0 \\ s_2 & 0 & s_4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & s_4 & s_5 & s_6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & s_6 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & s_7 & 0 & 0 \\ 0 & 0 & 0 & s_5 & 0 & 0 & s_8 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & s_1 \end{bmatrix} \end{array}$$

By using the information of the initial state and the state-event matrix, the next section provides a way of simulating an arbitrary FSM by the implementation of an S-Function in MATLAB. Before addressing the details of the implementation, the following subsection considers another useful method based on PN for the modelling of a DES.

2.2. Modelling DES with PN

A PN is another mathematical tool widely used for the design, the modelling, and the simulation of a DES [10]. The modelling process of a DES with a PN is quite natural and intuitive, due to its graphical representation as in the case of FSM. One advantage of the PN modelling technique is the compact representation of the systems. Moreover, the PN formalism resides in a strong mathematical basis from the linear algebra. Formally, a PN model is a pair (B, M_0), where B is a Petri net structure (PNS) $\{P, T, F\}$, such that:

- $P = \{p_1, p_2, \cdots, p_m\}$ is a finite set of places;
- $T = \{t_1, t_2, \cdots, t_n\}$ is a finite set of transitions;
- $F = I \cup O$ is a flow relation, where $I(p_i, t_j) \to \mathbb{N}^+$ and $O(t_i, p_i) \to \mathbb{N}^+$ are the input and output functions;
- $M_0 \in (\mathbb{N}^+)^m$ is a special vector known as the initial marking of the net, where $m = |P|$.

Pictorially, circles represent the places, while rectangles or bars, represent the transitions. The flow relation $F = I \cup O$, is represented as directed arcs or arrows. On the other hand, the matrices $B^-(i, j) := I(p_i, t_j)$ and $B^+(i, j) := O(t_i, p_i)$, capture the structure of the flow function, while $B := B^+ - B^-$, represents the incidence matrix of the PN. Thus, B is a matrix of size $[m \times n]$, where m is the number of places, and n the number of transitions. The net's state, or marking, is a vector $M(k) \in (\mathbb{N}^+)^m$, where m is the number of places in the PN. A marking represents the state of the net at time k, i.e., the number of tokens in each place at the time k.

In the classical definition of a PN, the number of tokens cannot be negative. Also, it is supposed that the index k is updated at every time that an event occurs. For simplicity, the marking $M(k)$ is represented by using subscripts as M_k, and $M_k(p_i)$, $p_i \in P$ for representing the number of tokens in place p_i at the time k. The *initial marking* M_0 represents an initial tokens distribution over the net's places. Thus, $M_0(p_i)$ for $p_i \in P$, represents the initial number of tokens in place p_i. The marking M_0 may enable one or more transitions to be fired. An *enabled* transition $t_i \in T$ at the marking M_0, denoted by $[M_0\rangle_t$, is one that fulfils $M_0(p_j) \geq B^-(p_j, t_i)$, $\forall\, p_j \in P$. Given any marking, say M_k, the set of all its enabled transitions is simply denoted as $[M_k\rangle$. The firing of enabled transitions leads to the dynamic behaviour of a PN, captured by the state equation:

$$M_{k+1} = M_k + B\vec{u}_k$$

The interpretation of the previous equation is as follows. The marking M_k, of size $[m \times 1]$, represents the system state at time k. The vector $\vec{u}_k$, of size $[n \times 1]$, represents the firing of one or more enabled transitions by the marking M_k. The matrix B, of size $[m \times n]$, is the incidence matrix of the net. The vector M_{k+1} represents the state reached by net's evolution. If $[M_k\rangle t_i$ and t_i is fired, then using (1), the net reaches a new marking computed as $M_{k+1} = M_k + B\vec{t}_i$. In this equation, $\vec{u}_k = \vec{t}_i$ is a vector with a one in the $i-th$ position and zero anywhere else. This marking's evolution is denoted as $M_k \xrightarrow{t_i} M_{k+1}$, in order to emphasize the fact that from marking M_k the net fires t_i and reaches M_{k+1}. The marking evolutions may consecutively enable other transitions to be fired, which leads to the concept of reachability set of a PN. The reachability set of the PN (B, M_0) is denoted by $R(B, M_0)$, for emphasizing the fact that it depends on initial condition M_0. Thus, $R(B, M_0)$ is the set of all markings M_k evaluated by (1), by only considering the firing of enabled transitions. A firing transition sequence of the PN (B, M_0) is a sequence of transitions $\sigma = t_i t_j t_k \cdots t_l$ such that $M_0 \xrightarrow{t_i} M_1 \xrightarrow{t_j} M_2 \xrightarrow{t_k} \cdots M_l \xrightarrow{t_l} M_s$, where the length of σ, denoted by $|\sigma|$, is the number of its transitions. If the number of transitions in σ is not finite, then $|\sigma| = \infty$. A short representation for this trajectory is $M_0 \xrightarrow{\sigma} M_s$, for highlighting the fact that from M_0, the net fires σ, and reaches M_s. If $M_k \xrightarrow{\sigma} M_s$ for some M_k and σ, then $[M_k \sigma$ means that the marking M_k, enables the firing of the entire sequence σ.

The Parikh Vector $\vec{\sigma} \in \mathbb{N}^m$ maps every transition in the set T to its number of occurrences in sequence σ. Thus, if $\sigma = t_i t_j t_i$ then, $\vec{\sigma}$ is a $n-vector$ with a two in the $i-th$ position, one on the $j-th$ position, and zero anywhere else. The firing language of an PN (B, M_0) is

$\mathcal{L}(B, M_0) := \{\sigma \in T^* \mid \sigma = t_i t_j t_k \dots t_l\}$, such that $M_0 \xrightarrow{t_i} M_1 \xrightarrow{t_j} M_2 \xrightarrow{t_k} \cdots M_r \xrightarrow{t_l} M_s$, where T^* is the Kleen closure, as in automata theory [7], of the transition's set.

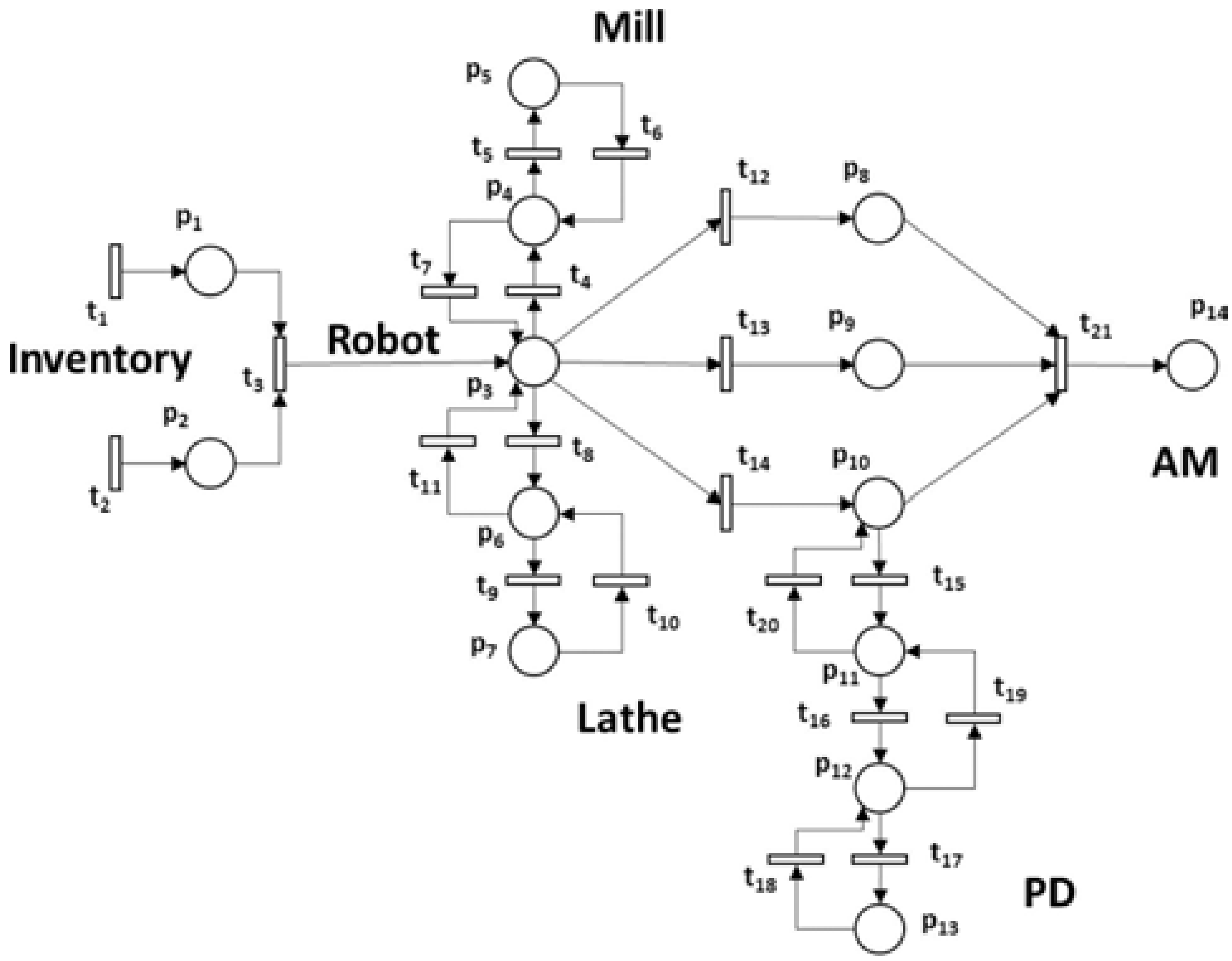


Figure 2. PN model representing a FMS. The system is composed of a mill machine, a lathe and a robot, connected through buffers. The incoming raw material arrives from the inventory to the robot's section. Then they are routed to the mill, lathe, painting or assembling stations.

Figure 2 represents a PN model of a FMS adapted from [11]. Be careful to not confuse the name of a FMS in this subsection with the name of a FSM of the previous one. It is composed of a robot arm (p_3), a mill machine (p_5), a lathe (p_7), a painting device (p_{13}) and an assembling machine (p_{14}). A set of buffers $\{p_1, p_2, p_4, p_6, p_8, p_9, p_{10}, p_{12}\}$ connects together the system. The FMS is able to process different components at the intermediate stages, which are lastly assembled at the AM stage (p_{14}). The firing of t_1 and t_2 represents the arriving of raw material from a non-depleting inventory, which is aleatory. It was interpreted and adapted from its original layout in [11]. The incidence matrix of the PN is of size $B[14 \times 21]$, while the initial marking $M_0[14]$ is a zero vector, meaning that the FSM is empty. The incidence matrix B of the FMS is the following:

$$
\begin{bmatrix}
1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & -1 & 0 & 0 & 1 & -1 & 0 & 0 & 1 & -1 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & -1 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 1 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 1 & -1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{bmatrix}
$$

The next section is devoted to the implementation of suitable functions for the integration of FSM models as well as PN models into a Simulink model.

3. Simulating DES in MATLAB

The S-functions are a mechanism in the MATLAB environment for extending the capabilities of SIMULINK. By writing an S-function, the user is able to implement complex behaviours of real systems that directly interact with other blocks in a SIMULINK model. The S-function API defines the structure of an S-function and accepts MATLAB, C, C++ or FORTRAN as coding languages. The S-function is compiled as MEX files, which are linked, loaded and executed dynamically. By using a special syntax, it is possible to write functions for continuous, discrete and hybrid systems. Moreover, with a proper solver and a suitable integration step, the modelling and simulation of a DES is possible with accuracy and great detail. The SIMULINK library provides a block called S-Function as a placeholder for the user defined S-functions. The block includes a dialog box where it is possible to specify the file containing the function and its parameters.

There are two different categories for the S-functions, called Level-1 and Level-2. Each of them has its advantages and disadvantages. A detailed comparison among them is out of the scope of this chapter. An interested reader may refer to the MATLAB documentation for more information. The Level-2 is the newest category and is the one used in the rest of this work. The functions are written in MATLAB as m-files.

The name of the m-file has to be that of the function it implements and has to include five standard sections. The *setup* section is executed in the initialization stage of the simulation process and allows specifying the number of inputs, outputs, states and parameters, among other characteristics of the function. The *DoPostPropSetup* is executed after the setup and allows defining the blocks of memory used by the function. It also allows the definition of discrete state variables. The *InitConditions* section specifies the initial conditions of the block, while the *output* section specifies the outputs it produces in the sense of the control theory. The *Update* is perhaps the most important section since it is the place where the system's dynamic is implemented. The Update and Output section are executed at every integration step in the simulation process.

3.1. Implementing the FSM dynamics

The state-event matrix of a FSM captures its behaviour and allows a straightforward implementation of its dynamics within a SIMULINK model. Following the structure of an S-function, the next code provides an overview of the main aspects of its implementation. The function is called my_fsm and corresponds with the file name.

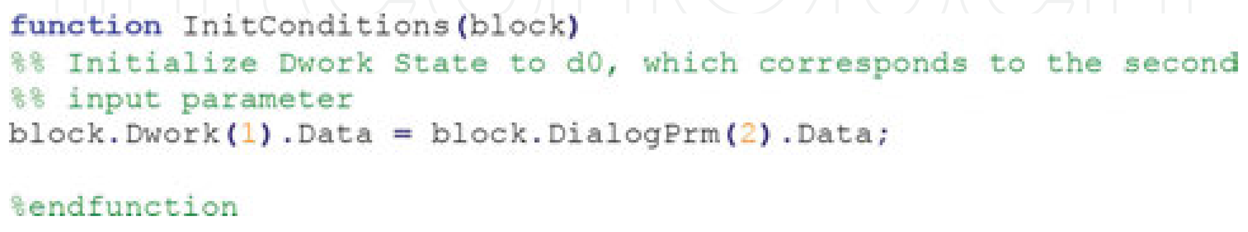
```
function my_fsm(block)
% Level-2 M file S-Function for the implementation of a FSM
% dynamics. inherited sample time demo.
%   $Parameter:
%            -State-Event Matrix D
%            -Initial State d0
%   $Revision: 0.1 $

  setup(block);

%endfunction
```

The setup defines the number of input parameters and its characteristics, among others.

```
function my_fsm(block)
…
function setup(block)
%% Set the number of input parameters
block.NumDialogPrms  =       2;
%% Register the number of input and output ports
block.NumInputPorts  =       1;
block.NumOutputPorts =       1;
        …
%% Set the properties of the input port. The vector dimension is one
%% since it is equal to an input symbol in the FSM alphabet
block.InputPort(1).Dimensions        =       1;
block.InputPort(1).DirectFeedthrough = false;
%% Set the properties of the output port. The vector dimension is
%% one since it is equal to the current state of the FSM
block.OutputPort(1).Dimensions       =       1;
…
%endfunction setup
```

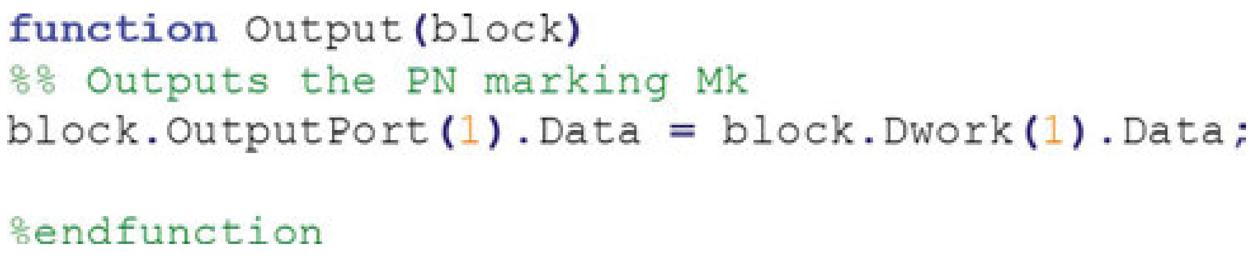
The DoPostPropSetup allows defining the state of the FSM that has to be preserved during the simulation process. Since this implementation is for a deterministic FSM, it is one dimensional and used as discrete state. The name is chosen conveniently to be "State."

The InitConditions allows defining which of the states of the FSM is selected as initial. The initial state is provided by the user as the second parameter of the block.



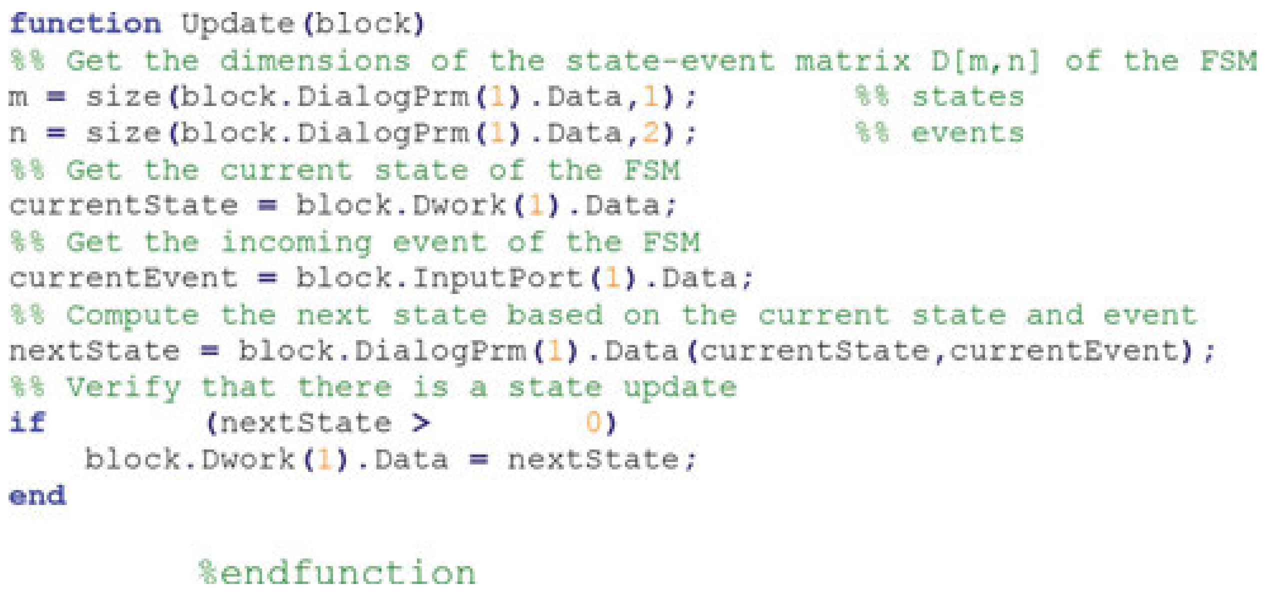
```
function InitConditions(block)
%% Initialize Dwork State to d0, which corresponds to the second
%% input parameter
block.Dwork(1).Data = block.DialogPrm(2).Data;

%endfunction
```

The Output function returns the current state of the FSM, which is stored in the 24 DWork vector previously defined in the DoPostPropSetup section.



```
function Output(block)
%% Outputs the PN marking Mk
block.OutputPort(1).Data = block.Dwork(1).Data;

%endfunction
```

Finally, the Update section implements the change of state experiments by a FSM due to the input signals it receives. For this purpose, the state-event matrix is represented with the rows labelled as the states of the FSM while the columns are labelled as its events. To simplify the codification within a MATLAB function, only integers are used for representing both the state as well as the events. In this way $D(1, 2)=3$ means that when the FSM is in the state one, i.e. s_1, and the input is the event two, i.e. e_2, then the FSM reaches the state three, i.e. s_3. When $D(i,\ j)=0$ means that at the $i-th$ state, the $j-th$ event is not defined, and accordingly, the state of the FSM is not changed. By assuming this convention in the codification of the FSM, the next code is straightforward and implements the change of state in a FSM due to the incoming events.



```
function Update(block)
%% Get the dimensions of the state-event matrix D[m,n] of the FSM
m = size(block.DialogPrm(1).Data,1);          %% states
n = size(block.DialogPrm(1).Data,2);          %% events
%% Get the current state of the FSM
currentState = block.Dwork(1).Data;
%% Get the incoming event of the FSM
currentEvent = block.InputPort(1).Data;
%% Compute the next state based on the current state and event
nextState = block.DialogPrm(1).Data(currentState,currentEvent);
%% Verify that there is a state update
if         (nextState >         0)
    block.Dwork(1).Data = nextState;
end

          %endfunction
```

3.2. FSM simulation example

The function my_fsm detailed in the previous subsection is used for the simulation of the FSM of **Figure 1**. The state-event matrix is coded as a matrix $D[8\times8]$ of integer where every entry represents a state, as discussed in the previous subsection. Thus, for example, $D(1, 1)=2$, as expected according to the FSM in **Figure 1**. The ε event from "Cooking Complete" to "Idle", i.e. from s_8 to s_1, is coded as e_8. That is, the event e_8 corresponds to the last column in the state-event matrix. The entire matrix coded in the MATLAB workspace is:

$$D=\begin{bmatrix}2&3&0&0&0&0&0&0\\0&3&4&0&0&0&0&0\\2&0&4&0&0&0&0&0\\0&0&4&5&6&0&0&0\\0&0&0&0&6&0&0&0\\0&0&0&0&0&7&0&0\\0&0&0&5&0&0&8&0\\0&0&0&0&0&0&0&1\end{bmatrix}$$

By using the above matrix D, **Figure 3** shows a SIMULINK model that integrates the my_fsm for simulating the FSM model of the microwave oven in the **Figure 1**. The FSM block encapsulates a Level-2 S-function block with the my_fsm S-function inside. The parameters are the matrix D and the initial state $d0=1$ defined in the MATLAB workspace in the same directory of the SIMULINK model. A random number generator, together with constant of one, represents the aleatory generation of events for the FSM block.

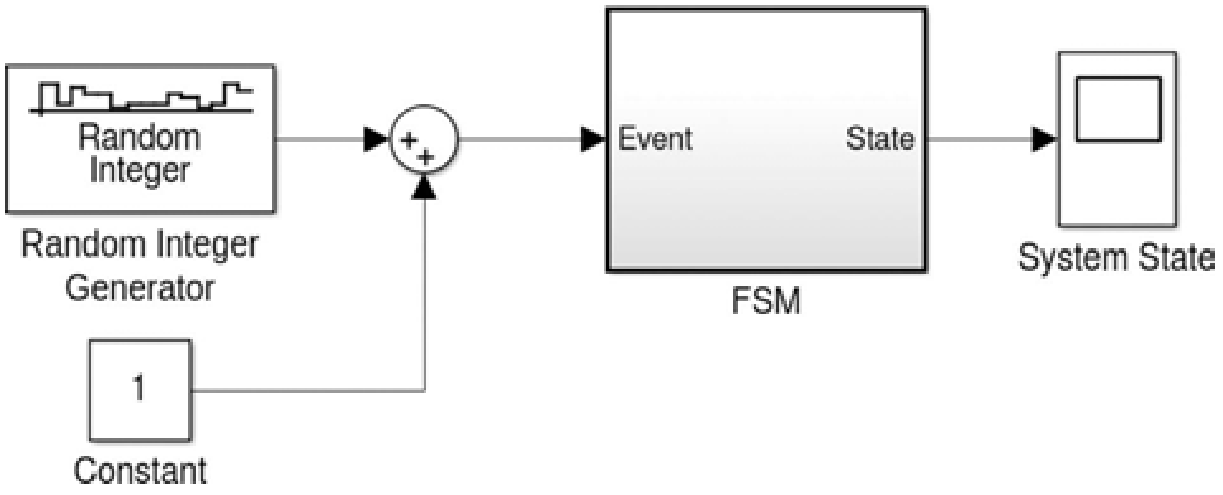


Figure 3. A SIMULINK model that uses the my_fsm function representing the microwave oven. The random integer generator produces a number in the range $[0, k-1]$ which represents the incoming events to the FSM model. The constant is added to avoid the generation of the zero event.

The constant is added to avoid the generation of the zero event which is meaningless in the context of the simulation process of the microwave oven. The FSM block clearly defines the input and output ports for the incoming events and system state, respectively. The scope allows the visualization of the events reached by the FSM as the process simulation evolves.

Figure 4 shows a simulation of 200 events of the microwave oven model in **Figure 3**. At the time zero, the FSM is in the state s_1, as defined by $d0=1$. Then, the system changes to the state s_3, which means that the incoming event was e_2, i.e. the user pressed the "Half Power" button in the oven panel. At that time, the simulation process shows that the user pressed the "Full Power" since the system state has down changed to s_2. Then, the system remains at the same state s_2 for some time. After that, the system state changes to s_3 again and it immediately moves to the state s_5 by briefly passing to the state s_4 before.

The system state remains at s_5 for a while and then it moves to the state s_6, then to s_7 and then back to s_5 again (the user opened the oven's door!).

Then, it seems that the user closed the door (e_5) and pressed the "Start" button (e_6). Thus, the cooking process ended above the event fifteen and the system state returns to its initial idle condition at s_1. Other interesting behaviours of the systems could be analysed from the chart. For example, it could be noticed that in a second execution, over the event 18, the oven was completing a more direct cooking process where the oven's door has not opened once the cooking process started. Above the event nineteen, the system returns to the idle state s_1. This could be considered a typical cooking process for a microwave oven.

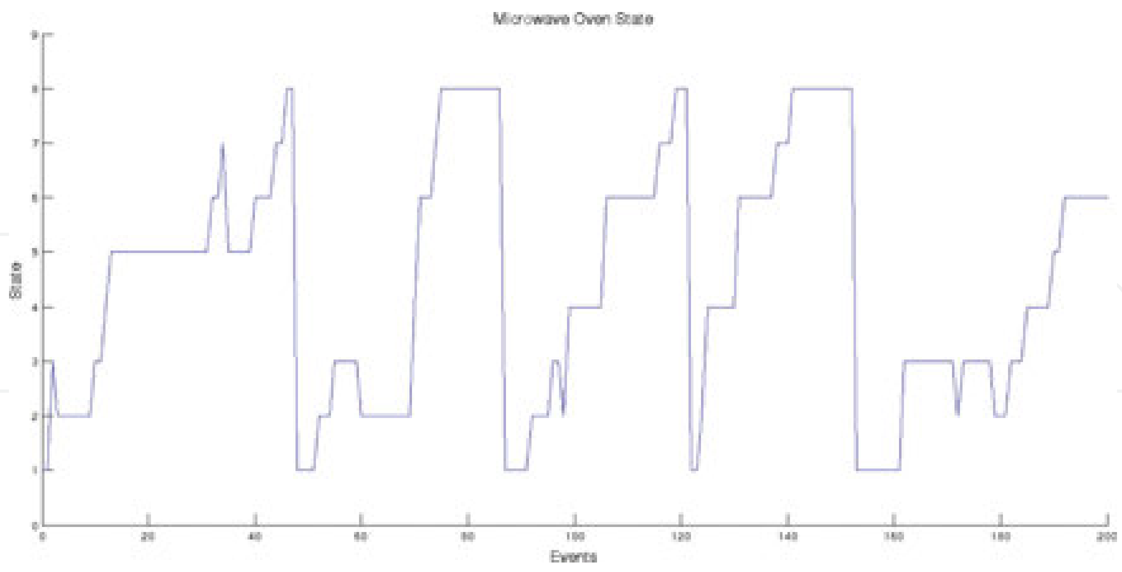


Figure 4. The different states of the FSM representing the microwave oven for a simulation process of 200 events. The chart shows four complete cocking process, represented by those reaching the state eight. The difference in those completed processes represents different action carried out by the user.

In a similar way, the chart in the **Figure 4** and others that may be obtained from different simulation processes of the model in **Figure 3** could be interpreted, allowing optimizations of the FSM behaviour to meet security and performance requirements.

3.3. Implementing the PN dynamics

The dynamic behaviour of a PN is governed by its state equation. The enabling condition for the transitions in a PN is an additional requirement for the trajectory evolutions in a model. The setup stage allows defining the size of the net model.

The implementation considers two parameters. The first one is the incidence matrix B and the second one is the initial marking M_0. These parameters are used for defining the sizes of the input and output ports of the block.

```
function setup(block)
    ...
%% Set the properties of the input port. The vector dimension is
%% equal to the number of transitions in the PN, i.e., the size of
%% the second dimension of the first input parameter.
block.InputPort(1).Dimensions = size(block.DialogPrm(1).Data,2);
block.InputPort(1).DirectFeedthrough = false;
%% Set the properties of the output port. The vector dimension is
%% equal to the number of places in the PN, i.e., the size of the
%% first dimension of the first input parameter.
block.OutputPort(1).Dimensions = size(block.DialogPrm(1).Data,1);
...

%endfunction setup
```

The DoPostPropSetup stage allows the definition of the state of the PN. The PN marking has to be preserved between simulation steps and also is the output of the block.

```
function DoPostPropSetup(block)
%% Setup Dwork vectors
block.NumDworks                        =        1;
%% The vector required to be stored corresponds to the PN marking Mk
block.Dwork(1).Name                    =        'Mk';
        %% The size corresponds to the number of places of the PN
block.Dwork(1).Dimensions              = size(block.DialogPrm(1).Data,1);
...

%endfunction
```

The InitConditions stage initializes the marking M_k, defined in the previous stage, to the second input parameter, which corresponds to the initial marking M_0.

```
function InitConditions(block)
%% Initialize Dwork Mk to M0, which corresponds to the second input
%% parameter
block.Dwork(1).Data = block.DialogPrm(2).Data;
```

The Output stage is used to provide to external blocks the current marking of the PN, which corresponds to the DWork vector defined in the DoPostPropSetup stage.

```
function Output(block)
%% Outputs the PN marking Mk
block.OutputPort(1).Data = block.Dwork(1).Data;

%endfunction
```

The Update stage is where the PN dynamics is implemented. This stage verifies the transitions that are allowed to fire by the input provided by an external agent, or controller, in the sense of the Control Theory. Likewise, it verifies that those allowed transitions are also enabled by the current marking M_k of the PN. A random number generator allows an aleatory firing among the transitions that are ready to fire. Finally, the current marking M_k is updated in accordance to the PN state equation.

```
function Output(block)
%% Outputs the PN marking Mk
block.OutputPort(1).Data = block.Dwork(1).Data;

%endfunction
function Update(block)
%% Obtain the dimensions of the incidence matrix B[m,n]
m = size(block.DialogPrm(1).Data,1);
n = size(block.DialogPrm(1).Data,2);
j=0;
%% Compute the enabled transitions that are allowed to fire
t_en=[];
for i=1:n
%% Check whether the i-th transition has been allowed to fire
   if         (block.InputPort(1).Data(i,1))
   %% This is the test of the enabling condition
   if         ( block.Dwork(1).Data>=(-block.DialogPrm(1).Data(:,i)))
      %% This transition is enabled. Add it to t_en
      t_en=[t_en;i];
        end
end

end
%% Fire an arbitrary transition among those that are enabled at the
%% current marking Mk and that have been allowed to fire. As well,
&& compute the marking change dMk.
        if        (size(t_en))
              rand('twister', sum(100*clock));
      r = size(t_en,1);
      j = fix(r*rand +        1);
              i = t_en(j,1);
      dMk = block.DialogPrm(1).Data(:,i);
        else
      dMk =         0;
        end
%% Update the current marking Mk
block.Dwork(1).Data = block.Dwork(1).Data + dMk;

%endfunction
```

3.4. PN simulation example

The S-function my_ptn is used for the simulation of the PN model in **Figure 2**. The SIMULINK model is depicted in **Figure 5a**. The only elements required for the simulation process are the incidence matrix $B[14\times21]$ and the initial marking $M_0[14]$.

The model includes a block of 21 elements to represent that all of the transition of the model are allowed to fire. In this way, the dynamics of the PN model entirely depends on the marking of the net. The scope allows the visualization of the marking of all the places of the PN.

With a discrete solver and a fixed step of one, this model allows the simulation of the FMS. As shown in **Figure 5b**, the subsystem for the PN model includes blocks for inputs and outputs. These blocks could be used in the modelling of several controllability and observability problems by using matrices of proper size. For example, a matrix for the input function block may be arranged with columns representing the transitions of the PN and the rows representing the input commands to the system.

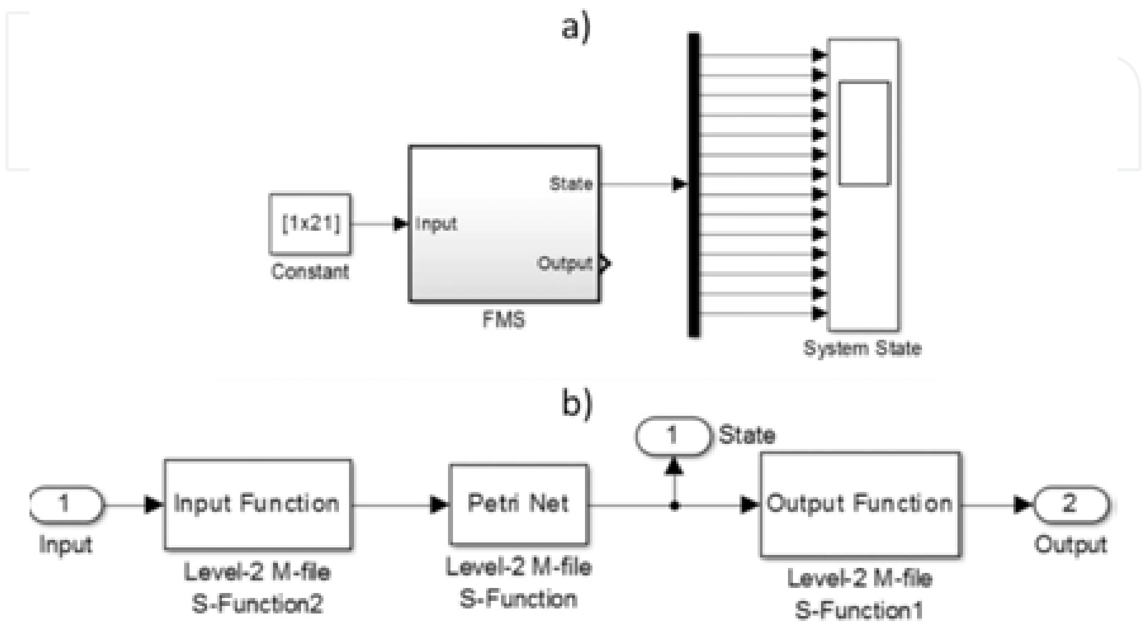


Figure 5. The integration of the S-Function implementing a PN model into the SIMULINK environment. In (a) a model for the FMS is depicted, with a constant input allowing all the 21 transitions to fire and a scope for signal visualization. In (b) an insight of the FSM block is shown, which includes blocks for inputs and outputs.

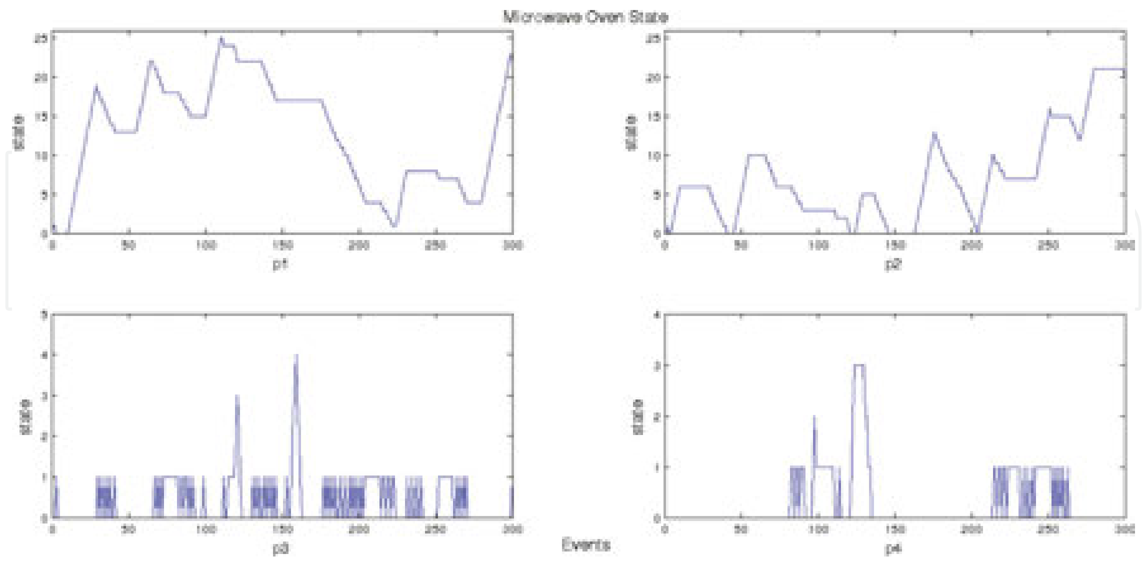


Figure 6. First part of the state of the PN representing the Flexible Manufacturing System. The chart shows the marking evolution of the places from p_1 to p_4 for a simulation process of 300 events. The first two places shows the pattern of the arriving material from the inventory to the system.

Similarly, a matrix for the output function block may be arranged with columns representing the places of the PN and the rows representing the output signals from the system. However, a deep study of these topics are out of the scope of this work, and are here mentioned for providing a more complete simulation model that could be used for more purposes. Thus, for both cases in this simulation, the core function for the input and output blocks are identity matrices.

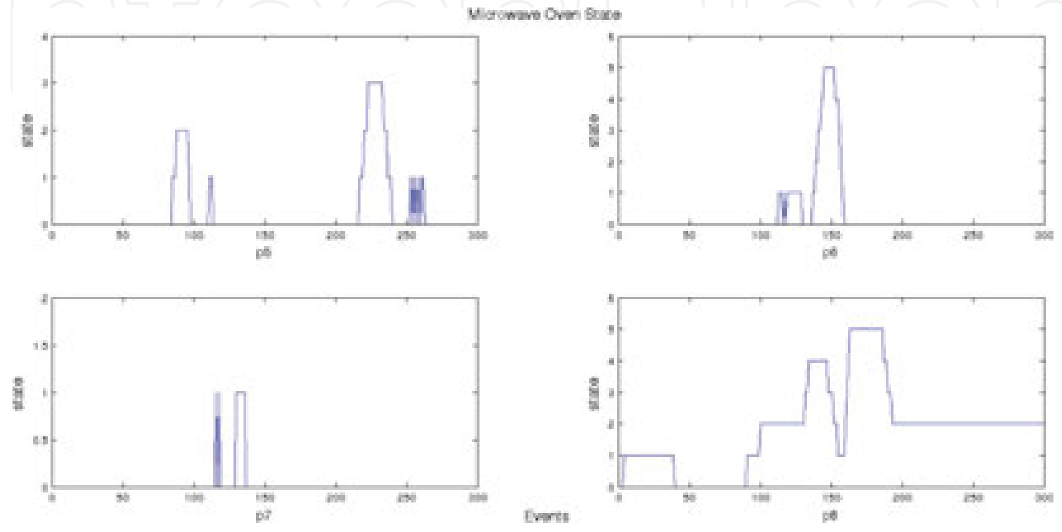


Figure 7. Second part of the state of the PN representing the FMS. The chart shows the marking evolution of the places p_5 to p_8. The place p_5 corresponds to the mill machine, while the places p_6 and p_7 correspond to the lather. The place p_8 shows the pieces waiting the AM machine.

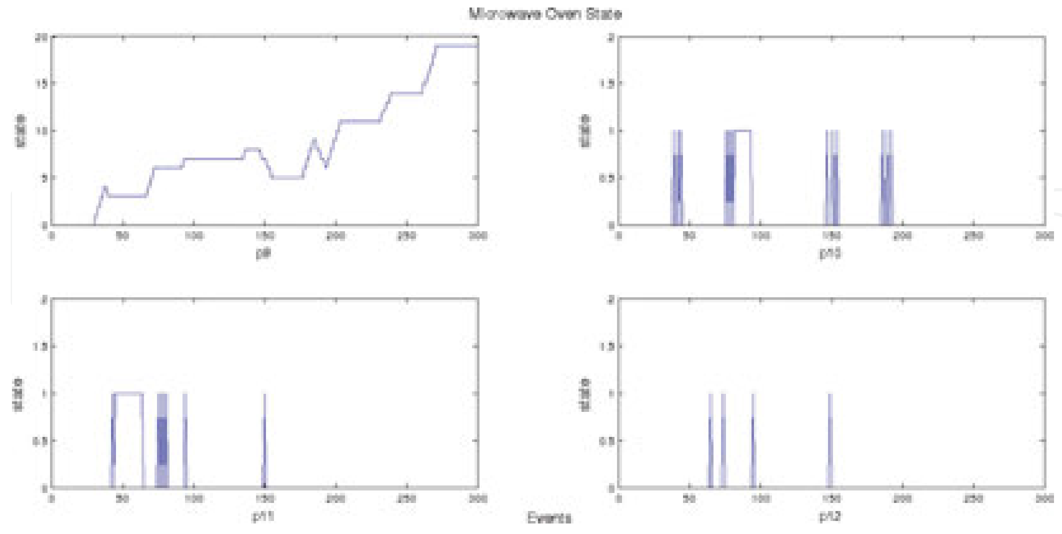


Figure 8. Third part of the state of the PN representing the FMS. The chart shows the marking evolution of the places from p_9 to p_{12}. The place p_9 corresponds to a waiting stage for the pieces prior to its assembling in AM. The places p_{10}, p_{11} and p_{12} correspond to the painting stage.

Figure 6 shows the marking of all the places in the PN model for a simulation process of 1000 events (seconds). Since the integration step was fixed to one, then every second in the scope could be interpreted as an event in the DES. The aleatory behaviour of the signal in the scope is due to the random selection of the transition firings in the Update section of the S-function, as detailed in the last subsection. It is easy noting an accumulation of tokens, or parts, in the place p_9 as well as in the place p_{14}. On the one hand, the accumulation of tokens in place p_9 means that the event associated to transition t_{13} is firing at a rate greater that of t_{12} and t_{14}. On the other hand, the accumulation of tokens in the place p_{14} is normal since there is where the finished products are stored (**Figures 7–9**).

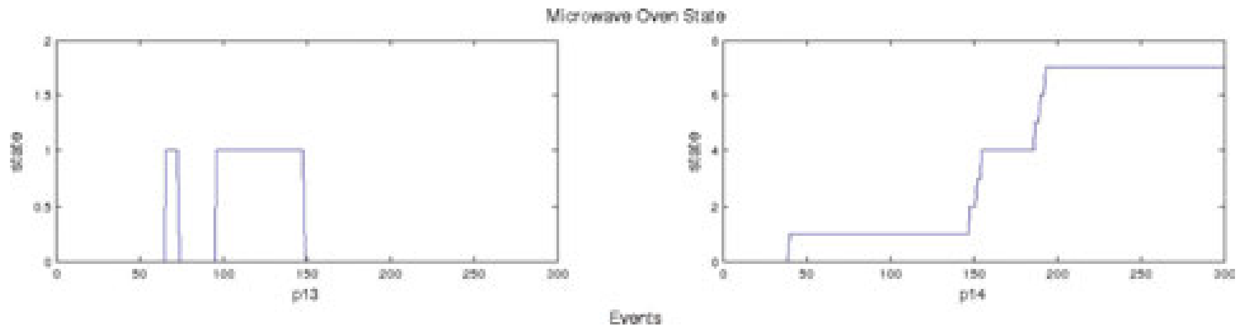


Figure 9. Last part of the state of the PN representing the FMS. The chart shows the marking evolution of the places p_{13} and p_{14}. The place p_{13} corresponds to the last section of the painting stage while the place p_{14} represents a buffer of pieces finished in the FMS.

Indeed, it is to be expected that the number of tokens in the place p_{14} increases over time. A good exercise is to modify the PN model including an extra transition with p_{14} as its unique input place, run the simulation and analyze the effects in the marking of this place. Such an extra transition may represent the interconnection of this system to another section in a more complex assembling line.

The markings of the places p_1 and p_2 represents an increase in the number of raw parts arriving to the FSM. The behaviour of the marking in the other places follows an aleatory pattern due to the random number generator used in the selection of the firing transition inside the Update section in the S-function.

4. Conclusions

This chapter showed a suitable way of simulating Discrete-Event Systems within a SIMULINK model in the MATLAB framework. The dynamics of a FSM as well as a PN has been implemented by using Level-2 MATLAB S-function. One of the advantages of the technique developed in this work is that for simulating a system, only the matrices that define a DES are required.

By using a discrete solver with a fixed step of one, accurate simulation processes in a SIMULINK model are possible. Two application examples illustrate the developed techniques. On the one hand, a FSM model representing a microwave oven has been simulated. On the other hand, a PN model representing a FMS has been simulated, as well.

Extension for FSM including marked states and non-determinism are simple to implement based in the code here provided. Similarly, extension for PN models including observability and controllability problems are as well simple to implement.

The link for free downloading the code for the examples developed in this chapter is: http://www.mathworks.com/matlabcentral/fileexchange/54959-simulation-of-discrete-event-systems-in-matlab

Author details

Raul Campos-Rodriguez*, Mildreth Alcaraz-Mejia* and Uriel Sanchez-Ramirez

*Address all correspondence to: rcampos@iteso.mx and mildreth@iteso.mx

Electronics, Systems and Informatics Department, ITESO University, Tlaquepaque, Jalisco, Mexico

References

[1] E. D. Sontag. *Mathematical Control Theory: Deterministic Finite Dimensional Systems*. Springer, 2nd Ed., New York, USA, 1998.

[2] C. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Springer, 2nd Ed., New York, USA, 2009.

[3] R. Boel and G. Stremersch (Editors). *Discrete Event Systems: Analysis and Control*. Springer, 1st Ed., New York, USA, 2000.

[4] G. A. Wainer and P. J. Mosterman (Editors). *Discrete-Event Modeling and Simulation: Theory and Applications*. CRC Press, 1st Ed., Florida, USA, 2010.

[5] R. Campos-Rodriguez and M. Alcaraz-Mejia. *A Matlab/Simulink Framework for the design of controllers and observers for discrete-event systems*, Electronics and Electrical Engineering, 2010, 3(99), pp. 63–68.

[6] R. Campos-Rodriguez, M. Alcaraz-Mejia and J. Mireles-Garcia. *Supervisory control of discrete event systems by using observers*. Proceedings of IEEE 15th Mediterranean Conference on Control & Automation, 2007, pp. 1–7, Athens, Greece, DOI 10.1109/MED.2007.4433816.

[7] J. E. Hopcroft and J. D. Ullman. Introduction to automata theory, languages, and computation, vol. 1. Addison-Wesley, 1979.

[8] P. J. Ramadge and W. M. Wonham. *Supervisory control of a class of discrete event processes*. SIAM J. Control and Optimization 25 (1), pp. 206–230. 1987.

[9] F. Wagner, R. Schmuki and T. Wagner. *Modeling Software with Finite State Machines: A Practical Approach*. Auerbach Publications, 2006.

[10] T. Murata. *Petri nets: Properties, analysis and applications*, Proceedings of the IEEE, 77 (4), pp. 541, 580. 1989.

[11] M. H. de Queiroz, J. E. R. Cury and W. M. Wonham. *Multitasking Supervisory Control of Discrete-Event Systems.* Discrete Event Dynamic Systems: Theory and Applications, 15, pp. 390–393, Dordrecht, The Netherlands, 2005. Example also available online at http://wwweb.eecs.umich.edu/umdes/manufact2.html