A Tool for Modeling and Simulation of Discrete-Event Systems

GPenSIM v4.0

General Purpose Petri Net Simulator

GPenSIM Web site:
http://www.davidrajuh.net/gpensim/

Reggie Davidrajuh
University of Stavanger, Norway
Email: reggie.davidrajuh@uis.no

Version 4.1 © September 2010

CONTENTS:

PREFACE

Petri net is being widely accepted by the research community for modeling and simulation of discrete event systems. There are a number of Petri net tools available for academic and commercial use. These tools are advanced tools powerful enough to model complex and large systems. In this book, we introduce a new Petri Net simulator called GPenSIM (General Purpose Petri Net Simulator). GPenSIM runs on MATLAB platform. GPenSIM is designed with one specific goal: *allowing Petri net models to integrate with other MATLAB toolboxes*.

By integrating Petri net models with other toolboxes, numerous benefits can be reaped. For example, by integrating with MATLAB Fuzzy Toolbox, we can experiment with Fuzzy Petri Nets; by combining with MATLAB Control Systems Toolbox, we can create hybrid discrete-continuous systems. Hence, the main goal of this book is to introduce GPenSIM – a platform with which we can create Petri net models incorporating many other toolboxes, libraries, and functions that are already available on the MATLAB platform.

There are many examples worked out in this book. These examples are simple and easy to follow. However, this book is not an introduction to Petri nets. Reader should know Petri net basics beforehand in order to start working with this book. Both the simulator GPenSIM and codes for examples (M-files) can be downloaded from the web site: http://www.davidrajuh.net/gpensim.

Reggie Davidrajuh
Stavanger, Norway
September 2010

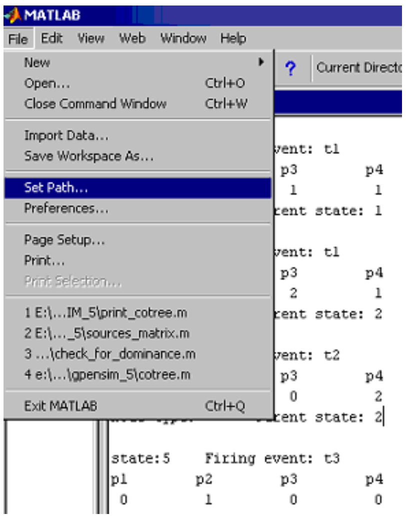
1. Installing GPenSIM

Installation takes five simple steps:

1. Unzip the GPenSIM pack:
Unzip the GPenSIM toolbox functions file(s) under a directory, say "d:\GPenSIM\GPenSIM32\". Note: Due to size limitations, there may be one zip file (GPenSIM-v4.0.zip) or two zip files (GPenSIM-v4.0-pack-1.zip and GPenSIM-v4.0-pack-2.zip) zip files.

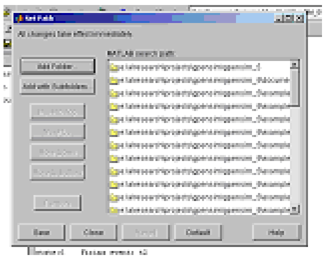
Similarly, unzip the examples file (Examples-v4.0.zip) under a directory, say "d:\GPenSIM\Examples\"

2. Set MATLAB Path Command:
Start MATLAB. Go to the file menu in MATLAB, and select "set path" command:


MATLAB
File Edit View Web Window Help
New
Open... Ctrl+O
Close Command Window Ctrl+W
Import Data...
Save Workspace As...
Set Path...
Preferences...
Page Setup...
Print...
Print Selection...
1 E:\...IM_5\print_cotree.m
2 E:\..._5\sources_matrix.m
3 ...\check_for_dominance.m
4 e:\...\gpensim_5\cotree.m
Exit MATLAB Ctrl+Q
Current Directo

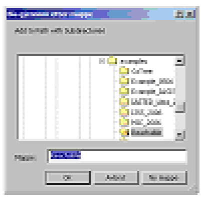
```
vent: t1
p3        p4
 1         1
rent state: 1

vent: t1
p3        p4
 2         1
rent state: 2

vent: t2
p3        p4
 0         2
rent state: 2

state:5    Firing event: t3
p1        p2        p3        p4
 0         1         0         0
```
Setting path command

Select "Add folder":



Adding folder

3. Add GPenSIM Directory:
A new dialog box will appear. Browse through the directories and select the directory where you have unzipped the GPenSIM toolbox functions.


Adding GPenSIM directory

4. Test Installation

Go to MATLAB command window and type ‘gpensim’; if the following (or similar) output is printed, then the installation is complete.

```
>> gpensim
--------
GPenSIM version 4.0;   Lastupdate: september 2010
http://www.davidrajuh.net/gpensim
--------
>>
```

2. Introducing Petri net

This section gives a brief introduction to Petri nets. For further details, interested readers are referred to Murata(1989); Davidrajuh (2003); Cassandras and Lafortune (2007) [10]. Carl Adam Petri invented Petri nets in 1962, as part of his dissertation titled “Kommunikation mit Automaten” at the University of Bonn (Petri and Reisig, 2008).

2.1 Elements of Petri nets

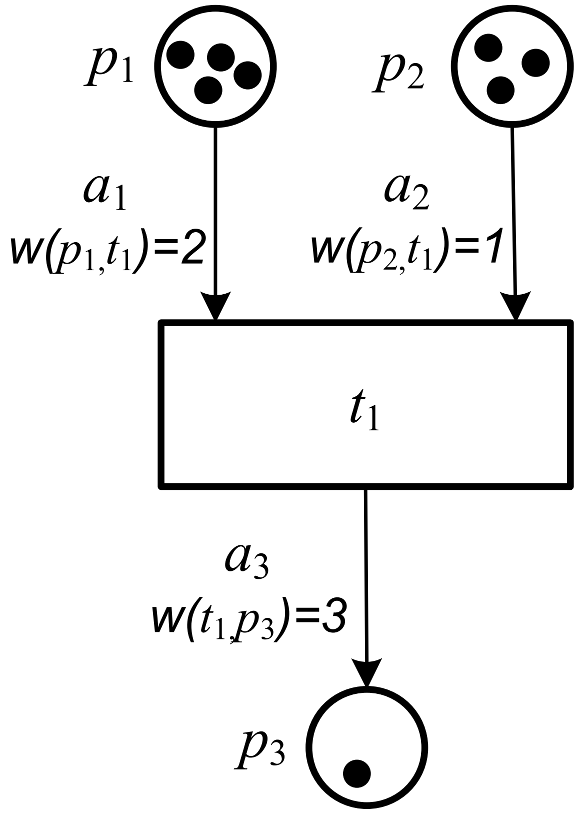


Figure-1. Sample Petri net

A Petri net contain four types of elements: tokens, places, arcs, and transitions. Tokens represent objects in the Petri net models, such as materials in a material flow system, data in a information flow. A token is represented with a dot in Petri net models. When the number of tokens becomes large, it is usually represented with the number of tokens; see figure 1.

Places are passive elements such as input and out buffers, conveyor belts, etc. Places hold tokens. Figure 1 shows places p_1, $p_2$and p_3 with 4, 3 and 1 tokens (black spots). Each place is capable of holding any number of tokens.

Arcs are connections between places and transitions. Arcs are bipartite meaning it is not possible to have an arc connecting two places together or two transitions together. Each arc has a weight, which is the number of tokens that are transported simultaneously when the transitions of which the arc is connected to fires.

Transitions are active elements like machines, robots, etc. Transitions correspond to events and are connected by arcs to places. When a transition fire, the number of tokens within the places connected to the firing transition, are changed according to the arcs weights and directions; when a transition fires it consumes tokens (input parts) from the input places and puts tokens (output parts) into the output places. For a transition to be able to fire, the number of tokens in the input places must be equal or higher than the weights of the arcs connecting the input places to the transition. The transition will then be an *enabled transition*. Figure 2 shows the state of the sample Petri net from figure 1 after the transition T1 has fired once.

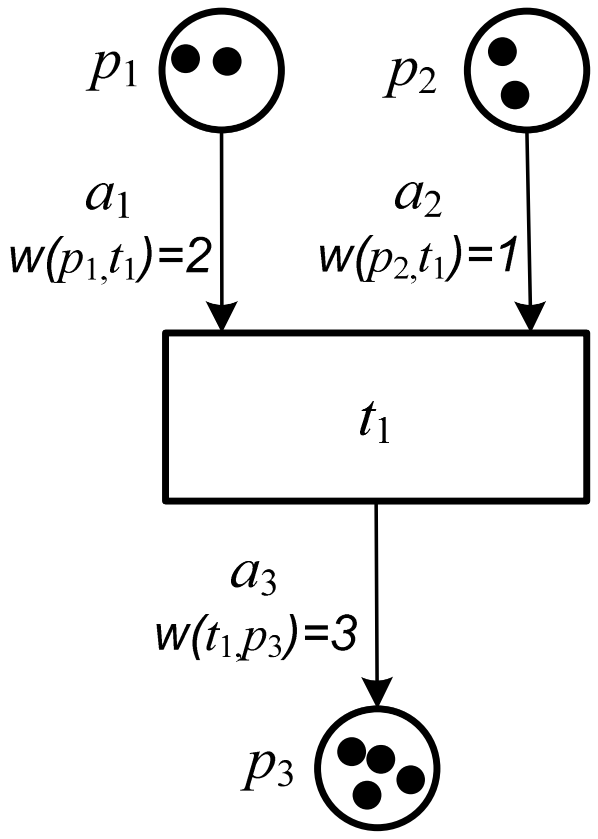


Figure-2. Sample Petri net after one cycle

2.2 Formal Definition of Petri nets

A Petri net is a four-tuple (P, T, A, x_0)

Where,

P is the set of places, $P = [p_1, p_2, \ldots, p_n]$,

T is the set of transitions, $T = [t_1, t_2, \ldots, t_m]$,

A is set of arcs (from places to transitions and from transitions to places),

$A \subseteq (P \times T) \cup (T \times P)$, and

x is the row vector of markings (tokens) on the set of places

$x = [x(p_1), x(p_2), \ldots, x(p_{n1})] \in N^n$, x_0 is the initial marking.

2.2.1 Input and Output Places of a Transition

In the Petri net in figure 2, the places p_1 and p_2 are inputs to transition t_1, and p_3 is an out place of transition t_1. It is convenient to use *I(t$_j$)* to represent the set of input places to transition t_j and *O(t$_j$)* to represent the set of output places to transition t_j when describing a Petri net:

$$I(T_j) = \{p_i \in P : (p_i, t_j) \in A\}$$
$$O(t_j) = \{p_i \in P : (p_i, t_j) \in A\}$$

We see from figure 2, that the weight of the arc from input place p_1 to transition t_1 has a weight = 2. This is denoted by: $w(p_1, t_1) \;=\; 2$.

2.3 Enabled Transitions

A transition $t_j \in T$ in a Petri net is said to be *enabled* if (Cassandras and Lafortune, 2007):
$x(p_i) \geq w(p_i, t_j)$ for all $p_i \in I(t_j)$.

The transition t_1 in figure 2 is enabled, since the numbers of tokens in the input places p_1 (2) and p_2 (2) are at least as large as the weight of the arcs connecting them to t_1 ($w(p_1, t_1) \;=\; 2$ and $w(p_1, t_1) \;=\; 2$).

2.4 Petri net dynamics

The markings of a Petri net, which is the distribution of tokens to the places, represent the state of the Petri net. A Petri net representing a discrete event system, where the transitions represent events, goes through many states during a simulation process. The different states could be represented with the row vector of markings (the 4.th-tuple):

$$x = [x(p_1), x(p_2), \ldots, x(p_{n1})]$$

The number of states an *infinite capacity net* can have is generally infinite, since each place can hold an arbitrary non-negative integer number of tokens (Murata, 1989). A *finite capacity net* on the other hand, will have a given number of possible states.

The *state transition function,* $f : \aleph^n \times T \to \aleph^n$, of a Petri net is defined for a transition $t_j \in T$ if and only if, $x(p_i) \geq w(p_i, t_j)$ for all $p_i \in I(t_j)$.
If $f(x, t_j)$ is defined then $x' = f(x, t_j)$, where
$x'(p_i) = x(p_i) - w(p_i, t_j) + w(t_j, p_i), \quad i = 1, \ldots, n.$

2.4.1 Coverability Tree

Petri Nets helps proving many behavioral properties of a system, including:

- Reachability, Boundedness, Conservativeness, Liveness, Reversibility

One technique used to prove properties of a Petri Net is a coverability tree; a coverability tree consists of a tree of markings and possible transitions between. Nodes that are a repetitive state are left as leaves and not extended. The Coverability tree can be infinite (markings consists 'omega') or finite (markings do not contain 'omega'). An infinite coverability tree is unbounded. Reachability is merely a question of whether there is a path from one node to another in the tree.

2.5 Why Petri nets?

Several tools could be used for simulation of discrete event systems; Automata, Stateflow, and Petri nets (high level) are some of the most commonly used (Davidrajuh and Molnar, 2009). The lack of structure possibilities (hierarchy) in Automata is a serious shortcoming, for modeling large systems since a large (and complex) system should be decomposed into modules and sub systems. Stateflow, developed by The MathWorks, extends the Simulink part of MATLAB with functionality similar to Petri net; charts are used for graphical representation of hierarchical and parallel states and for the event-driven transitions between them (Stateflow, 2010). A Petri net model of a discrete event system could easily be converted into a Stateflow model and vice versa, but learning Stateflow is much more difficult than learning Petri net due to the syntactic, semantic, and graphical details in Stateflow. Stateflow also requires some knowledge of Simulink, in addition to MATLAB, while the GPenSIM tool used for Petri net simulation in this paper runs under the MATLAB environment only. Petri nets is widely accepted by the research community for modeling and simulation of discrete event-driven systems, mainly due to graphical representation and the well defined semantics which makes it possible to use formal analysis of the models (Jensen, 1997).

2.6 A minute introduction to Petri net:

The simple Petri net shown in figure-3 is a model for business logic computation. The computation takes two database records and one business rule, and produces one business decision. In a Petri net, sources (like business rules and database records) and outputs (like business decisions) are called places, drawn as circles (e.g. Place-1). Computations (or events) are called transitions, drawn as vertical short bars (e.g. Transition-1). An arc connects a place to a transition, or a transition to a place, representing a path for a discrete part to flow. A place usually holds a number of parts, like database records. The number of parts inside a place is indicated by the tokens - black spots within a place.

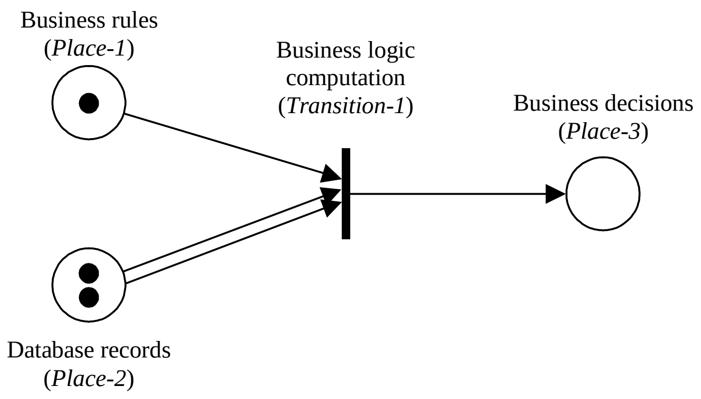


Figure-3. Petri net model for business logic computation

Part-I: GPenSIM Basics

3. Modeling with GPenSIM: The Basics

In GPenSIM, definition of a **Petri net graph** (***static*** details) is given in the **Petri net Definition File (PDF).** There may be a number of PDFs, if the Petri net model is divided into many modules, and each module is defined in a separate PDF. While the Petri net definition file has the static details, the **main simulation file (MSF)** contains the dynamic information (such as initial tokens in places, firing times of transitions) of the Petri net.

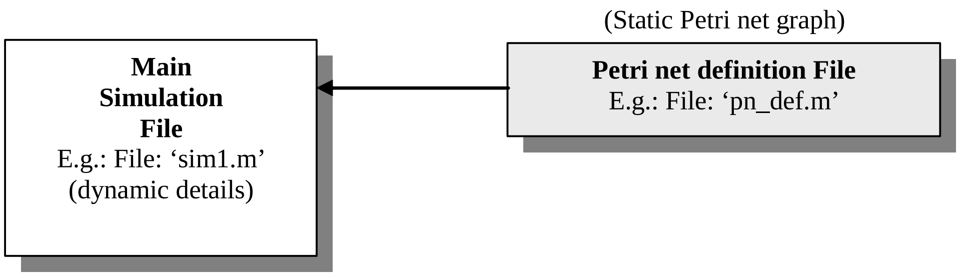


Figure-4. Separating the ***static*** and dynamic Petri net details

3.1 Transition Definition Files

In addition to these two files (main simulation file - MSF and Petri net definition file - PDF), there can be a number of **transition definition files (TDF)** too. These TDF are classified into two types: TDF_PRE and TDF_POST. TDF_PRE files are run before firing a transition; TDF_POST files are run after firing a transition.

3.1.1 Using TDF_PRE and TDF_POST

According to the Petri net theory, a transition can fire ("enabled transition") when there are enough tokens in the input places. However, in real-life situations, an event representing a transition can have additional restrictions for firing; for example, event-2 has preferences (priority) over event-1, thus event-2 is allowed to fire even though both event-1 and event-2 are enabled to fire. In GPenSIM literature, these additional restrictions are called "user-defined conditions".

The user-defined conditions for firing a transition are kept in a TDF_PRE file. *After a transition fires*, there may be some book keepings need to be done; these can be coded into a TDF_POST file.

Names of the TDFs must follow a strict naming policy, as they will be chosen and run automatically: for example, the TDF_PRE for the transition 'trans1' must be named 'trans1_pre.m'; similarly, the TDF_POST for the transition 'trans1' must be named 'trans1_post.m'.

3.1.2 Using TDF as a test probe

In addition to executing user-defined conditions, a TDF provides a unique functionality: acting as a probe to simulation engine: Let us explain:

1. The role of PDF: the only use of a PDF is to represent a static Petri net graph.

2. The role of MSF: A PDF will be loaded into memory by MSF right before the simulation start. Thus, an MSF first loads PDF (or PDFs in modular approach) into memory and then starts the simulation. MSF will be blocked during simulation runs, and when simulation is complete, the control will be passed back to MSF along with the simulation result. Therefore MSF does not have any control of what going on **during simulation**.
3. The role of TDF: Though MSF does not have any control of what going on **during simulation**, however, TDFs will be called during simulation, before and after transition firings. Thus, if we want to inspect run-time (simulation) properties then a TDF can be used as a probe (more details given in the section on TDF).

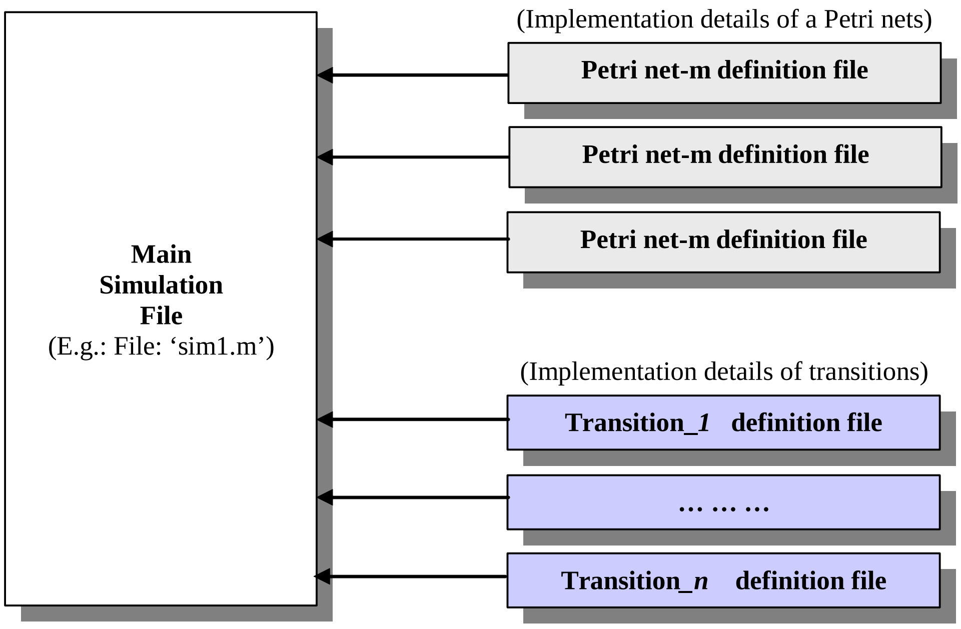

(Implementation details of a Petri nets)
Petri net-m definition file
Petri net-m definition file
Petri net-m definition file
Main Simulation File
(E.g.: File: 'sim1.m')
(Implementation details of transitions)
Transition_1 definition file
… … …
Transition_n definition file

Figure-5. Transition Definition Files

3.2 Global info

The different files (main simulation file MSF, Petri net definition files PDFs, and transition definition files TDFs) can access and exchange global parameters values through a packet called '**global_info**'. If a set of values is need to be passed to different files then these values are packed together as a **global_info** packet. **global_info** packet is visible in all the files, so that the values in the packet can be read and even changed. See chapter 9 for details.

3.3 Integrating with MATLAB environment

The most important reason for developing GPenSIM and the most advantage of it is its integration with the MATLAB environment, so that we can harness diverse toolboxes available in the MATLAB environment; see figure 6.

For example, by writing a user M-file that combines GPenSIM with Fuzzy Logic toolbox, we can experiment with Fuzzy Petri Nets; by combining GPenSIM with the Control systems toolbox, we can experiment hybrid discrete-continuous control applications, etc.

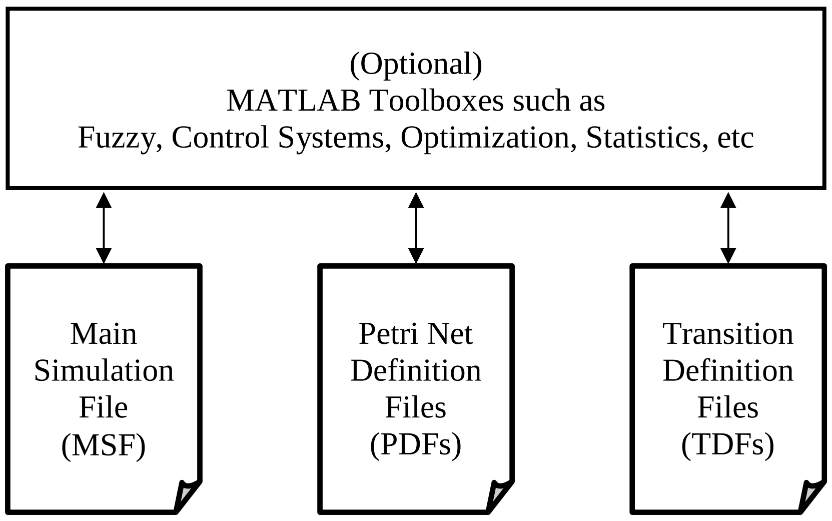

(Optional)
MATLAB Toolboxes such as
Fuzzy, Control Systems, Optimization, Statistics, etc
Main Simulation File (MSF)
Petri Net Definition Files (PDFs)
Transition Definition Files (TDFs)

Figure-6. Integrating GPenSIM with the MATLAB environment

4. Using GPenSIM

The methodology for creating a Petri net model consists of two steps:

Step-1. Defining the Petri net graph in a Petri net Definition File (PDF): this is the static part. This step consist of three sub-steps:

a. Identifying the basic elements of a Petri net graph: the places,
b. Identifying the basic elements of a Petri net graph: the transitions, and
c. Connecting the elements with arcs

Step-2. Assigning the dynamics of a Petri net in the Main Simulation File (MSF):

a. The initial markings on the places, and possibly
b. The firing times of the transitions

After creating a Petri net model, simulations can be done.

4.1 Example-01: A Simple Example

The two steps are explained below, using the sample Petri net model shown in figure 7.

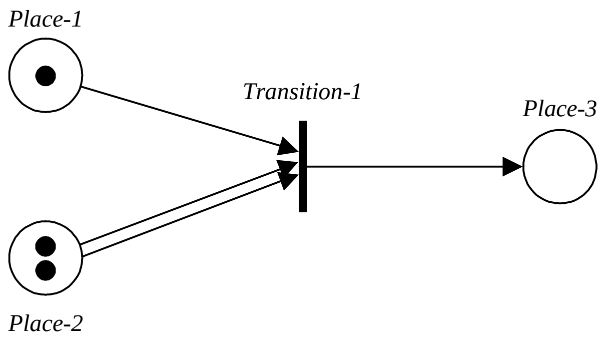


Figure-7. A Simple Petri Net Model

4.1.1 Step-1: Defining the Petri net graph

Defining the elements of a Petri net is done in a Petri net definition file (PDF). PDF is to identify the elements (places, transitions) of a Petri net, and to define the way these elements are connected.

The Petri net graph shown in figure 7 has three places, one transition, and three arcs. The PDF for the graph is given below:

```
% Example-01: A Simple Example
% file: 'simple_pn_def.m'
% this file defines the simple petri net graph
function [PN_name, set_of_places, set_of_trans, set_of_arcs] ...
     = simple_pn_def(global_info)

PN_name = 'A Simple Petri Net';
set_of_places = {'Place-1', 'Place-2', 'Place-3'};
set_of_trans = {'Transition-1'};
set_of_arcs =  {'Place-1', 'Transition-1', 1, ...
    'Place-2', 'Transition-1', 2, ...
    'Transition-1', 'Place-3', 1};
```

Explanation:

First, assign a name (or label) for the Petri net.

```
> PN_name = 'A Simple Petri Net';
```

Second, the places are to be identified with place names:

```
> set_of_places = {'Place-1', 'Place-2', 'Place-3'};
```

Third, the transitions are to be identified by stating their names.

```
> set_of_trans = {'Transition-1'};
```

Finally, how the elements are connected is defined: connecting arcs are to be defined by listing the source, the destination and the weights of each arc. For example, the first arc is from 'Place-1' (source), to 'Transition-1' (destination) with a unit arc weight:

```
> set_of_arcs =  {'Place-1', 'Transition-1', 1, ...
    'Place-2', 'Transition-1', 2, ...
    'Transition-1', 'Place-3', 1};
```

4.1.2 Step-2: The main simulation file: assigning the initial dynamics

After writing the Petri net definition file (PDF, e.g. 'simple_pn_def.m'), we need to write the main simulation file (MSF). In the MSF, first we load the *static* Petri net graph, by passing the name of the PDF (without the ending '.m') to the function 'petrinetgraph':

```
> png = petrinetgraph('simple_pn_def');
```

Second, the *dynamics* such as initial markings on the places and the firing times of the transition are to be assigned. Normally, we stuff these two information into a packet (e.g. 'dynamic_info' in this example) and then pass this packet to function 'gpensim'.

```
> dynamic_info.initial_markings = {'Place-1',3, 'Place-2',5};
> dynamic_info.firing_times = {'Transition-1', 10};
```

4.1.3 The Simulations

Function gpensim will do the simulations if the Petri net graph (the static part) and the initial markings and firing times (the dynamic part) are passed to it:

```
> Sim_Results = gpensim(png, dynamic_info);
```

The output argument Sim_Results is the simulation results.
The output argument Sim_Results is a structure for the simulation results. In order to comprehend the simulation results easily, the function '**print_statespace**' could be used.

4.1.4 Viewing the simulation results with 'print_statespace'

```
> print_statespace(Sim_Results);
```

The output is given below:

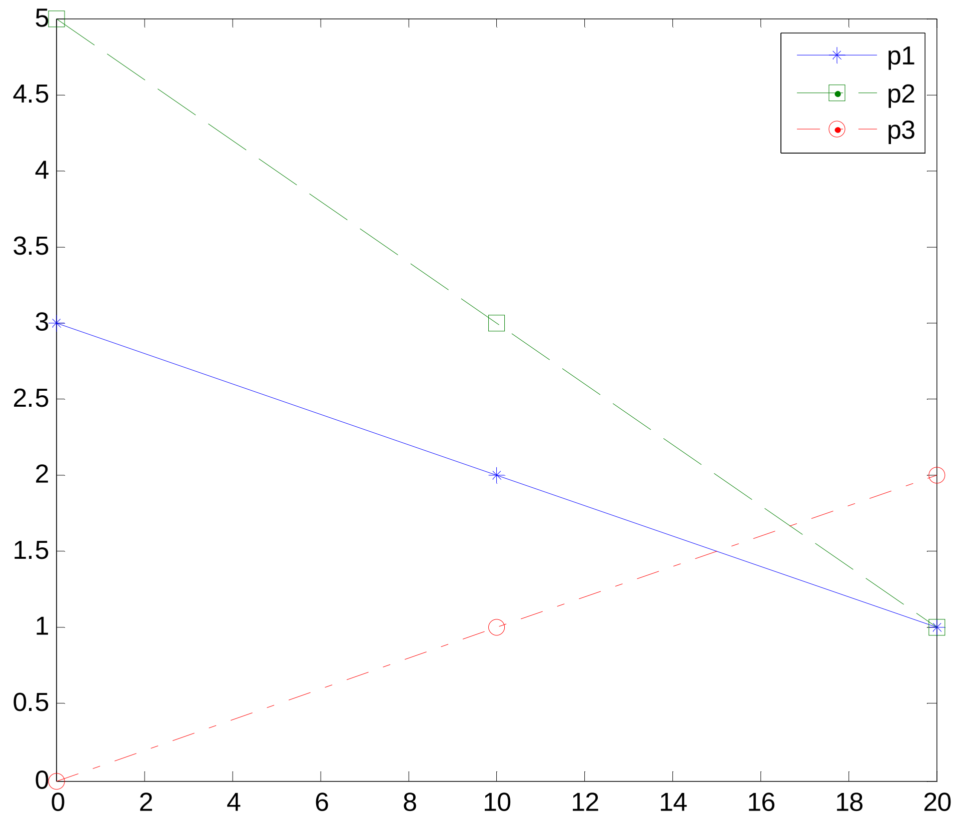
Explanation:
Of course, 'Transition-1' takes 10 milliseconds to produce a token on 'Place-3', after removing 1 and 2 tokens from 'Place-1' and 'Place-2' respectively.

```
Time: 0
New markings:
p1        p2        p3
 3         5         0

At time: 0  enabled transtions are:  t1

At time: 0  firing transtions are:  t1

Time: 10
Fired Transition: t1
New markings:
p1        p2        p3
 2         3         1

At time: 10  enabled transtions are:  t1

At time: 10  firing transtions are:  t1

Time: 20
Fired Transition: t1
New markings:
p1        p2        p3
 1         1         2

At time: 20  enabled transtions are:

At time: 20  firing transtions are:
>>
```

In addition to the ASCII output, we can also view the output graphically. For example,

```
> plotp(Sim_Results, {'Place-1', 'Place-2', 'Place-3'});
```

The above statement will plot how the tokens in the places vary with time: see the figure given below:



4.2 Summary

Step-1 is about creating the PDF that defines the static Petri net graph. The PDF for the Petri net shown in figure 5 is repeated below:

```
% Example-01: A Simple Example
% file: 'simple_pn_def.m'
% this file defines the simple petri net
function [PN_name, set_of_places, set_of_trans, set_of_arcs] ...
     = simple_pn_def(global_info)
PN_name = 'A Simple Petri Net implementation';
set_of_places = {'Place-1', 'Place-2', 'Place-3'};
set_of_trans = {'Transition-1'};
set_of_arcs =  {'Place-1', 'Transition-1', 1, ...
    'Place-2', 'Transition-1', 2, ...
    'Transition-1', 'Place-3', 1};
```

Step-2 is for assigning the initial dynamics (initial markings and firing times) in the MSF. After the assignment, the simulations can be run and the results can also be plotted. The MSF for the Petri net shown in figure 5 is repeated below:

```
% Example-01: A Simple Example
% the main file to run simulation
dynamic_info.initial_markings = {'Place-1',3, 'Place-2',5};
dynamic_info.firing_times = {'Transition-1', 10};

png = petrinetgraph('simple_pn_def');
Sim_Results = gpensim(png, dynamic_info);
```

```
print_statespace(Sim_Results);
plotp(Sim_Results, {'Place-1', 'Place-2', 'Place-3'});
```

4.3 Static PN structure

In the main simulation file given in the previous subsection, first we get a ***static*** Petri Net structure (called **png** in the example) as the output parameter of function **gpensim**:

```
png = petrinetgraph('simple_pn_def');
```

The static PN structure **png** is a compact representation of the static Petri net graph. A static PN structure consists of 5 elelements; e.g. in **png**:

```
                 name: 'A Simple Petri Net'
        global_places: [1x3 struct]
         No_of_places: 3
   global_transitions: [1x1 struct]
         global_arcs: [1x3 struct]
     incidence_matrix: [1.00 2.00 0 0 0 1.00]
```

The elements of a static PN structure are:

1) name: the ASCII string identifier of the Petri net
2) global_places: the set of all places in the Petri net
3) global_transitions: the set of all transitions in the Petri net
4) global_arcs: the set of all arcs in the Petri net, and
5) incidence_matrix: the matrix that depicts how the places and transitions are connected together.

It must be emphasized that ***static*** PN structure is much simpler than ***run-time*** PN structure. A static PN structure is one of the parameters that are input to the function **gpensim** to ***start simulation***. During simulation ('run-time'), state information and other run-time information will be added to the PN structure, thus the PN structure will contain dynamic information in addition to static details; during simulation the PN structure is called 'run-time' PN structure. Details of run-time PN structure is given in the next section.

4.4 Assigning names to Places & Transitions

CAUTION! There is a serious restriction in naming: ONLY first 10 characters of NAMES are significant.

This means, names for two places (**pReggieDav**idrajuh_1), and (**pReggieDav**idrajuh_2) are the same names (REFER TO THE SAME PLACE) because first 10 characters of these two names are the same.

However, (**pReggie_1_**Davidrajuh), and (**pReggie_2_**Davidrajuh) are different names simply because first 10 characters of these two names are different.

5. Transition Definition File (TDF)

The previous section explained the methodology for modeling and simulation with GPenSIM consisting of two steps. However, in the previous section, the step-1 was limited to creating only the PDF; there were no TDFs created. In this section, we shall discuss about the TDFs too, by working through the example shown in figure 8.

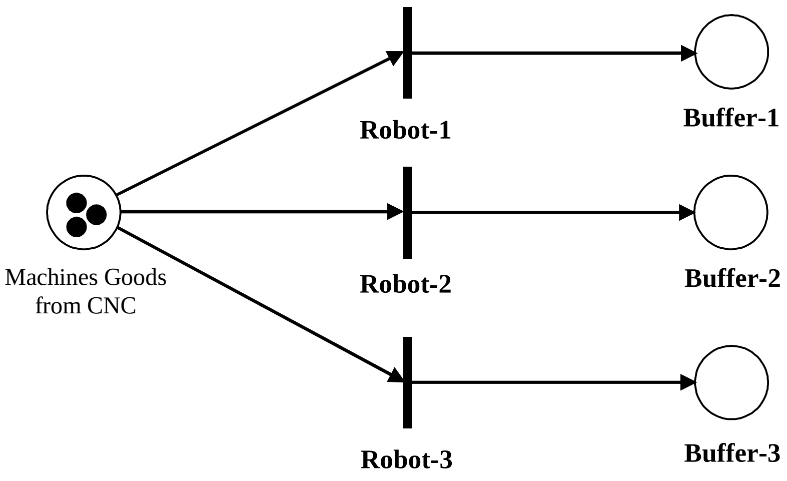


Figure-8. Petri net model of a production facility

5.1 Example-02: TDF_PRE Example

Figure 8 shows a Petri net model of a production facility where three robots are involved in sorting products (machined parts) from an input buffer (for machined goods) to output buffers. There are three output buffers (places) available. There are also three robots (transitions) that take the machined parts from the input buffer and put them to the respective output buffers.

The conditions: The output buffers have limited capacity. Buffer-1, buffer-2, and buffer-3 can accommodate a maximum of 3, 5, and 2, machined parts respectively. In addition, the robots should be operated in a manner that, at any time, buffer-2 should have more parts than buffer-1, and buffer-1 should have more parts than buffer-3.

The conditions stated above shall be coded in the TDF_PRE files.

5.1.1 Creating M-Files

In this example, the following M-files are created in the two steps:

- Step-1: In addition to creating the **PDF**, **TDF_PREs for the three transitions must be also created**. This is because, there are user-defined conditions attached to the transitions.

- Step-2: In the **MSF**: assigning the initial dynamics (initial markings and firing times) and running the simulations.

5.2 Step-1: the definition files

5.2.1 Defining the Petri net graph

Let's call the PDF for the Petri net in figure 6 as 'tdf_example_def.m':

```
% Example-02: TDF example
% file: tdf_example_def.m:
function [PN_name, set_of_places, set_of_trans, set_of_arcs] ...
     = tdf_example_def (global_info)

PN_name = 'TDF Example: Petri Net for production facility';
set_of_places = {'pFrom_CNC', 'pBuffer_1', 'pBuffer_2', 'pBuffer_3'};
set_of_trans = {'tRobot_1', 'tRobot_2', 'tRobot_3'};
set_of_arcs = {'pFrom_CNC','tRobot_1',1, 'pFrom_CNC','tRobot_2',1, ...
    'pFrom_CNC','tRobot_3',1, ...
    'tRobot_1','pBuffer_1',1, 'tRobot_2','pBuffer_2',1,...
    'tRobot_3','pBuffer_3',1};
```

5.2.2 Coding the user-defined firing conditions of the Transitions

tRobot-1 will fire only if the number of tokens (machined parts) already put in output **pBuffer-1** is less than 3. In addition, number of tokens in **pBuffer-1** should be less than that of **pBuffer-2**; coding these two user-defined conditions into the TDF_PRE for **tRobot-1** is given below. As the name of the transition is '**tRobot_1**', this TDF must be named '**tRobot_1_pre.m**'.

```
% file: tRobot_1_pre.m:
function [fire, new_color,override,selected_tokens,global_info] = ...
   tRobot_1_pre(PN, new_color, override, selected_tokens, global_info)

b1 = get_place(PN, 'pBuffer_1');
b2 = get_place(PN, 'pBuffer_2');
fire = (b1.tokens < b2.tokens)& (b1.tokens < 3);
```

Similarly, the definition files for **tRobot-2** and **tRobot-3** are created, satisfying the given conditions:

```
% file: tRobot_2_pre.m:
function [fire, new_color,override,selected_tokens,global_info] = ...
      tRobot_2_pre(PN, new_color,override,selected_tokens,global_info)

b2 = get_place(PN, 'pBuffer_2');
fire = (b2.tokens < 5);
```

```
% file: tRobot_3_pre.m:
function [fire, new_color,override,selected_tokens,global_info] = ...
     tRobot_3_pre(PN, new_color,override,selected_tokens,global_info)

b1 = get_place(PN, 'pBuffer_1');
b3 = get_place(PN, 'pBuffer_3');
fire = (b1.tokens > b3.tokens) & (b3.tokens < 2);
```

5.3 Step-2: Assigning the initial dynamics and running the simulations

Given below is the main simulation file (‘tdf_example.m’):

```
% Example-02: TDF example
% the main file to run simulation tdf_example.m
png = petrinetgraph('tdf_example_def');
dynamics.initial_markings = {'pFrom_CNC', 20};%initial machined parts
dynamics.firing_times = {'tRobot_1',10,'tRobot_2',5,'tRobot_3',15};

Results = gpensim(png, dynamics);
print_statespace(Results);
plotp(Results, {'pFrom_CNC', 'pBuffer_1', 'pBuffer_2', 'pBuffer_3'});
```

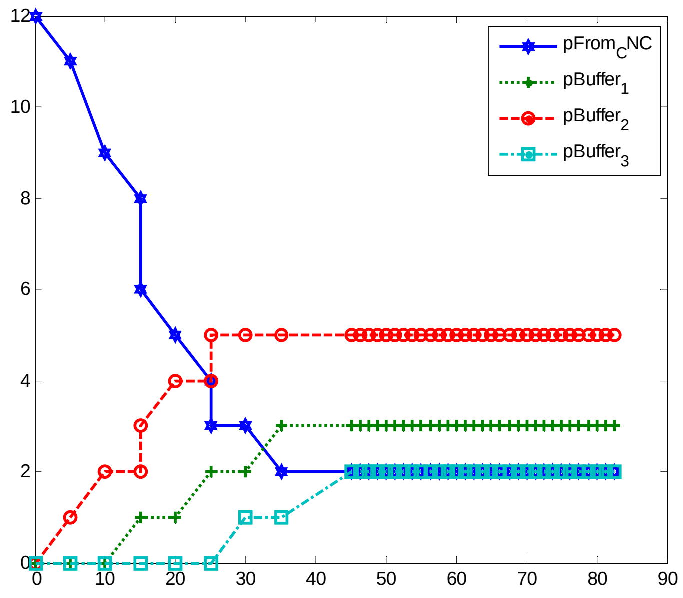
The output of **print_statespace** is given below is one of the 2 possible outcomes.

5.3.1 Outcome-1:

```
State:0 (Initial State)
pBuffer_1 pBuffer_2 pBuffer_3 pFrom_CNC
 0         0         0         10
At time: 0
  Enabled transtions are:
 tRobot_1   tRobot_2   tRobot_3
At time: 0
  Firing transtions are:
 tRobot_2

    Time: 5
State: 1
Fired Transition: tRobot_2
Current State:
pBuffer_1 pBuffer_2 pBuffer_3 pFrom_CNC
 0         1         0         9
At time: 5
  Enabled transtions are:
 tRobot_1   tRobot_2   tRobot_3
At time: 5
  Firing transtions are:
 tRobot_1   tRobot_2

    Time: 10
State: 2
Fired Transition: tRobot_2
Current State:
pBuffer_1 pBuffer_2 pBuffer_3 pFrom_CNC
 0         2         0         7
At time: 10
  Enabled transtions are:
 tRobot_1   tRobot_2   tRobot_3
At time: 10
  Firing transtions are:
 tRobot_1   tRobot_2

    Time: 15
State: 3
Fired Transition: tRobot_2
Current State:
```

```
pBuffer_1 pBuffer_2 pBuffer_3 pFrom_CNC
 0         3         0         6
At time: 15
  Enabled transtions are:
 tRobot_1   tRobot_2   tRobot_3
At time: 15
  Firing transtions are:
 tRobot_1   tRobot_2

    Time: 15
State: 4
Fired Transition: tRobot_1
Current State:
pBuffer_1 pBuffer_2 pBuffer_3 pFrom_CNC
 1         3         0         5
At time: 15
  Enabled transtions are:
 tRobot_1   tRobot_2   tRobot_3
At time: 15
  Firing transtions are:
 tRobot_1   tRobot_2   tRobot_3

……………….
……………..
   Time: 45
State: 10
Fired Transition: tRobot_3
Current State:
pBuffer_1 pBuffer_2 pBuffer_3 pFrom_CNC
 3         5         2         0
At time: 45
  Enabled transtions are:
>>
```

Given below is the plot of how the number of tokens in different places varies with time:



5.4 Run-time PN structure

Incidentally, TDF_PRE can also be used as a probe into simulation engine. The MSF prepares the static Petri net (PN) structure and the initial dynamic information so that the simulation can be started. Once the simulation is started, there is no way of knowing what's going on. The MSF is blocked until the simulation is complete and the result is given back to the MSF. Then, we can analyze the results e.g. with the help of **print_statespace**.

During simulations, control is passed to TDF_PRE if there is any. In the TDF, a copy of run-time PN structure is available so that we can inspect it to study what's going on. Let's take a look into TDF for **Robot_1** discussed in the previous subsection:

```
% file: tRobot_1_pre.m:
function [fire, new_color,override,selected_tokens,global_info] = ...
    tRobot_1_def(PN, new_color,override,selected_tokens,global_info)
...
PN  % dump contents of PN every time tRobot_1_pre is called
```

In TDF given above, we see that **run-time PN structure** is one of the 5 input parameters. This run-time PN structure has all the important run-time details; hence, we can inspect this PN structure to study what's going on during simulation. **Run-time PN structure has 21 elements**, given below are some of them possessing important run-time properties:

1. **PN.global_places:** has **complete set of current tokens** for each place
2. **PN.global_transtions:** has details about how many times **each transition has fired** so far
3. **PN.current_time:** the **internal clock time**
4. **PN.token_serial_number:** the total number of tokens generated so far
5. **PN.X:** the **current marking (current state)**
6. **PN.Firing_Transitions**: indicates **which transitions are currently firing**
7. **PN.Enabled_Transitions**: indicates **which transitions are currently enabled**

```
1    STATIC                    Name:  'TDF Example: Production facility'
2   Run-time          global_places:  [1x4 struct]
3   Run-time     global_transitions:  [1x3 struct]
4    STATIC             global_arcs:  [1x6 struct]
5    STATIC        incidence_matrix:  [3x8 double]
6   Run-time           current_time:  45.00
7   Run-time    token_serial_number:  30.00
8   Run-time                      X:  [10.00 3.00 5.00 2.00]
9   Run-time     Firing_Transitions:  [0 1 1]
10  Run-time    Enabled_Transitions:  [1 0 0]
```

5.5 Example-03: Implementing Preference through TDF_PRE

In this example (figure 9), transitions **t1** and **t2** both competes for tokens in **pS;** we prefer **t1** over **t2**.

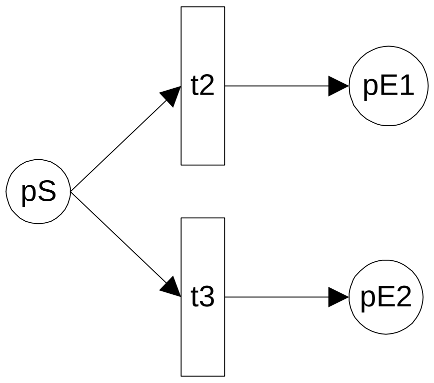


Figure-9. Petri net model of a production facility

MSF:

```
% MSF: prefer.m
dyn.firing_times = {'t1',10, 't2',7};
dyn.initial_markings = {'pS',3};

png = petrinetgraph('prefer_def');
sim_results = gpensim(png, dyn);
print_statespace(sim_results);
plotp(sim_results, {'pE1', 'pE2'});
```

PDF:

```
function [PN_name, set_of_places, set_of_trans, ...
    set_of_arcs] = prefer_def(global_info)
% PDF: prefer_def

PN_name='Preference example';
set_of_places={'pS', 'pE1', 'pE2'};
set_of_trans={'t1','t2'};
set_of_arcs = {'pS','t1',1, 't1','pE1',1,...
    'pS','t2',1, 't2','pE2',1};
```

5.5.1 Case-I: t1 is strictly preferred

Conditions for firing:

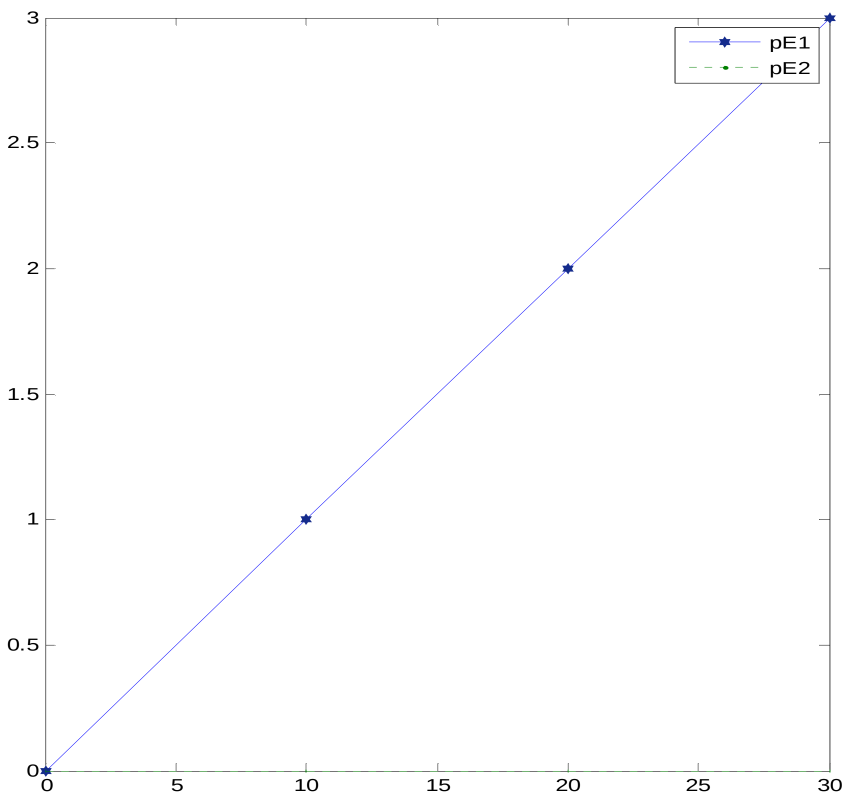
- **t1** will fire if it is enabled (meaning, no TDF for **t1**).
- **t2** will fire only is **t1** is not enabled

Surely, t2 will starve!

```
function [fire,PN, new_color,override,selected_tokens,global_info] = ...
    t2_pre (PN, new_color, override, selected_tokens, global_info)
```

```
% TDF_PRE for t2  ('t2_pre.m')

% Case-I:
if is_enabled(PN, 't1'),
    fire = 0;
else
    fire = 1;
end;
```

Simulation results:



```
Time: 0
New markings:
pS        pE1       pE2
 3         0         0

At time: 0  enabled transtions are:  t1 t2

At time: 0  firing transtions are:  t1

Time: 10
Fired Transition: t1
New markings:
pS        pE1       pE2
 2         1         0

At time: 10  enabled transtions are:  t1 t2
At time: 10  firing transtions are:  t1

Time: 20
Fired Transition: t1
New markings:
pS        pE1       pE2
 1         2         0

At time: 20  enabled transtions are: t1 t2

At time: 20  firing transtions are:  t1

Time: 30
Fired Transition: t1
New markings:
pS        pE1       pE2
 0         3         0

At time: 30  enabled transtions are:
At time: 30  firing transtions are:
```

5.5.2 Case-II: t1 is preferred, but t2 can also fire

Conditions for firing:

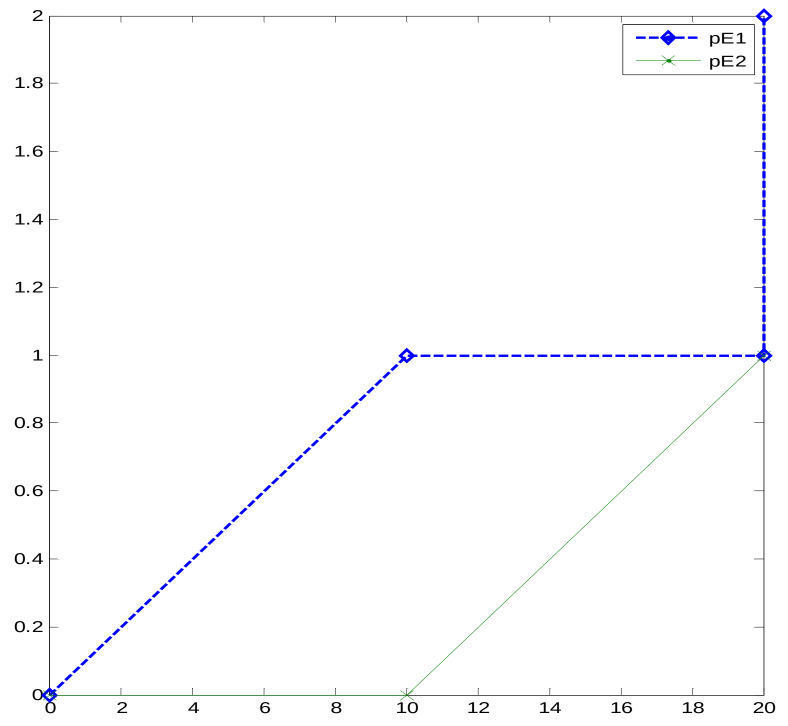
- (as before) **t1** will fire if it is enabled (meaning, no TDF for **t1**).
- **t2** will fire is **t1** is not enabled or *if **t1** has fired at least once*

Now, **t2** can fire as soon as **t1** has fired for the first time.

TDF:

```
function [fire,PN, new_color,override,selected_tokens,global_info] = ...
    t2_pre (PN, new_color, override, selected_tokens, global_info)
% TDF for t2  ('t2_pre.m')

% Case-II:
t1 = get_trans(PN, 't1');

if or(~is_enabled(PN, 't1'), (t1.times_fired >= 1)),
    fire = 1;
else
    fire = 0;
end;
```

Simulation results:
The following may occur where t2 may also fire.



5.6 Using TDF_POST

We study an application of TDF_POST through an example in section XXX.

6. Internal Clock

Internal clock is discrete in the sense it is updated whenever a transition is complete. If we take a close look into the figures generated by the **plotp** function, the figures look like ramp rather than pulses. This is due to poor sampling (recording), as simulation results with timing are recorded only when a transition complete firing. In other words, simulation results are recorded only when there is a new state.

We will discuss an import internal clock issue thorough an example. When a transition completes firing, the internal clock is advanced by the firing time of the transition. When a Petri net system has enabled transitions, but none is firing, then the internal clock time is advanced by an amount which is equal to ¼ of the minimum firing time of all transitions.

6.1 Example-04: Delay Example

In the figure shown below, let **p1** has 5 initial tokens. Also let firing time of **t1** is 7 seconds.

Though **t1** can fire 5 times successively, we want it to fire only at the start of every 30 seconds. This means, **t1** is delayed by 30 - 7 = 23 seconds.

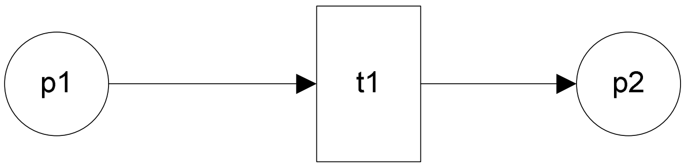


Figure-10. Delay Example

During the waiting time of 23 seconds (**t1** is enabled but not firing), time advancement will be done in time units of 7/4 = 1.75 seconds. See the gpensim system file '*timed_pensim.m*' for implementation details.

MSF:

```
% Example-04: delay example
% file: delay_demo.m:

png = petrinetgraph('delay_demo_def');

dynamic.initial_markings = {'p1',3};
dynamic.firing_times = {'t1', 7};

sim = gpensim(png, dynamic, global_info);
print_statespace(sim);
plotp(sim, {'p1','p2'});
```

PDF:

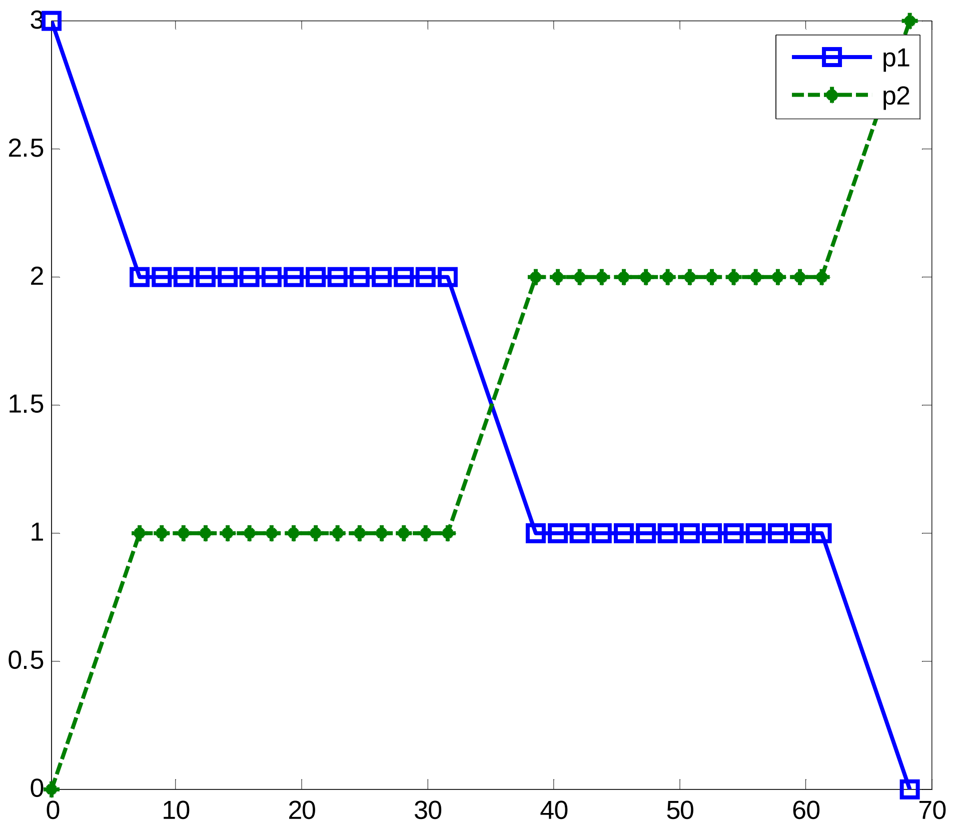
```
% Example-04: delay example
% file: delay_demo_def.m:
function [PN_name, set_of_places, set_of_trans, set_of_arcs]...
    = delay_demo_def(global_info)

```

```
PN_name = 'Delay Demo';
set_of_places = {'p1', 'p2'};
set_of_trans = {'t1'};
set_of_arcs =  {'p1', 't1', 1,  't1', 'p2', 1};
```

TDF:

```
function [fire, new_color, over_ride, selected_tokens,global_info] = ...
    t1_def(PN, new_color, over_ride, selected_tokens, global_info)
% function fire = t1_pre

rest = mod(PN.current_time, 30);
fire =  (rest < 5);   % any number less than 7 would do
```

Simulation results:



7. Measuring Activation Timing

We are going to find out how much time each transitions take or occupy out of the total time. From the simulation results, there are two functions that can compute activation time of each transition given in the input list. Function '**extractt**' creates a simple matrix called ***duration matrix*** in which first column is the transition (transition index) that fired, the second column is the start time for firing and the third column is the completion time for firing.
Function 'extractt' returns duration matrix with three columns:

1) Column-1: The firing transition
2) Column-2: firing start time
3) Column-3: firing finishing time

Alternatively, we can use the function '**occupancy**' to measure activation times: function **occupancy** first computes the duration matrix by calling the function **extractt**. Then, from the duration matrix, it computes the ***occupancy matrix***. Occupancy matrix consists of just two rows. The first row presents total activation times of each transition given in the input list. The second row presents activation in percentage of the total time. The function occupancy also prints the activation times and percentages on screen.

7.1 Example-05: Measuring Activation Time

This example is the same delay example, shown in figure 10. This time, we will compute the idle time of the transition (activation time of the transition, precisely) with the help of the functions **extractt** and **occupancy**.

The only change this time in the MSF is that addition of the last two lines:

MSF:

```
% Example-05: delay example for measuring activation time
% file: delay_demo.m:

png = petrinetgraph('delay_demo_def');

dynamic.initial_markings = {'p1',3};
dynamic.firing_times = {'t1', 7};

sim = gpensim(png, dynamic, global_info);
% print_statespace(sim);
% plotp(sim, {'p1','p2'});

duration_matrix  = extractt(sim, {'t1'})
occupancy_matrix = occupancy(sim, {'t1'})
```

Simulation results:
The duration matrix computed form the simulation results shows that the transition **t1** was fired at 0, 30, and 60 time units, and that every firing took 7 time units to complete.

Thus, the total time **t1** fired was 21 time units, and the activation percentage was (21/67 = 31.3%) percent.

```
duration_matrix =
     1     0     7
     1    30    37
     1    60    67

occupancy t1        :
  total time: 21
  Percentage time: 31.3433%

occupancy_matrix =
   21.0000
   31.3433
```

7.2 Example-06: Measuring Activation time

This is another example for measuring activation time. Figure 11 below shows a simple system where two transitions fire sequentially, one after the other.

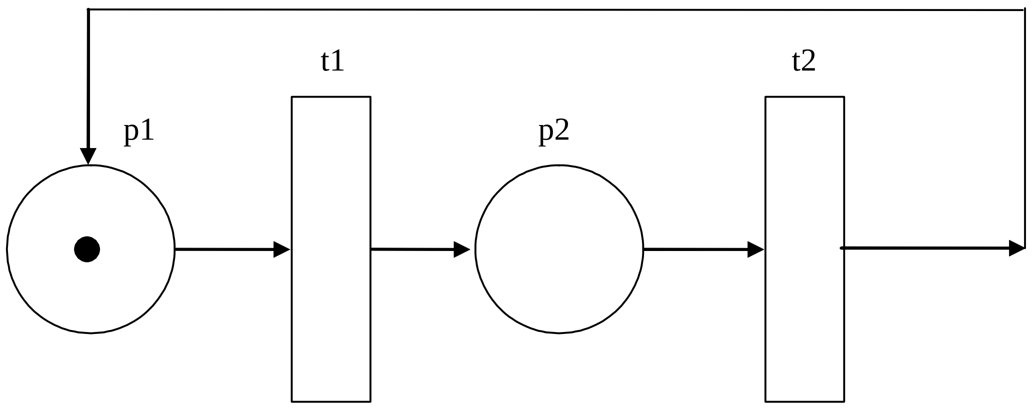


Figure-11. Transitions firing sequentially

The code below is for the main simulation file.

```
% Example-06: Measuring Timing
% MSF: measure_timing.m
clear, clc;
global_info.MAX_LOOP = 11; % GLOBAL DATA: MAX. SIMULATION CYCLES

png = petrinetgraph('measure_timing_def');
dynamicpart.initial_markings = {'p1', 10};
dynamicpart.firing_times = {'t1', 1, 't2', 100};
sim = gpensim(png, dynamicpart, global_info);
% print_statespace(sim); plotp(sim, {'p1', 'p2'});

duartion_martix = extractt(sim, {'t1', 't2'});
disp('Duartion Martix : '), disp(duartion_martix);
fprintf('\n\n');
occupancy_martix = occupancy(sim, {'t1', 't2'});
fprintf('\n\n');
disp('Occupancy Martix : '), disp(occupancy_martix);
```

Simulation results:

```
Duartion Martix :
     1     0     1
     1   101   102
     1   202   203
     1   303   304
     1   404   405
     1   505   506
     2     1   101
     2   102   202
     2   203   303
     2   304   404
     2   405   505


Simulation Completion Time: 506
occupancy t1        :
  total time: 6
  Percentage time: 1.1858%
occupancy t2        :
  total time: 500
  Percentage time: 98.8142%


Occupancy Martix :
    6.0000  500.0000
    1.1858   98.8142
```

8. Stochastic Firing Times

So far, the *firing times* for transitions are assumed to be *deterministic*; thus, the simulations presented so far are deterministic. However, in real life systems all the firing times are *stochastic*. GPenSIM provides a limited facility for stochastic firing times.

We can use any of the MATLAB-standard probability distribution functions for stochastic firing times. The following are the most used:

1) Guassian (normal) random function,
2) Binormial random function,
3) Poission random function, and
4) Uniform random function.

8.1 Example-07: Stochastic firing times

We refer to the CNC production system shown in figure 9; we no longer assume that the firing times are deterministic:

1) **Robot-1** takes random time Binaomially distributed with seed 10 and factor 0.9 milliseconds. (‘`binornd(10,0.9)`’)
2) **Robot-2** takes random time normally distributed with mean 1 and standard deviation 0.1 milliseconds. (‘`normrnd(1,0.1)`’)
3) **Robot-3** takes random time uniformly distributed with min 8 and max 10 milliseconds. (‘`unifrnd(8,10)`’)

Thus, the Petri net definition file is to be changed accordingly:

```
% Example-07: TDF example with stochastic timing
% the main simulation file
png = petrinetgraph('tdf_example_def');
dynamics.initial_markings = {'pFrom_CNC', 20}; % initial tokens

% here comes the STOCHASTIC TIMING
dynamics.firing_times = {'tRobot_1', 'binornd(10,0.9)',...
    'tRobot_2', 'normrnd(1,0.1)', 'tRobot_3', 'unifrnd(8,10)'};

Results = gpensim(png, dynamics);
print_statespace(Results);
plotp(Results, {'pFrom_CNC', 'pBuffer_1', 'pBuffer_2', 'pBuffer_3'});
```

Note: Due to stochastic timing, up to three different outcomes are possible!!

9. Modular Model Building

Figure 12 shows architecture of an adaptive supply chain based on service component architecture; see Davidrajuh (2007) for details. Figure 13 shows the equivalent Petri net model.

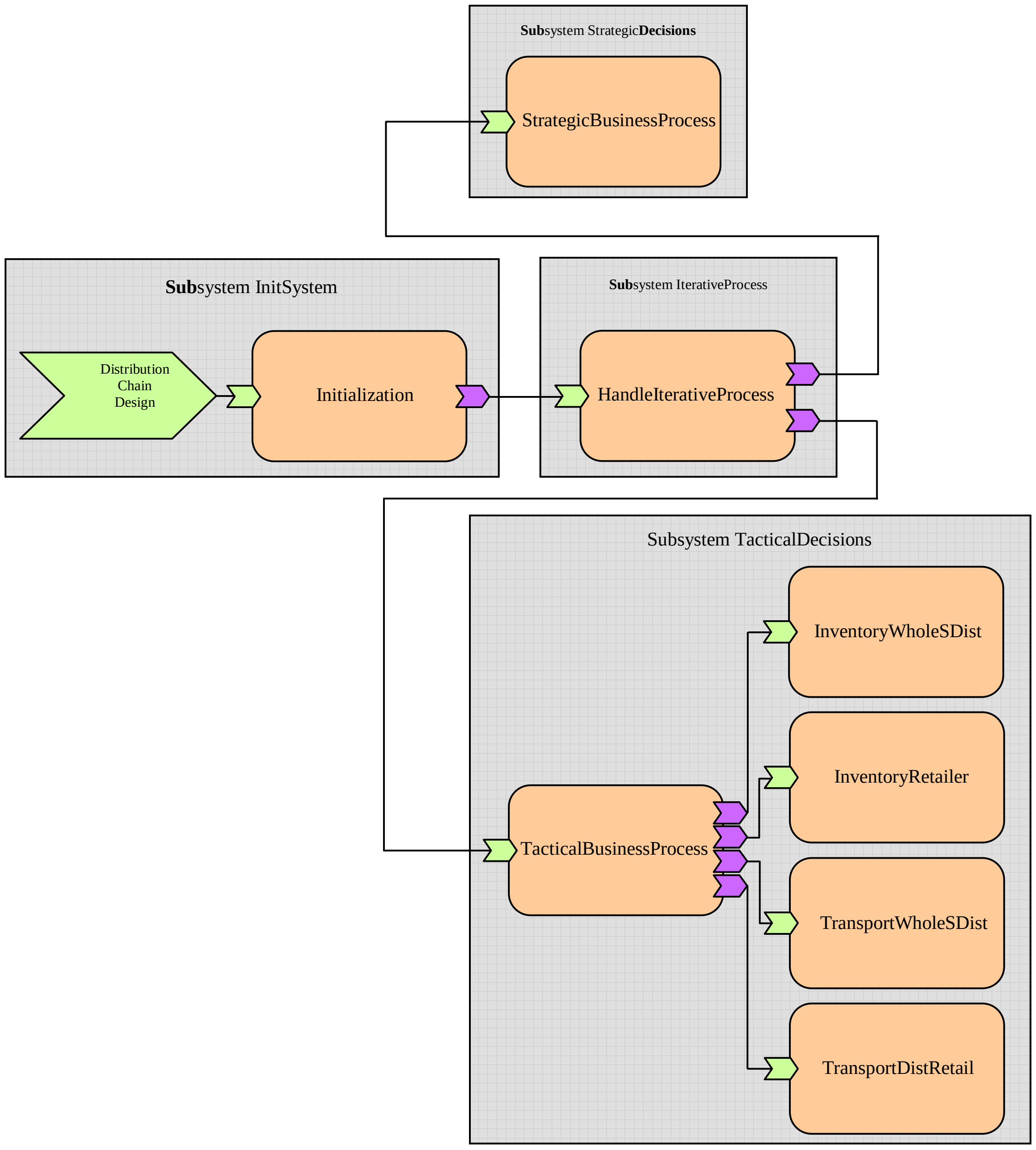


Figure-12. The system assembly

9.1 Example-08: Modular Model for Adaptive Supply Chain

The Petri net model shown in figure 13 has many elements (11 places and 12 transitions) and many connections (27 arcs). Though possible, it will be cumbersome to create one Petri net definition file PDF for the whole Petri net graph. Instead, we can divide the Petri net graph into modules as shown in figure 13, and then create individual PDFs for each of the module; finally, all the PDFs are combined to form the complete model.

In the following subsection, we use modular (many PDFs, one PDF for each module) approach. Section 9.2 presents the TDF for the transition tRES; interested reader is referred to Davidrajuh (2007) for details.

Figure-13. The Petri net model of the distribution chain

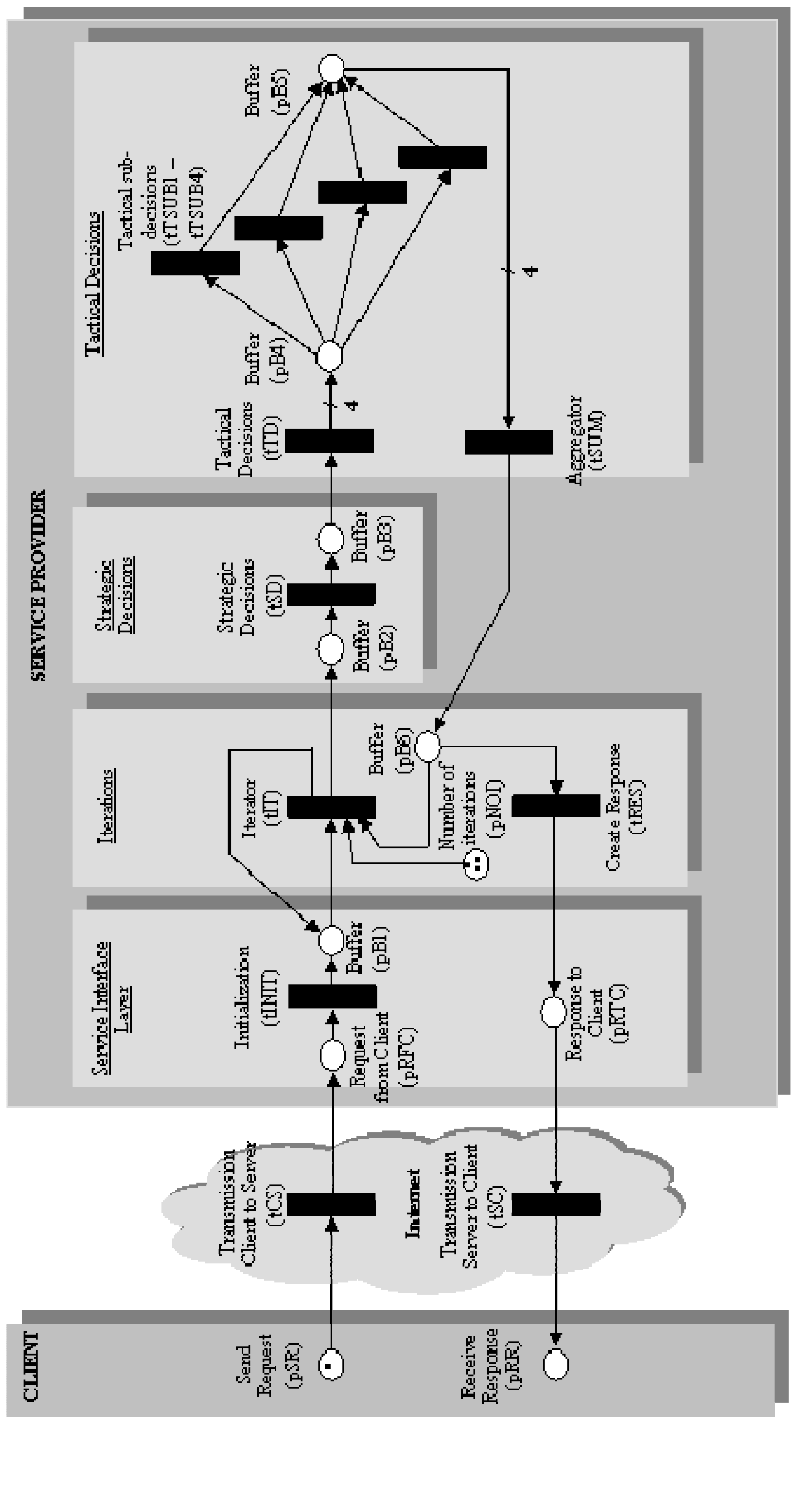


Figure 3: The Petri net model of the distribution chain

9.2 The Modular Approach

Figure 13 shows a modular Petri net model, consisting of a number of modules such as ‘Service Interface Layer’, ‘Initialization module’, ‘Strategic module’, etc. For each module, a PDF will be created. In addition, there will be a PDF for the connection between modules. For example, we can cerate a PDF for each of the following:

1) Client (‘client_def.m’),
2) Internet transmission (‘internet_def.m’),
3) Service Interface Layer (‘sil_def.m’),
4) Initialization module (‘init_def.m’),
5) Iterations module (‘interate_def.m’),
6) Strategic module (‘strategy_def.m’),
7) Tactical & sub tactical module (tactic_def.m’), and finally
8) Profile for connecting the modules together (‘conn_pro.m’).

In the main simulation file, all these 8 PDFs must be passed to the function ‘petrinetgraph’.

9.2.1 The main simulation file: ‘MIC_2006_new.m’

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%  MIC - 2006 (modular model)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
png = petrinetgraph({'client_def', 'internet_def', 'sil_def',
'conn_pro',...
    'iterate_def', 'strategy_def', 'tactic_def'});

dyn.initial_markings = {'pSR',1, 'pNOI', round(unifrnd(2,4)), 'pB6',1};
dyn.firing_times = {'tCS','normrnd(5000,50)', 'tSC','normrnd(5000,50)',...
    'tINIT','unifrnd(280,320)',...
    'tRES','unifrnd(1, 10)', 'tSD','unifrnd(80, 100)',...
    'tTD','unifrnd(25, 35)', 'tSUB1','unifrnd(10, 15)',...
    'tSUB2','unifrnd(10, 15)', 'tSUB3','unifrnd(10, 15)',...
    'tSUB4','unifrnd(10, 15)'};

Results = gpensim(png, dyn);
print_statespace(Results);
```

9.2.2 Client (‘client_def.m’)

```
function [PN_name, set_of_places, set_of_trans, set_of_arcs] ...
        = client_def()
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% File: client_def.m : Definition of client
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
PN_name = 'Client';
set_of_places = {'pSR', 'pRR'};
set_of_trans = [];
set_of_arcs = [];
```

9.2.3 Internet transmission (‘internet_def.m’),

```
function [PN_name, set_of_places, set_of_trans, set_of_arcs] ...
         = internet_def()
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% File: internat_def.m: Definition of internet transmission
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
PN_name='Internet Transmission';
set_of_places = [];
set_of_trans = {'tCS','tSC'};
set_of_arcs = [];
```

9.2.4 Service Interface Layer ('sil_def.m'),

```
function [PN_name, set_of_places, set_of_trans, set_of_arcs] ...
        = sil_def()
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% File: sil_def.m: Definition of the Service Interface Layer
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
PN_name='Service Interface Layer';
set_of_places = {'pRFC', 'pRTC', 'pB1'};
set_of_trans = {'tINIT'};
set_of_arcs = {'pRFC','tINIT',1, 'tINIT','pB1',1};
```

9.2.5 Iterations module ('interate_def.m')

```
function [PN_name, set_of_places, set_of_trans, set_of_arcs] ...
         = iterate_def()
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% File: iterate_def.m: Definition of the Iterations module
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
PN_name='Iterations Module';
set_of_places = {'pNOI', 'pB6'};
set_of_trans = {'tIT','tRES'};
set_of_arcs = {'pNOI','tIT',1, 'pB6','tIT',1, 'pB6','tRES',1};
```

9.2.6 Strategic module ('strategy_def.m')

```
function [PN_name, set_of_places, set_of_trans, set_of_arcs] ...
         = strategy_def()
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% File: strategy_def.m: Definition of the Strategic Module
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
PN_name = 'Strategic Module';
set_of_places = {'pB2', 'pB3'};
set_of_trans = {'tSD'};
set_of_arcs = {'pB2','tSD',1, 'tSD','pB3',1};
```

9.2.7 Tactical & sub tactical module ('tactic_def.m')

```
function [PN_name, set_of_places, set_of_trans, set_of_arcs] ...
        = tactic_def()
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% File: tactic_def.m: Definition of the Tactical & subtactical modules
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
PN_name = 'Tactical & sub-tactical Module(s)';
set_of_places = {'pB4', 'pB5'};
set_of_trans = {'tTD','tSUB1','tSUB2','tSUB3','tSUB4','tSUM'};
set_of_arcs = {'tTD','pB4',4, ...
    'pB4','tSUB1',1, 'pB4','tSUB2',1, 'pB4','tSUB3',1, 'pB4','tSUB4',1,...
    'tSUB1','pB5',1, 'tSUB2','pB5',1, 'tSUB3','pB5',1, 'tSUB4','pB5',1, ...
    'pB5','tSUM',4};
```

9.2.8 Profile for connecting the modules together ('conn_pro.m')

```
function [PN_name, set_of_places, set_of_trans, set_of_arcs] ...
         = conn_pro()
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% File: conn_pro.m: Definition of the connections between the modules
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
PN_name = 'Connections Profile';
set_of_places = [];
set_of_trans = [];
set_of_arcs = {'pSR','tCS',1,...    % client - internet
    'tCS','pRFC',1, ...             % internet - SIL
    'pRTC','tSC',1, ...             % SIL - internet
    'tSC','pRR',1,...               % internet - client
    'pB1','tIT',1,...               % init - iterations
    'tIT','pB1',1, ...              % iterations - init
    'tIT','pB2',1,...               % iterations - strategy
    'pB3','tTD',1, ...              % strategy - tactical
    'tSUM','pB6',1,...              % tactical - iterations
    'tRES','pRTC',1,...             % iterations - SIL
    };
```

9.3 Transition definition file for tRES ('tRES_def.m')

```
function [fire, new_color, override, selected_tokens, global_info] = ...
      tRES_def (PN, new_color, override, selected_tokens, global_info)
%% function tRES_def
%%

p1 = get_place(PN, 'pNOI');
fire =  (p1.tokens == 0);
```

10. Coverability Tree

Coverability tree (co-tree) is a very important issue in the analysis of Petri net models. In coverability analysis, we determine the states that are reachable from a given initial state.

This section shows how GPenSIM can be used to obtain co-tree of a Petri net. The methodology is creating a co-tree of a Petri net is almost same as running simulations on a Petri net; the only difference is that in step-3, instead of the function 'gpensim', we use the function 'cotree':

- Step-1. Creating Petri net definition files (PDFs) and transition definition files (TDFs)
- Step-2. Creating main simulation file (SMF) with dynamic info (initial markings and firing times)
- Step-3. Running the SMF using the function 'cotree' instead of 'gpensim'

10.1 Example-09: Cotree with finite states

This simple example deals with the Petri net shown in figure 14. The co-tree of this Petri net is shown in figure 15. Let us find the co-tree using GPenSIM:

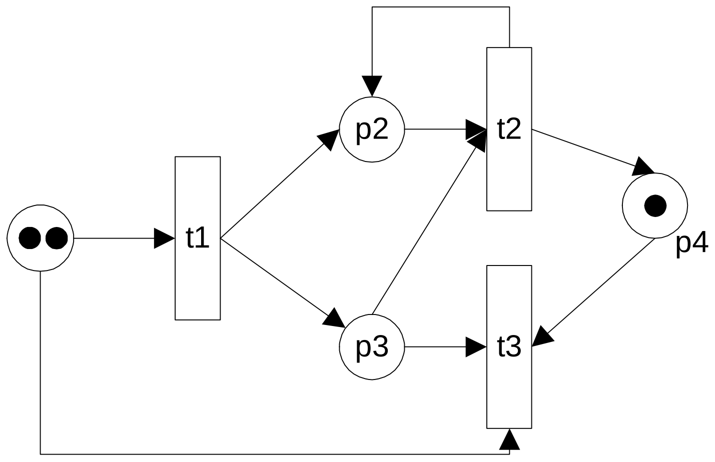


Figure-14. The Petri net for coverability analysis

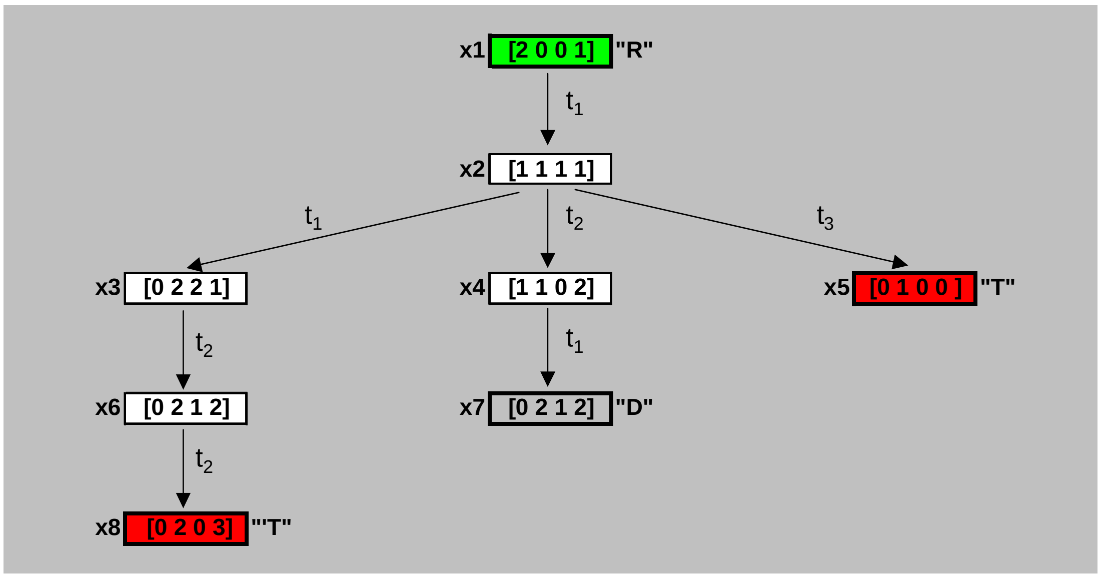


Figure-15. The reachable states of the Petri net shown in figure 14.

10.1.1 Petri net definition file

The Petri net definition file is given below:

```
% PDF for Example-09: Cotree example-1
function [PN_name, set_of_places, set_of_trans, set_of_arcs]...
    = cotree_example_def()

PN_name = 'COTREE Example: Petri Net in Figure 14';
set_of_places = {'p1', 'p2', 'p3', 'p4'};
set_of_trans  = {'t1', 't2', 't3'};
set_of_arcs   = {'p1','t1',1, 't1','p2',1, 't1','p3',1, ...
                  'p2','t2',1, 'p3','t2',1, 't2','p2',1,'t2','p4',1,...
                  'p1','t3',1,'p3','t3',1,'p4','t3',1};
```

10.1.2 The main file

The main file (after phases 2 & 3) is given below:

```
% Example-09: Cotree example-1
% the main file to find the reachable states
clear, clc; % clear the workspace & screen first

png = petrinetgraph('cotree_example_def');
dyn.initial_markings = {'p1', 2, 'p4', 1}; % tokens initially
Results = cotree(png, dyn.initial_markings);
print_cotree(Results);
```

The function **print_cotree** will print the following on the screen, which is equivalent to the graphical co-tree shown in figure 14

```
COTREE Example: Petri Net in Figure 14

state:1  ROOT node
p1        p2        p3        p4
 2         0         0         1

state:2    Firing event: t1
p1        p2        p3        p4
 1         1         1         1
Node type: ' '   Parent state: 1

state:3    Firing event: t1
p1        p2        p3        p4
 0         2         2         1
Node type: ' '   Parent state: 2

state:4    Firing event: t2
p1        p2        p3        p4
 1         1         0         2
Node type: ' '   Parent state: 2

state:5    Firing event: t3
p1        p2        p3        p4
 0         1         0         0
Node type: 'T'   Parent state: 2

state:6    Firing event: t2
p1        p2        p3        p4
 0         2         1         2
Node type: ' '   Parent state: 3

state:7    Firing event: t1
p1        p2        p3        p4
 0         2         1         2
Node type: 'D'   Parent state: 4

state:8    Firing event: t2
p1        p2        p3        p4
 0         2         0         3
Node type: 'T'   Parent state: 6


Boundedness:
p1 : 2
p2 : 2
p3 : 2
p4 : 3

>>
```

The screen output given above is equivalent to the graphic shown in figure 15.

10.1.3 Event simulation instead of coverability tree

Lets try event simulation of the same Petri net.

```
% the main file to find the reachable states
```

```
clear, clc; % clear the workspace & screen first

png = petrinetgraph('cotree_example_def');
dyn.initial_markings = {'p1', 2, 'p4', 1}; % tokens initially
dyn.firing_times = {'t1',2, 't2',1, 't3',3}; % tokens initially

Results = gpensim(png, dyn);
print_statespace(Results);
```

The function **print_cotree** will print the state flow on the screen:

```
COTREE Example: Petri Net in Figure 15

Time: 0
New markings:
p1        p2        p3        p4
 2         0         0         1

At time: 0  enabled transtions are:  t1

At time: 0  firing transtions are:  t1

Time: 2
Fired Transition: t1
New markings:
p1        p2        p3        p4
 1         1         1         1

At time: 2  enabled transtions are: t1 t2 t3

At time: 2  firing transtions are:  t1 t2

Time: 3
Fired Transition: t2
New markings:
p1        p2        p3        p4
 0         1         0         2

At time: 3  enabled transtions are:

At time: 3  firing transtions are:  t1

Time: 4
Fired Transition: t1
New markings:
p1        p2        p3        p4
 0         2         1         2

At time: 4  enabled transtions are:  t2

At time: 4  firing transtions are:  t2

Time: 5
Fired Transition: t2
New markings:
p1        p2        p3        p4
 0         2         0         3
```

```
At time: 5  enabled transtions are:

At time: 5  firing transtions are:
```

10.2 Example-10: Cotree with infinite states

This simple example deals with the Petri net shown in figure 16. The co-tree of this Petri net is shown in figure 17. Let us find the co-tree using GPenSIM:

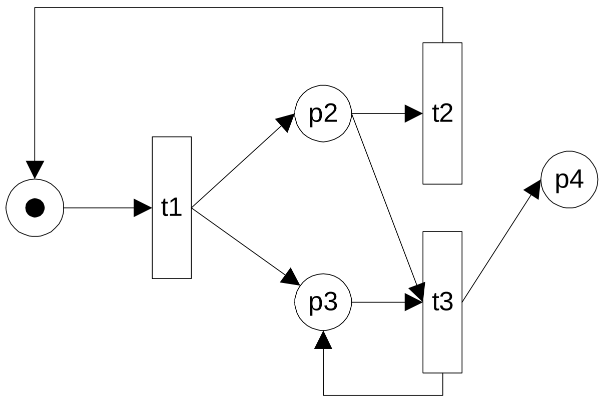


Figure-16.   Cotree example

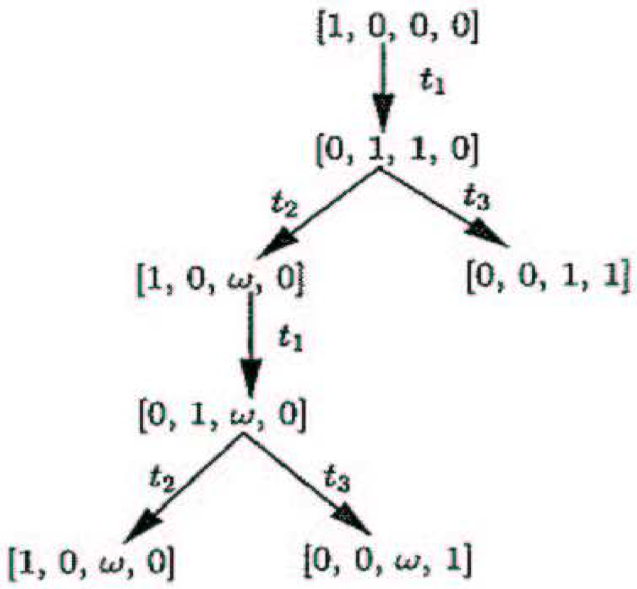


Figure-17.   Co-tree

10.2.1 Petri net definition file

The Petri net definition file is given below:

```
% PDF Example-10: Cotree example-2
% file:
function [PN_name, set_of_places, set_of_trans, set_of_arcs]...
    = fig_9_def()

PN_name = 'Petri net in fig 4.12';
set_of_places = {'p1', 'p2', 'p3', 'p4'};
set_of_trans = {'t1','t2', 't3'};
set_of_arcs =  {'p1', 't1', 1, 't1', 'p2', 1, 't1', 'p3', 1,...
    'p2','t2',1, 't2','p1',1, 'p2','t3',1 ...
    'p3','t3',1, 't3','p3',1, 't3','p4', 1};
```

10.2.2 The main file

The main file (after phases 2 & 3) is given below:

```
% Example-10: Cotree example-2
% the main file to get co-tree
clear, clc;
png = petrinetgraph('fig_9_def');
dyn.initial_markings = {'p1',1};

CT = cotree(png, dyn);
print_cotree(CT); %
```

The print system will print the following on the screen, which is equivalent to the graphical co-tree shown in figure 17.

```

Petri net in fig 4.12'

state:1  ROOT node
p1        p2        p3        p4
 1         0         0         0

state:2    Firing event: t1
p1        p2        p3        p4
 0         1         1         0
Node type: ' '   Parent state: 1

state:3    Firing event: t2
p1        p2        p3        p4
 1         0         Inf       0
Node type: ' '   Parent state: 2

state:4    Firing event: t3
p1        p2        p3        p4
 0         0         1         1
Node type: 'T'   Parent state: 2

state:5    Firing event: t1
p1        p2        p3        p4
 0         1         Inf       0
Node type: ' '   Parent state: 3
```

```
state:6    Firing event: t2
p1        p2        p3        p4
 1         0         Inf       0
Node type: 'D'   Parent state: 5

state:7    Firing event: t3
p1        p2        p3        p4
 0         0         Inf       1
Node type: 'T'   Parent state: 5


Boundedness:
p1 : 1
p2 : 1
p3 : Inf
p4 : 1
```

11. Global Info

Global variables and parameters can be passed through different files (e.g. SMU, PDFs, and TDFs) by making use of the 'global info' packet. The methodology of using 'global info' is explained below through the use of an example.

11.1 Use of 'MAX_LOOP'

'MAX_LOOP' value, if added to the '**global_info**' packet, will be read by the gpensim function to limit the simulation cycles to the given value.

NOTE:
Increase MAX_LOOP for large number of iterations (loops)

11.1.1 Example-11: MAX_LOOP

This is same as the example-06. This time, we will experiment with global MAX_LOOP setting.

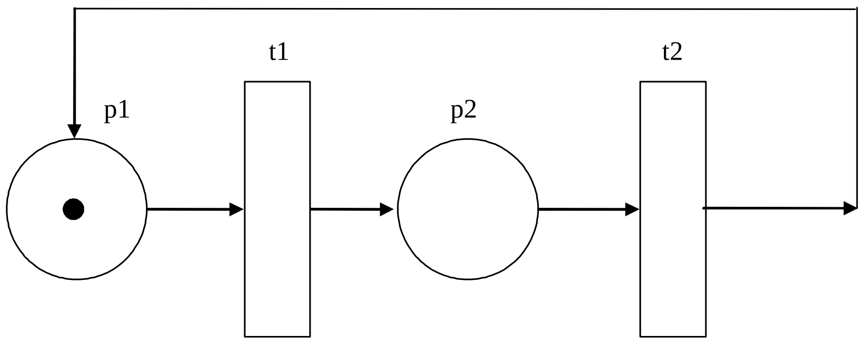


Figure-18. Transitions firing sequentially

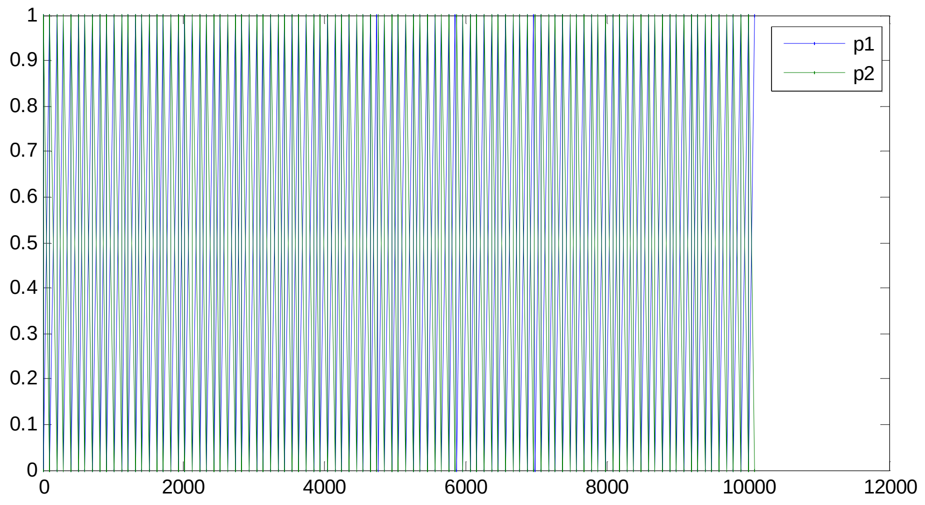
The Petri net shown in figure 18 run for ever. Thus, unless specified in the SMU, default maximum loop number is 200 (default MAX_LOOP=200). We can stop the simulations after a couple of simulation cycles. The statement given below limits the simulation cycles to 11, by assigning the value 11 to 'MAX_LOOP':

```
> global_info.MAX_LOOP = 11; % GLOBAL DATA: MAX. SIMULATION CYCLES
```

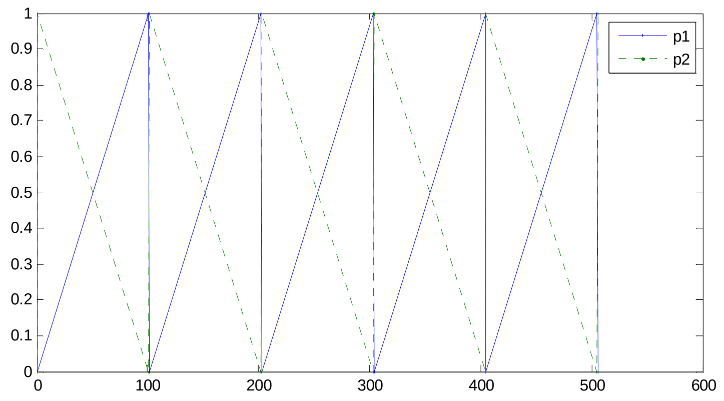
The code below is for the main simulation file.

```
% Example-11: Measuring Timing
% MSF: measure_timing.m
clear, clc;
global_info.MAX_LOOP = 11; % GLOBAL DATA: MAX. SIMULATION CYCLES

png = petrinetgraph('measure_timing_def');
dynamicpart.initial_markings = {'p1', 10};
dynamicpart.firing_times = {'t1', 1, 't2', 100};
sim = gpensim(png, dynamicpart, global_info);
plotp(sim, {'p1', 'p2'});
```

Simulation results: When MAX_LOOP is not explicitly specified (meaning by default, MAX_LOOP=200):



Simulation results: When MAX_LOOP is explicitly specified to be 11 (in SMU, MAX_LOOP=11):



11.2 Use of 'LOOP_NUMBER'

When you simulate large Petri net models, during the simulations you will notice that the MATLAB hangs, without giving you any sign of life. It will be better, if you can see some outputs during simulations so that you are assured that the simulations are going on and that the system is dead ('hanging'). By setting the **LOOP_NUMBER** flag in global_info, you can see the loop numbers when the simulation goes on.

Let us go back to the simple example given in section 3.2, the simple Petri net. This time, we will set the LOOP_NUMBER flag in the MSF:

```
%% LOOP_NUMBER flag is set in global_info
global_info.LOOP_NUMBER = 1;

png = petrinetgraph('simple_pn_def');
dynamic_info.initial_markings = {'Place-1',3, 'Place-2',5};
dynamic_info.firing_times = {'Transition-1', 10};

Sim_Results = gpensim(png, dynamic_info, global_iinfo);
print_statespace(Sim_Results);
```

The output on screen is different as loop numbers are printed during simulations. According to the screen output, the simulations are complete after 3 loops.

```
Loop nr:  1
Loop nr:  2
Loop nr:  3

A Simple Petri Net definition
Number of places: 3
Initial Markings:
Place-1   Place-2   Place-3
 3         5         0
step:1     Firing  event:  Transition-1       (Starting  time:  0)   Finishing
Time: 10
Current markings:
Place-1   Place-2   Place-3
 2         3         1
step:2     Firing  event:  Transition-1      (Starting  time:  10)   Finishing
Time: 20
Current markings:
Place-1   Place-2   Place-3
 1         1         2
Completion time: 20
```

NOTE:

It is always a good idea to set the LOOP_NUMBER flag (global_info.LOOP_NUMBER = 1) in the MSF. By setting the LOOP_NUMBER flag, simulation loop number will be displayed during the simulation, thus we know that simulation is going on and the computer is not 'hanging'.

11.2.1 What are loops?

(See chapter 19 "Design of GPenSIM" for more details)

OK, we do see loop numbers during simulations, a kind of assurance that something is going on. But what are loops? To understand loops, we need to understand the theory for general discrete event simulations (DES).

Any DES software consists of three main elements:

1. *Global timer:* Global timer (or current time) synchronizes all the activities. Global timer must not be changed by any transitions (events). In GPenSIM, global timer can be accessed in TDFs, by calling **pn.current_time**, where **pn** is the run-time Petri net structure.
2. *Event Scheduler:* Event scheduler is a **loop** mainly performing two actions:
    a. First: checking for any enabled transitions; if there are any enabled transition and if they can fire, then they will be put in *queue* called firing transitions (implemented in file **start_firing.m**).
    b. Second: checking the queue for firing transitions. When a firing transition is complete, it will be removed from the queue (implemented in file **complete_firing.m**)

   In GPenSIM, file **timed_pensim.m** implements event scheduler.
3. *Queue*: (discussed above)

Thus, loop number comes from **timed_pensim** which is called by **gpensim**. The loop number states how many cycles of event scheduler has taken place so far.
NOTE: Chapter 16 “Design of GPenSIM” gives more details

11.3 Use of ‘DELTA_TIME’

Section 6 “Internal Clock” describes an example (example-04: delay) in which there are enabled transitions but not firing (blocked). This is situation, the clock is advanced by a time interval equal to one-fourth of the minimal firing time of any transition. We can override this value for timer advancement, by assigning a new value to “DELTA_TIME”.

Lets repeat the example-04. We will study three cases this time:

1. DELTA_TIME is not explicitly specified (by default, delta_time equals to ¼ of least firing time)
2. DELTA_TIME = 5
3. DELTA_TIME = 0.1

11.3.1 Example-12: DELTA_TIME

This example is the same as example-04. But this time, we will experiment setting DELTA_TIME. In the figure shown below, let **p1** has 5 initial tokens. Also let firing time of **t1** is 7 seconds. Though **t1** can fire 5 times successively, we want it to fire only at the start of every 30 seconds. This means, **t1** is delayed by 30 - 7 = 23 seconds.

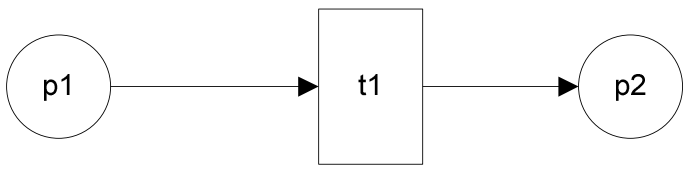


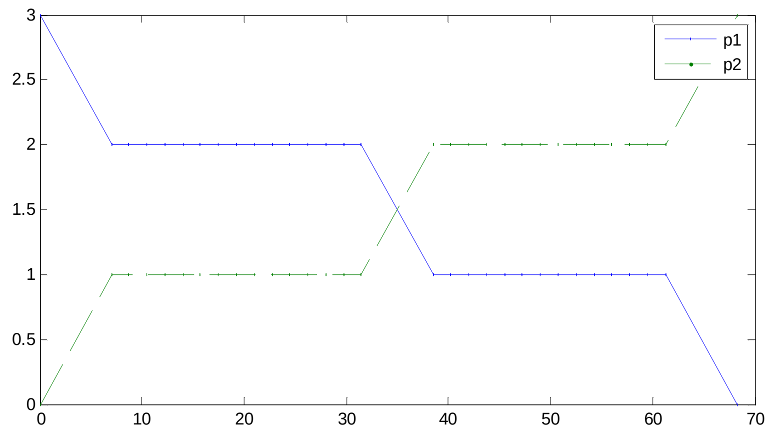
Figure-19. Delay Example

During the waiting time of 23 seconds (**t1** is enabled but not firing), time advancement will be done in time units of 7/4 = 1.75 seconds, if DELTA_TIME is not explicitly specified.

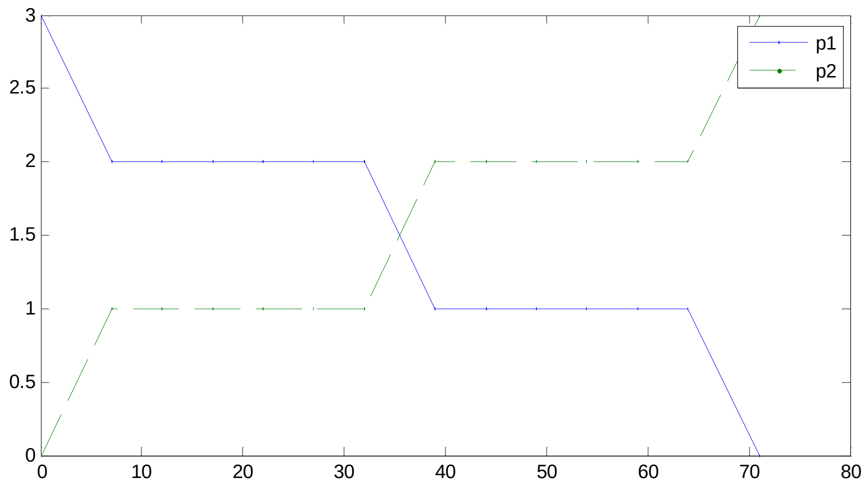
MSF:

```
% Example-12: DELTA_TIME
% file: delay_demo.m:
global_info.MAX_LOOP = 1000;
global_info.DELTA_TIME = 0.1;

png = petrinetgraph('delay_demo_def');

dynamic.initial_markings = {'p1',3};
dynamic.firing_times = {'t1', 7};

sim = gpensim(png, dynamic, global_info);
print_statespace(sim);
plotp(sim, {'p1','p2'});
```

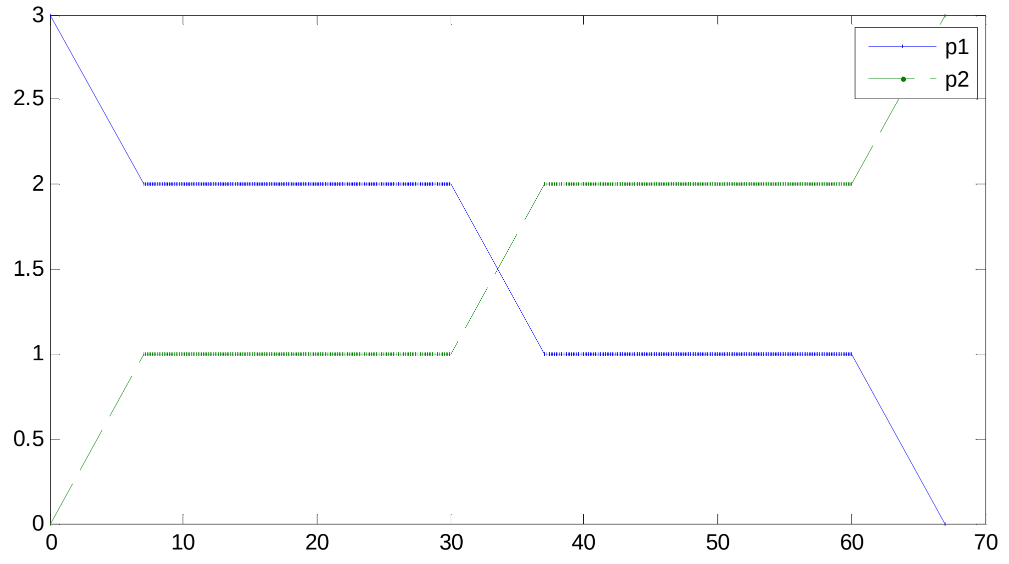
Simulation results: When DELTA_TIME is not explicitly specified (meaning by default, DELTA_TIME =1.75):



Simulation results: When DELTA_TIME is explicitly specified to be 5.0:



Simulation results: When DELTA_TIME is explicitly specified to be 0.1:



12. TDF_POST

As stated in the earlier sections, there are two types of Transition Definition Files (TDF):

- TDF_PRE, which are run before firing a transition
- TDF_POST, which are run after firing a transition

12.1 Example-13: Binary Semaphore

Figure 20 shown below depicts a web server consisting of two server machines that will fire alternatively. First, client requests are queued at **pSTART**. Then two routers (**tX1** and **tX2**) remove the client requests from the **pSTART** queue and put it to the queues for Web Server 1 (**p1**) and Web Server 2 (**p2**) respectively. In order to evenly distribute client requests to both servers, one would expect that the two routers fire alternatively, meaning that no router fires more times than the other.

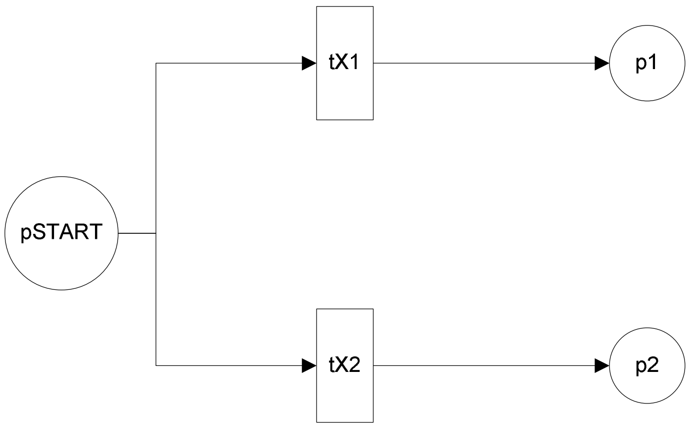


Figure-20. Load balancing by alternative firing

To allow the routers (transitions) fire alternatively, we can implement a binary semaphore that can be read and manipulated by the definition files of both transitions.

12.1.1 Petri net definition file ('loadbalance_def.m'):

```
% PDF for Example-13: Binary Semaphore example
% file: loadbalance_def.m:
% definition of petri net graph for Norwegian trafic lights

function [PN_name, set_of_places, set_of_trans, set_of_arcs]...
                    = loadbalance_def(global_info)
PN_name='Web Server Load Balancer';

set_of_places={'pSTART', 'p1', 'p2'};
set_of_trans={'tX1','tX2'};

set_of_arcs={'pSTART','tX1',1, 'tX1','p1',1,...
             'pSTART','tX2',1, 'tX2','p2',1};
```

12.1.2 Main Simulation File (‘loadbalance.m’):

```
% Example-13: Example for binary semaphore
% MSF: loadbalance.m

clear, clc;
global_info.semafor = 1;    % GLOBAL DATA: binary semafor

png = petrinetgraph('loadbalance_def');
dynamicpart.initial_markings = {'pSTART', 10};
dynamicpart.firing_times = {'tX1', 10, 'tX2', 20};

sim = gpensim(png, dynamicpart, global_info);
plotp(sim, {'p1', 'p2'});
```

Note: gpensim takes three input parameters: in addition to the usual static (‘png’) and dynamic (‘dynampart’) details, the third parameter is the global info (‘global_info’). Global info consists of two elements:

1) The binary semaphore with initial value 1; this means, tX1 should fire first.
2) MAX_LOOP: the use of this value is explained in the previous sections

12.1.3 TDF_PRE for tX1 (‘tX1_pre.m’):

```
function [fire, PN,new_color, override, selected_tokens, global_info] = ...
    tX1_pre(PN, new_color, override, selected_tokens, global_info)
%
%

if (global_info.semafor==1),
    fire = 1;
else
    fire = 0;
end;
```

12.1.4 TDF_POST for tX1 (‘tX1_post.m’):

```
function [PN, global_info] = ...
    tX1_post(transition, PN, global_info)
% function tX1_post
%

global_info.semafor = 2; % release semafor to tX2
```

12.1.5 TDF_PRE for tX2 (‘tX2_pre.m’):

```
function [fire, PN,new_color, override, selected_tokens, global_info] = ...
    tX2_pre(PN, new_color, override, selected_tokens, global_info)
% TDF tX2_pre
%


if (global_info.semafor==2),
    fire = 1;
```

```
else
    fire = 0;
end;
```

12.1.6 TDF_POST for tX2 ('tX2_post.m'):

```
function [PN, global_info] = ...
    tX2_post(transition, PN, global_info)
% function tX2_post
%

global_info.semafor = 1; % release semafor to tX1
```

The plot given below shows that the queues are filled evenly; this is because of the transitions fires alternatively.

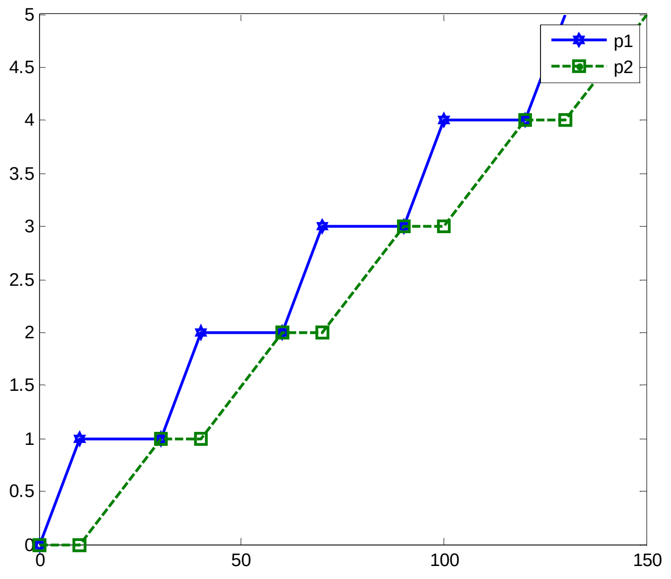


Figure-21. Printout of binary semaphore in action

13. Improving Simulation Results for Printout

Let's take look again at the printout of simulation results from the previous section. The figure, given below, look like ramp rather than pulses. This is due to poor sampling (recording). Simulation results are recorded only whenever transition complete firing. In other words, simulation results are recorded only when there is a new state.

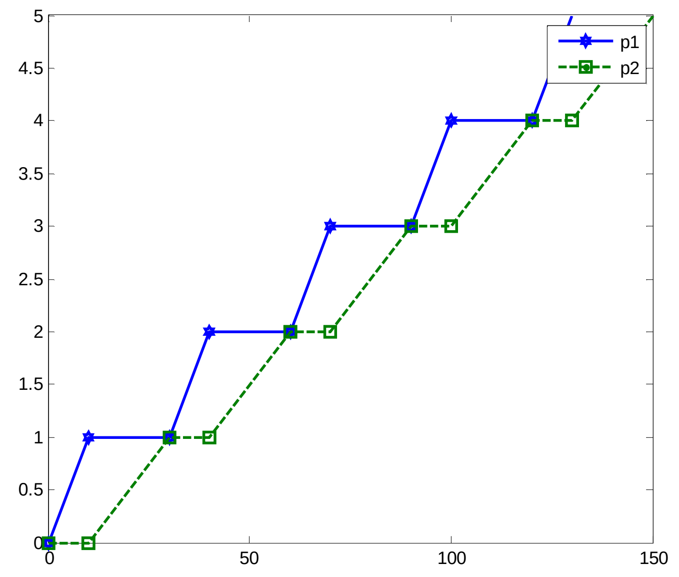


Figure-22. Printout of binary semaphore (same as figure-21)

We can improve sampling by adding a small loop that will generate new states faster. Example-14 given below explains the trick.

13.1 Example-14: Improving results printout of binary semaphore

In this example, we will add a small loop to the system; the small loop consisting of a place **pXtra** and a transition **tXtra** is solely included to speed up the sampling rate (or rate of reaching newer states). The firing time of the transition **tXtra** has to be small, lets say – one tenth of the least firing time of any transition in the system (**tX1** or **tX2**). Note: Do not assign zero value firing time of the transition tXtra; with zero value, the system will never take off.

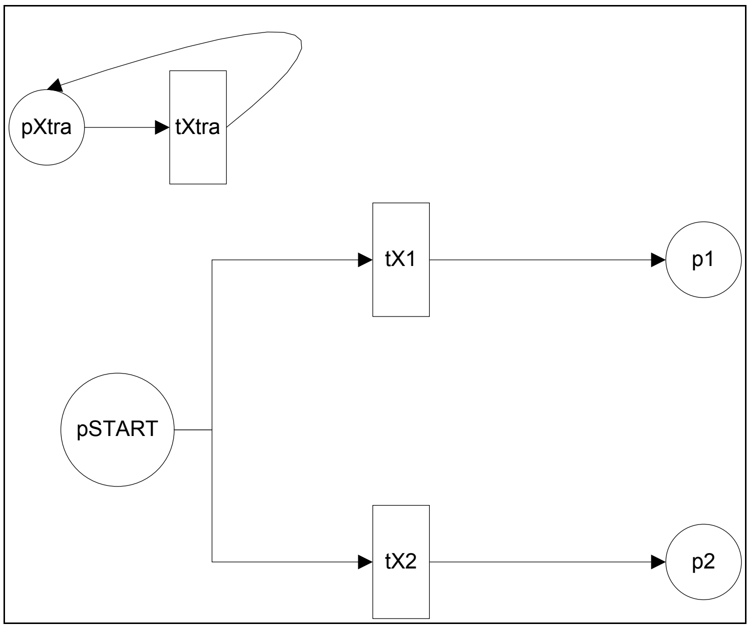


Figure-23. Adding a small loop to speed up sampling rate

Except adding the small loop (pXtra – tXtra – pXtra), there is no change in coding for example-13.

MSF:

```
% Example-14: Example for binary semaphore
% MSF: loadbalance_2.m

clear, clc;
global_info.semafor = 1;    % GLOBAL DATA: binary semafor

png = petrinetgraph('loadbalance_2_def');
dynamicpart.initial_markings = {'pSTART', 10, 'pXtra',1}; % pXtra added
dynamicpart.firing_times = {'tX1', 10, 'tX2', 20, 'tXtra',1}; % tXtra added

sim = gpensim(png, dynamicpart, global_info);
plotp(sim, {'p1', 'p2'});
print_statespace(sim);
```

PDF:

```
% Example-14: Binary semaphore example with better rpintout
% file: loadbalance_2_def.m: PDF

function [PN_name, set_of_places, set_of_trans, set_of_arcs]...
                    = loadbalance_2_def(global_info)
 PN_name='Web Server Load Balancer';

set_of_places = {'pSTART', 'p1', 'p2', 'pCK'};
set_of_trans = {'tX1','tX2', 'tCK'};
set_of_arcs = {'pSTART','tX1',1, 'tX1','p1',1,...
             'pSTART','tX2',1, 'tX2','p2',1,...
             'pCK','tCK',1, 'tCK','pCK',1};
```

Simulation Results:
Figure-24 shows the new simulation results after inclusion of the small loop; new simulation results and its printout is due to faster sampling.

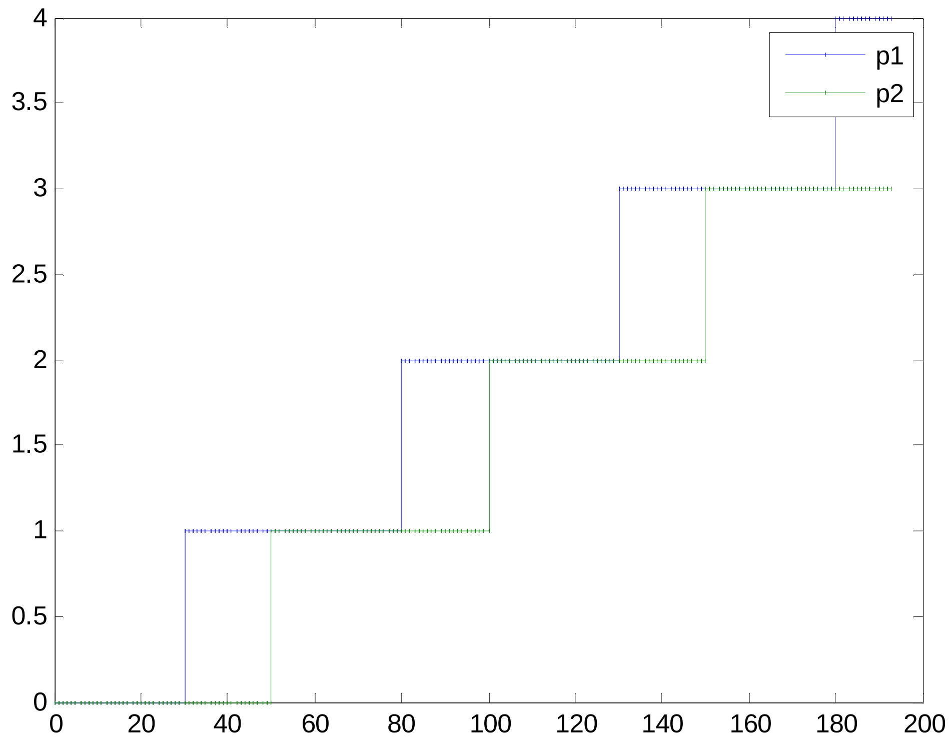


Figure-24. Improved printout due to faster sampling

14. Prioritizing Transitions

In discrete systems, we need to increase or decrease priority of an event(s), in order to give fair chance to the competing events. There are some basic facilities in GPenSIM to change priorities of transitions.

1) Initial declaration of priorities in the main simulation file.
2) Increasing priority of a specific transition
3) Decreasing priority of a specific transition

14.1 Priorities of transitions

Initial declaration of priorities in the main simulation file can be done using the global_info.

```
global_info.PRIORITY = {'t1', 5, 't2',2, 't3', 10};
```

In the above line, we are simply saying that **t3** has top priority, followed by **t2** and **t1** has the least priority. When we assign priority, we can assign any integer value, both negative and positive. Higher the value, better the priority is.

Increasing priority of a specific transition can be done using the function '**priority_increment**', which will increase the value just by 1.

```
PN = priority_increment(PN, 't1'); % priority of 't1' is now 6
```

Decreasing priority of a specific transition can be done using the function '**priority_decrement**', which will reduce the value by 1.

```
PN = priority_decrement(PN, 't3'); % priority of 't2' is now 9
```

14.2 Example-15: Alternating firing

Transitions t1, t2, and t3, should fire alternatively (figure 25).

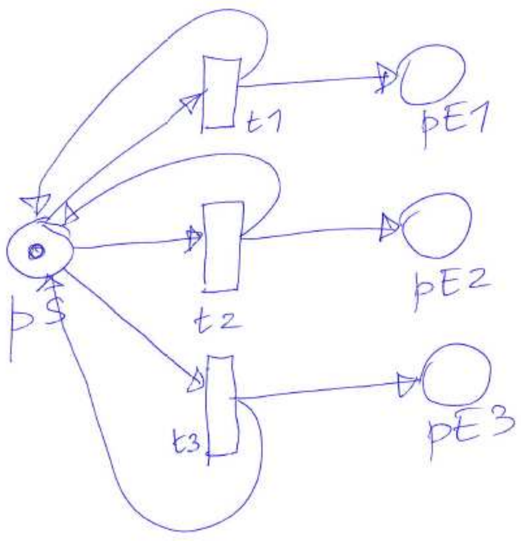


Figure-25. Alternating firing of t1, t2, and t3

MSF:

```
% Example-15: Priority Increment example
global_info.MAX_LOOP = 20;

png = petrinetgraph('prio_def');
dyn.initial_markings = {'pS', 1}; % tokens initially
dyn.firing_times = {'t1',1, 't2',1, 't3',1};

sim = gpensim(png, dyn, global_info);
plotp(sim, {'pE1', 'pE2', 'pE3'});
```

PDF:

```
% Example-15: Priority Increment example
% file: prio_def.m: definition of petri net

function [PN_name, set_of_places, set_of_trans,...
    set_of_arcs] = prio_def()

PN_name='Priority Example: Petri Net for production facility';
set_of_places={'pS', 'pE1', 'pE2', 'pE3'};

set_of_trans={'t1','t2','t3'};

set_of_arcs={'pS','t1',1, 'pS','t2',1, 'pS','t3',1,...
    't1','pE1',1, 't1','pS',1, ...
    't2','pE2',1, 't2','pS',1, ...
    't3','pE3',1, 't3','pS',1};
```

TDF_PRE for t1 ('t1_pre.m'):

```
function [fire, PN,new_color,override,selected_tokens,global_info] = ...
    t1_pre(PN, new_color,override,selected_tokens,global_info)
%
% t1_pre

PN = priority_increment(PN, 't2');
fire = 1;
```

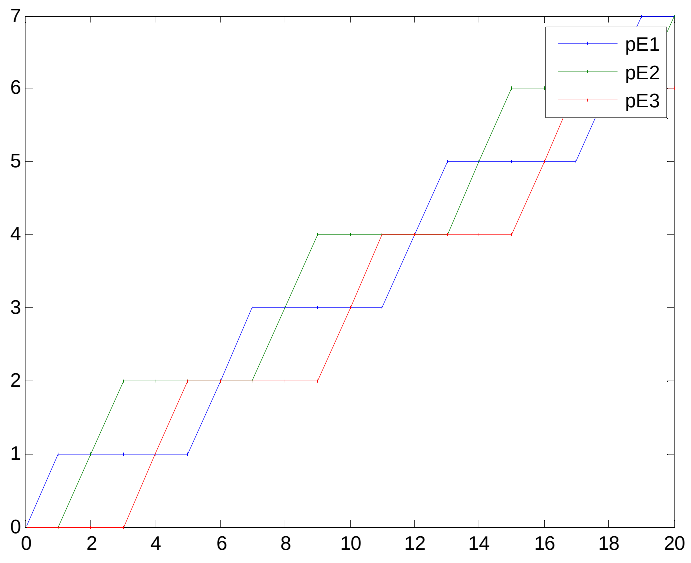
TDF_PRE for t2 ('t2_pre.m'):

```
function [fire, PN,new_color,override,selected_tokens,global_info] = ...
    t2_pre(PN, new_color,override,selected_tokens,global_info)
%
% t2_pre

PN = priority_increment(PN, 't3');
fire = 1;
```

TDF_PRE for t3 (‘t3_pre.m’):

```
function [fire, PN,new_color,override,selected_tokens,global_info] = ...
    t3_pre(PN, new_color,override,selected_tokens,global_info)
%
% t3_pre

PN = priority_increment(PN, 't1');
fire = 1;
```

Simulation Results:
The results show that the mechanism is little bit flawed, and need to be checked.



14.3 Example-16: Priority Decrement Example

This example is the same as for the previous example shown in figure 25. However, this time, we will allow t1 to fire 5 times uninterrupted, and then allow t1 and t2 fire alternatively for 10 more times. After this, all three can fire alternatively.

SMU:

```
% Example-16: Priority decrement
global_info.MAX_LOOP = 25;
global_info.PRIORITY = {'t1',10, 't2',5};

png = petrinetgraph('prio_def');
dyn.initial_markings = {'pS', 1}; % tokens initially
dyn.firing_times = {'t1',1, 't2',1, 't3',1};

sim = gpensim(png, dyn, global_info);
plotp(sim, {'pE1', 'pE2', 'pE3'});
```

PDF:

```
% Example-16: Priority Decrement
% file: prio_def.m: definition of petri net

function [PN_name, set_of_places, set_of_trans,...
    set_of_arcs] = prio_def()

PN_name='Priority Example: Petri Net for production facility';
set_of_places={'pS', 'pE1', 'pE2', 'pE3'};

set_of_trans={'t1','t2','t3'};

set_of_arcs={'pS','t1',1, 'pS','t2',1, 'pS','t3',1,...
    't1','pE1',1, 't1','pS',1, ...
    't2','pE2',1, 't2','pS',1, ...
    't3','pE3',1, 't3','pS',1};
```

TDF_PRE for t1 (‘t1_pre.m’):

```
function [fire, PN,new_color,override,selected_tokens,global_info] = ...
    t1_pre(PN, new_color,override,selected_tokens,global_info)
%
% t1_pre

PN = priority_decrement(PN, 't1');
fire = 1;
```

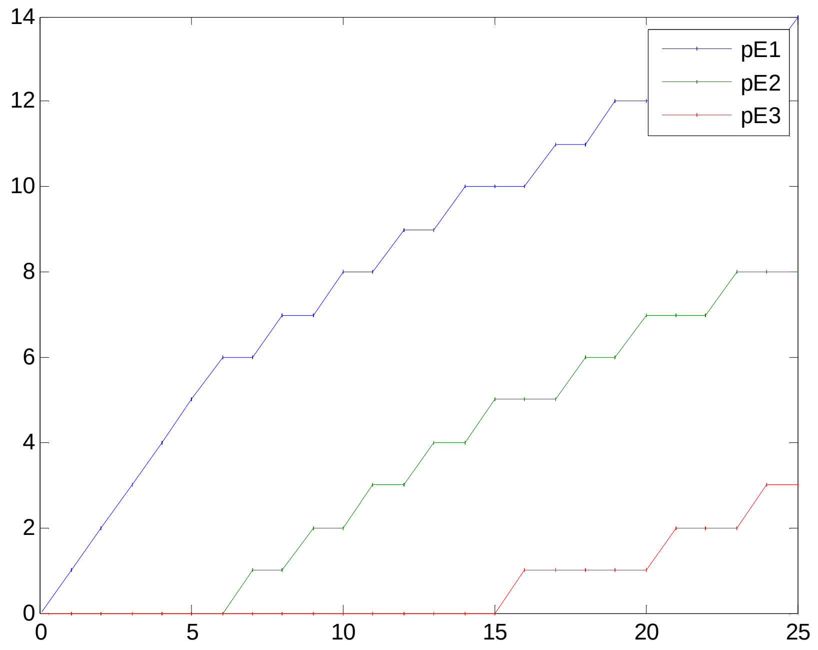
TDF_PRE for t2 (‘t2_pre.m’):

```
function [fire, PN,new_color,override,selected_tokens,global_info] = ...
    t2_pre(PN, new_color,override,selected_tokens,global_info)
%
% t2_pre

PN = priority_decrement(PN, 't2');
fire = 1;
```

TDF_PRE for t3 (‘t3_pre.m’):

```
function [fire, PN,new_color,override,selected_tokens,global_info] = ...
    t3_pre(PN, new_color,override,selected_tokens,global_info)
%
% t3_pre

PN = priority_decrement(PN, 't3');
fire = 1;
```

Simulation Results: Again, not perfect!!!


pE1
pE2
pE3
0
2
4
6
8
10
12
14
0
5
10
15
20
25

15. Using Resources

In engineering systems, there are always resources, like human resources to operate some machines, printers as common resources in a network, etc. Just like machines and robots, resources can also be represented with transitions (or places, depending on the situation). However, GPenSIM offers ‘global resources’ as a mechanism to simply the models, also provided is a print function called ‘print_schedule’ to print the usage of the resources.

Given below is a simple example that explains the usage of resources. An larger example on scheduling is given in the applications part.

15.1 Using Resources

The resources are to be declared first in the MSF. For example, if there three (human) resources named Al, Bob, and Chuck, then the following declaration will be added to the MSF:

```
dynamicpart.resources = {'Al', 'Bob', 'Chuck'};
```

Reserving a resource can be done through the function ‘resource_request’. For example:

```
[acquired, PN] = resource_reuqest(PN, 'T1'); % seek any resource

% seek specific resources, both ’Al’ and ’Bob’
[acquired, PN] = resource_request(PN, 'T1', {'Al', 'Bob'});
```

In the first case, transition ’T1’ seeks (reserves) one instance of a resource (any resource). If allocation was successful, the flag ‘acquired’ will be true. In the second case, ‘T1’ seeks two resources, but specific resources like ‘Al’ and ‘Bob’, this time.

Releasing the resources: a transition has to release all the resources it is holding, releasing some or specific resources is not possible.

```
% release all the resources (if any) held by ’T1’
[released, PN] = resource_ release(PN, 'T1');
```

15.1.1 Function ‘print_schedule’

```
% function print_schedule(sim_results)
% For every resource utilized, this function prints
% a matrix where each row represents:
%   [the transition that used the resource, start time, end time]
%
% In addition the following are also displayed:
%       K, ST, LE, SI, and LT
%
```

15.2 Example-17: Using Resources to realize critical section

This example is the same as the one that is described under the section “Global Info”; however, we make use of ‘resources’ rather than ‘global info’.

Figure 26 shown below depicts a web server consisting of two server machines that will fire alternatively. First, client requests are queued at **pSTART**. Then two routers (**tX1** and **tX2**) remove the client requests from the **pSTART** queue and put it to the queues for Web Server 1 (**p1**) and Web Server 2 (**p2**) respectively. In order to evenly distribute client requests to both servers, one would expect that the two routers fire alternatively, meaning that no router fires more times than the other.

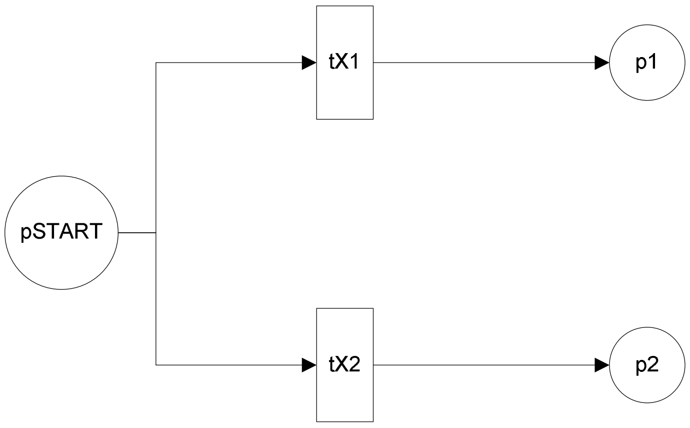


Figure-26. Load balancing by alternative firing

To allow the routers (transitions) fire alternatively, these two transition seek a semafor (resource). If a transition does not get the semafor, its priority is increased so that next time it will get it.

15.2.1 MSF: ’cr.m’

```
% Example-17: use of resource for realizing critical function
png = petrinetgraph('cr_def');
dynamicpart.initial_markings = {'pSTART', 20};
dynamicpart.firing_times = {'tX1', 10, 'tX2', 20};
dynamicpart.resources = {'semafor'}; % resource as semafor

sim = gpensim(png, dynamicpart);

plotp(sim, {'p1', 'p2'}), grid on;
print_schedule(sim);
```

15.2.2 PDF: ’cr_def.m’

```
% Example-72: Binary semaphore example
% file: cr_def.m: PDF

function [PN_name, set_of_places, set_of_trans, set_of_arcs]...
                    = cr_def(global_info)
```

```
PN_name='Implementing Critical region with resources';

set_of_places={'pSTART', 'p1', 'p2'};
set_of_trans={'tX1','tX2'};

set_of_arcs={'pSTART','tX1',1, 'tX1','p1',1,...
             'pSTART','tX2',1, 'tX2','p2',1};
```

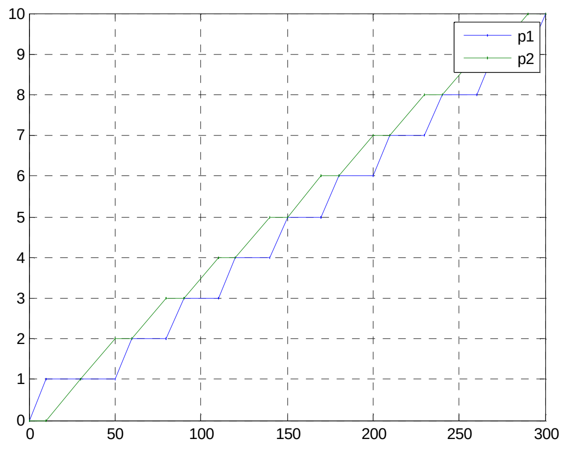
15.2.3 TDF: ’tX1_pre.m’

```
function [fire, PN,new_color, override, selected_tokens,
global_info] = ...
    tX1_pre(PN, new_color, override, selected_tokens, global_info)
% tX1_pre
%

[acquired, PN] = acquire_resource(PN, 'tX1');

if ~acquired,   % if not suceeded
    PN = priority_increment(PN, 'tX1'); % increase trans priority
end;

fire = acquired;
```

15.2.4 TDF: ’tX1_post.m’

```
function [PN,global_info] = tX1_post(transition, PN, global_info)
% tX1_post
%
[released, PN] = release_resource(PN, 'tX1'); % release semafor
```

15.2.5 Results: Plot



15.3 Example-18: Using Resource Specific

16. Using Hourly Clock

So far, we have treated clock as a unitless timer; it will always start at 0 during simulation start, and will increase afterwards. However, in business modeling applications, it will be much better to use an hourly clock, a clock that uses and shows time in hours, minutes, and seconds. The following example explains the issue.

CAUTION! CAUTION!
Time in hourly format must be given as a vector with 3 columns (e.g. 1:00 PM as [13, 0, 0]); you can mix times in 3 column hourly format with single numbers; however, these single numbers will be taken as seconds.

E.g.:

[0 40 0]	**is equivalent to 40 minutes (or 2400 seconds)**
'unifrnd(40,40)*60'	**is equivalent to 2400 seconds (40*60)**
180	**is equivalent to 180 seconds**

16.1 Example-19: Hourly Clock for Lunching Clerks

An office opens at 09:00 AM on every business day. Customers arrive at every 30 minutes. There are two clerks who will interact with the customers. The clerks take 40 minutes to service a customer.
The office closes at 01:00 PM, and no customer will be allowed into the office. However, those customer(s) already reside inside the office will be serviced.

1. Case-A: What time the last customer will leave the office, after finishing his/her business?
2. Case-B: Suppose, there will only one clerk available from 12:00 Noon, how the departure time of the last customer will change?

16.1.1 Functions for hourly clock

First of all, we want to start the simulation at 09:00 AM. This can be fed into the model through the global_info packet.

```
global_info.STARTING_AT = [9 0 0]; % start 09:00:00 HH:MM:SS
```

In MSF, to assign firing times to clerk (40 minutes each), and customer arrival (every 30 minutes), we may either use the hourly clock format or times in seconds:

```
dyn.firing_times = {'tGENNEW', 30*60, 'tCRK1', 'unifrnd(40,40)*60',...
    'tCRK2', [0 40 0]};
```

Note: Because of the use of hourly clock formats, the functions **print_statespace** and **plotp** display time information in hourly formats.

16.2 Case-A: Two clerks work all the time

MSF:

```
% Example-31: Hourly clock for lunching clerks

clear; clc;
global_info.LOOP_NUMBER = 1;
global_info.MAX_LOOP = 50;

global_info.STARTING_AT = [9 0 0]; % start 09:00:00 HH:MM:SS

%%%% COMPOSE %%%%%%%
png = petrinetgraph('clerksNEW_def');

%%%% DYNAMIC DETAILS %%%%
dyn.initial_markings = {'pGEN',1, 'pQUE',1};
dyn.firing_times = {'tGENNEW',30*60, 'tCRK1', 'unifrnd(40,40)*60',...
    'tCRK2', [0 40 0]};

%%%% SIMULATE %%%%%
[RES] = gpensim(png, dyn, global_info);
plotp(RES,  {'pEND'}), grid on;
print_statespace(RES);
```

PDF:

```
% Example-31: Hourly clock for lunching clerks
% PDF
function [PN_name, set_of_places, set_of_trans, set_of_arcs] ...
    = clerksNEW_def(global_info)

PN_name = 'The Two Clerks';
set_of_places = {'pGEN', 'pQUE', 'pEND'};
set_of_trans={'tGENNEW', 'tCRK1', 'tCRK2'};

set_of_arcs={'pGEN','tGENNEW',1 ,'tGENNEW','pGEN',1, ...
      'tGENNEW','pQUE',1, ...
      'pQUE','tCRK1',1,'tCRK1','pEND',1,...
      'pQUE','tCRK2',1,'tCRK2','pEND',1};
```

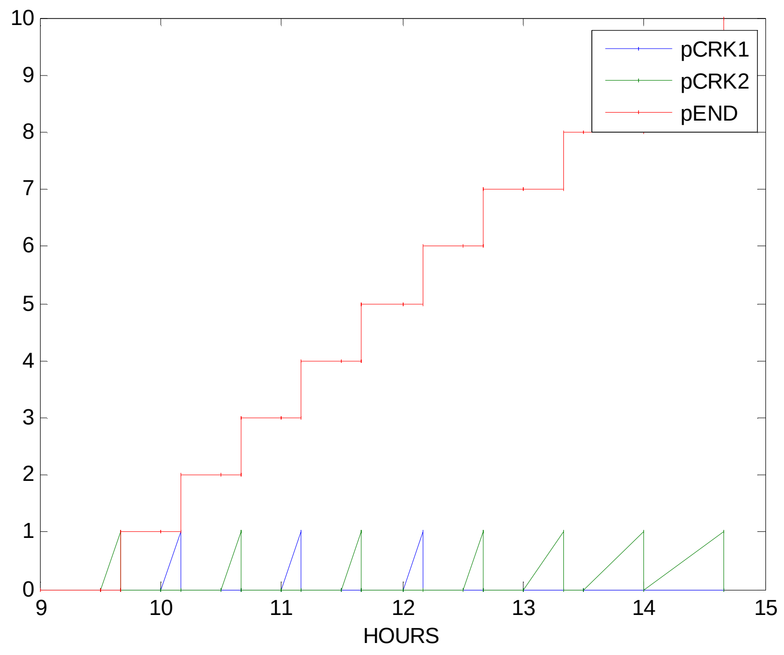
TDF for customer arrival:

```
% Example-31: Hourly clock for lunching clerks
% TDF for customer arrival generation

function [fire,new_color,override, selected_tokens,global_info] = ...
    tGENNEW_def (PN, new_color, override, selected_tokens, global_info)

ct = compare_time (PN.current_time, [13 0 0]);
if le(ct, 0),
    fire = 1;
else
    fire=0;
end;
```

16.2.1 Simulation results

Simulation results show that the last customer leaves at **14:10** when **both clerks function all the time**.

```
    Time: 14:10:00
State: 19
Fired Transition: tCRK1
Current State:
pEND      pGEN      pQUE
 10         1         0
```



16.3 Case-B: Only one clerk functions from 12:00 Noon

The only change will be the introduction of TDF for one of the clearks.

TDF for clerk-1 (‘tCLR1_def.m’):

```
% Example-31: Hourly clock for lunching clerks
% TDF for clerk-1

function [fire,new_color,override, selected_tokens,global_info] = ...
    tCRK1_def (PN, new_color, override, selected_tokens, global_info)

ct = compare_time (PN.current_time, [12 0 0]);
if lt(ct, 0),
    fire = 1;
else
```

```
    fire=0;
end;
```

16.3.1 Simulation results

Simulation results show that the last customer leaves at **14:40** when **only one clerk functions after 12:00 Noon**.

```
   Time: 14:40:00
State: 19
Fired Transition: tCRK2
Current State:
pEND      pGEN      pQUE
 10        1         0
```

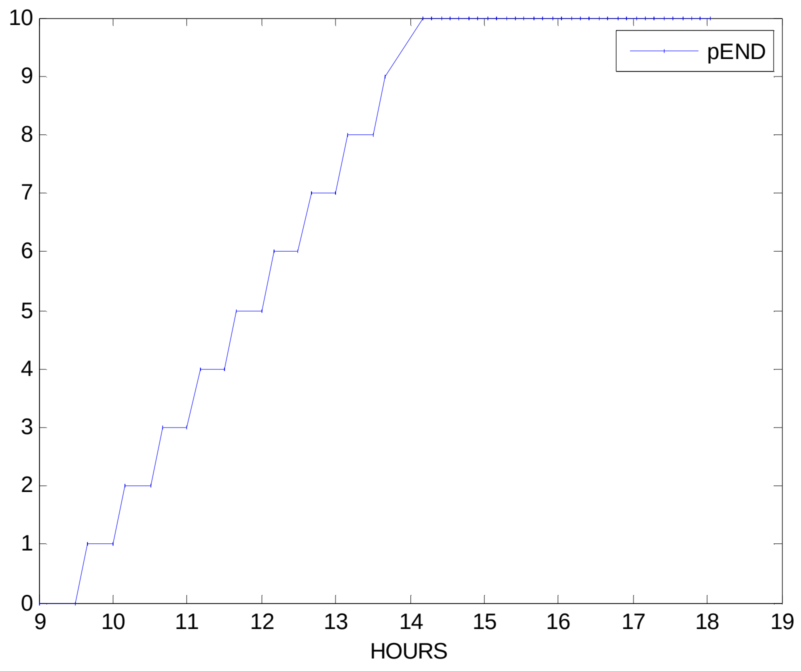


Figure-27. Plot showing time in hourly format.

17. Hybrid Systems: Petri Net Models with Fuzzy Inference

This section talks about incorporating MATLAB toolboxes within Petri net models. This section presents an example on how to incorporate fuzzy inference engines in Petri net models.

18. Colored GPenSIM

So far, we have treated tokens in place as indistinguishable. All the tokens inside a place are the same; it does not matter which token arrived into the place first or last. It does not matter either whether a token is deposited into a place by one transition or other. But, all these are going to be changed: from now on, every token is unique, identifiable with a unique token ID.

When using colors in GPenSIM, the following issues are important:

1. Only transitions can manipulate colors (see section 12)
2. Colors are inherited by default: that is when a token fires, it collects all the colors from the consumed (input) tokens and then it passes these to the deposited (output) tokens. However, color inheritance can be prevented by overriding (see section 12).
3. An enabled transition can select specific input tokens based on preferred colors (see section 13).
4. An enabled transition can select specific input tokens based on the time tokens are created (see section 14).
5. Structure of tokens; this is discussed in the following subsection

18.1 Structure of Tokens

A token has a structure consisting of 3 elements:

1. **tokID** (integer value): a unique token ID
2. **creation_time** (integer value): the time the token was created by a transition. Please note that this time may be different from (less than) the time the token was actually deposited into a place.
3. **t_color** (set of strings): a set of colors

E.g.:

```
        tokID: 101
creation_time: 30
      t_color: {'TAMIL', 'NORSK', 'ENGLISH'}
```

19. Color Inheritance

In GPenSIM, colored tokens can only utilized by transitions; since transitions are active, transition definition files can be coded with controlling colored tokens:

1. When a transition fire, **it inherits colors of all input tokens**; thus new tokens deposited into output places would have all the colors inherited from the input tokens. **NOTE: inheritance of colors can be prohibited by overriding**.
2. When a transition fires, **it can choose input tokens with specific colors**
3. When new tokens are deposited into the output place, **new colors can be added by the transition**. This new color will in addition to the inherited colors (unless inheritance is overridden – in this case of overriding, the deposited tokens into the output places will only have the new color added by the transition)

Let us experiment coloring with the help of a simple example candidly called 'simple_adder'

19.1 Example-15: Simple Adder

This example presents an adder that adds two numbers input by the user.

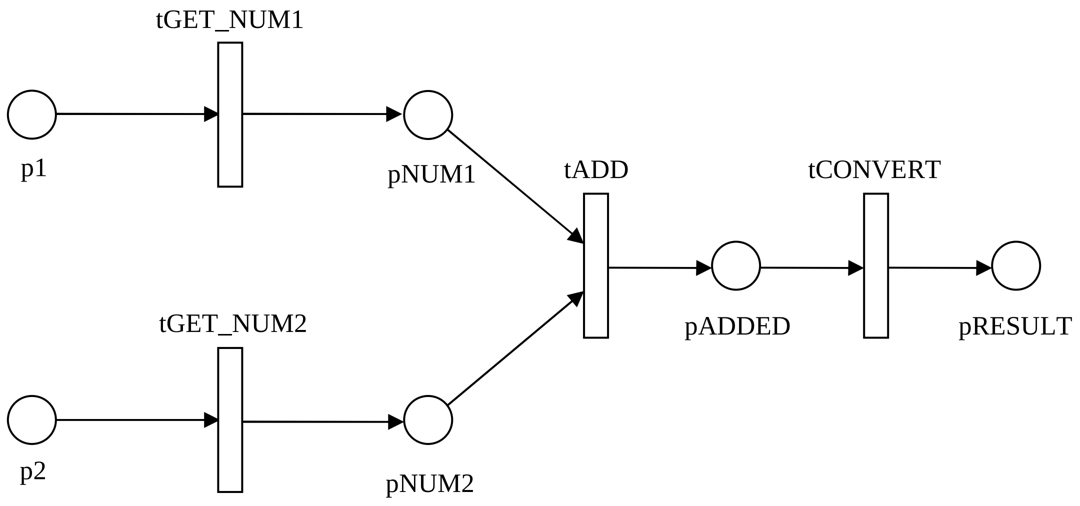


Figure-28. Simple Adder

Petri net model of a simple adder has 6 places and 4 transitions. Places **p1** and **p2** are just to keep the initial tokens so that the system can be started. Transitions **tGET_NUM1** and **tGET_NUM2** get an input number each from the user; let say the numbers fed by the user are 21 and 45. Then these two transitions convert the numbers into strings (**'**21**'** and **'**45**'**) and then add the strings as colors to the output tokens deposited into **pNUM1** and **pNUM2** respectively. Thus, the places **pNUM1** and **pNUM2** have tokens with input numbers as the colors.

Transition **tADD** does nothing in terms of colors. When it fires, by default, it deposits a token into the output place with the inherited colors. Hence, the token in place **pADDED** will have two colors ({'21', '45'}).

The final transition **tCONVERT** does five activities:

1. First it gets the two colors (strings ‘21’ and ‘45’) of the token in place **pADDED**.
2. Then it converts the strings into numbers (21 and 45),
3. It adds these two numbers together to make the sum (66).
4. Then it coverts the sum into a string (‘66’), and
5. Finally, it adds this string as color to the token deposited into the place **pRESULT**. The transition will also override inheritance so that the sum will be the only color of the token deposited into **pRESULT**

19.1.1 MSF: ’simple_adder.m’

```
% MSF for Example-15: simple_adder.m
clear, clc;
pn = petrinetgraph('simple_adder_def');
dynamicpart.initial_markings = {'p1',1, 'p2',1};

[results] = gpensim(pn, dynamicpart);
print_colormap(results, {'p1','p2','pNUM1', ,...
                  'pNUM2','pADDED', 'pRESULT'});
```

19.1.2 PDF: ’simple_adder_def.m’

```
% PDF for Example-15: simple_adder_def.m

function [PN_name, set_of_places, set_of_trans, set_of_arcs]...
                   = simple_adder_def(global_info)

PN_name='Color example: Simple Adder';
set_of_places={'p1', 'p2', 'pNUM1', 'pNUM2', 'pADDED','pRESULT'};
set_of_trans={'tGET_NUM1','tGET_NUM2','tADD','tCONVERT'};
set_of_arcs={'p1','tGET_NUM1',1, 'tGET_NUM1','pNUM1',1,...
             'p2','tGET_NUM2',1, 'tGET_NUM2','pNUM2',1,...
             'pNUM1','tADD',1, 'pNUM2','tADD',1,...
             'tADD','pADDED',1, 'pADDED','tCONVERT',1, ...
             'tCONVERT','pRESULT',1};
```

19.1.3 TDF: ’tGET_NUM1.m’

The TDF will ask the user to input a number:

```
function [fire, new_color, override, selected_tokens,global_info] = ...
    tGET_NUM1_def (pn, new_color, override, selected_tokens,global_info)
%% TDF: tGET_NUM1_def

num1 = input('input number-1: ');
new_color = num2str(num1);

fire=1;  %always fire
```

19.1.4 TDF: ’tGET_NUM2.m’

The TDF will ask the user to input another number:

```
function [fire, new_color, override, selected_tokens,global_info] = ...
    tGET_NUM2_def (pn, new_color, override, selected_tokens,global_info)
%% TDF: tGET_NUM2_def

num2 = input('input number-2: ');
new_color = num2str(num2);

fire=1;  %always fire
```

19.1.5 TDF: ’tADD.m’

There is no need for TDF tADD. It, by default, inherits colors from input tokens and put the colors to the output token.

19.1.6 TDF: ’tCONVERT.m’

```
function [fire, new_color, override, selected_tokens,global_info] = ...
    tCONVERT_def (pn, new_color, override, selected_tokens,global_info)
%% TDF: tCONVERT_def

% first, select any token
tokID = select_token(pn, 'pADDED', 1);

% second, get the colors of the selected token
colors = get_color(pn, tokID);

num1 = str2num(colors{1});  % convert color-1 into number
num2 = str2num(colors{2});  % convert color-2 into number

new_color = num2str(num1+num2);
override = 1; % only sum as color - NO inheritance

fire=1;  %always fire
```

19.1.7 Simulation Results

The statement,

```
    print_colormap(results, {'p1','p2','pNUM1','pNUM2','pADDED',
'pRESULT'});
```

prints colors of all the places. As shown in the screen dump below,

- **p1** has no colors,
- **p2** has no colors,
- **pNUM1** has ‘21’ as the color,
- **pNUM2** has ‘45’ as the color,
- **pADDED** has both ‘21’ and ‘45’ as colors, and
- **pRESULT** has ‘66’ as the color

```
input number-1: 21
input number-2: 45


Color Map for place: p1


Color Map for place: p2


Color Map for place: pNUM1
Time: 0
    '21'


Color Map for place: pNUM2
Time: 0
    '45'


Color Map for place: pADDED
Time: 0
    '21'    '45'


Color Map for place: pRESULT
Time: 0
66
```

19.2 Example-16: Alternative Design for Simple Adder

In the previous subsection, the sum is stored as a color inside a token that was deposited on the place **pRESULT**. You may prefer getting the sum as a variable too so that it can be freely used as you want. You can achieve this with a simple design change.

In addition to storing the sum as a color on the deposited token, you can also let the transition **tCONVERT** to store the sum as an element of **global_info**. In fact, **global_info** is meant for this kind of activities, getting information somewhere within a transition so that the information can be passed to subsequent transitions and back to the main simulation file. The new **tCONVERT** given below does the same five activities, but the last activity includes storing the sum as an element of **global_info**:

The final transition **tCONVERT** does five activities:

1. (no change) It gets the two colors (strings '21' and '45') of the token in place **pADDED**.
2. (no change) Then it converts the strings into numbers (21 and 45),
3. (no change) It adds these two numbers together to make the sum (66).
4. (no change) Then it coverts the sum into a string ('66'), and
5. **(REVISED)** Finally, it adds this string as color to the token deposited into the place **pRESULT**. The transition will also override inheritance so that the sum will be the only color of the token deposited into **pRESULT In addition, the sum will be stored as an element of global_info.**

The new TDF for **tCONVERT** is given below:

```
function [fire, new_color, override, selected_tokens,global_info] = ...
    tCONVERT_def (pn, new_color, override, selected_tokens,global_info)
%% TDF: tCONVERT_def

% first, select any token from pADDER
tokID = select_token(pn, 'pADDED', 1);

% second, get the colors of the selected token
colors = get_color(pn, tokID);

num1 = str2num(colors{1});  % convert color-1 into number
num2 = str2num(colors{2});  % convert color-2 into number
sum = num1 + num2;

new_color = num2str(sum);   % set the sum as the new color
global_info.sum = sum; %%% sum is added to global_info

override = 1; % only sum as color - NO inheritance

fire=1;  %always fire
```

There will be slight modifications in the MSF too:

1. To start the simulations, we have to pass **global_info** with the element '**sum**' to **gpensim**.
2. After simulations, we do not need to print the colormap to study the results; instead we will inspect the **global_info**.

The new MSF is given below:

```
% MSF for Example-16: Simple Adder with Color (Version 2)
% FILE simple_adder_2.m
```

```
clear, clc;
pn = petrinetgraph('simple_adder_def');
dynamicpart.initial_markings = {'p1',1, 'p2',1};

global_info.sum = 0; %% this is necessary

[results, global_info] = gpensim(pn, dynamicpart, global_info);

%% print value of the element 'global_info.sum'
disp(['The sum of two numbers : ', num2str(global_info.sum)]);
```

The result printed on the screen is given below:

```
input number-1: 21
input number-2: 45


The sum of two numbers : 66
>>
```

20. Token Selection based on Color

A transition may select input tokens based on color. This is done by executing the function **`select_token_color`**. There are 4 input parameters to this function: the Petri net structure at run-time, the input place of the transition, number of tokens to be selected, and finally the required color of the token.

The output parameter of the function is a set of IDs of the selected tokens (set of **tokID**). Of course, the number of returned **tokID** may be not equal to the number originally wanted by the transition, depending on availability.

Usage example: if a transition wants 4 tokens from the input place **pBUFF** with color 'TYPE-A', then the transition will execute the following statement:

```
X = select_token_color(PN,'pBUFF',4,'TYPE-A');
```

The returned value **X** is a set of **tokID** consisting of **tokID** for 0-4 tokens. If X is empty then no tokens are available with the required color. If X consists on 1, 2, or 3 **tokID**, then the request by the transition is partially fulfilled. If X consists of 4 **tokID**, then the request is fulfilled fully.

20.1 Example-17: Selecting Input Tokens with Specific Color

Figure given below depicts a production process. Place **pGEN** represents raw materials, and transition **tGEN** represents a machine that produces 3 types of products:

- 'type-A' with 10% production rate,
- 'type-B' with 30% production rate, and
- 'type-C' with rest 60% of the time.

Though buffer **pBUFF** contains all three types of products, Transition **tA** is supposed to select 'type-A' products only. Similarly, **tB** selects 'type-B' products and **tC** selects 'type-C' products only.

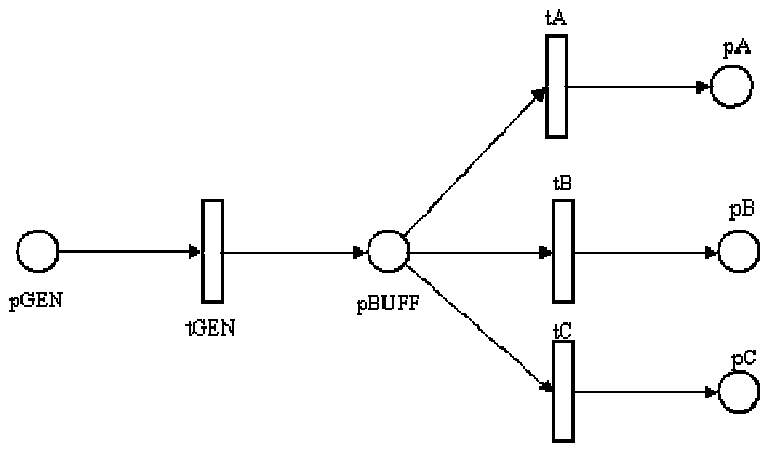


Figure-29. Selecting tokens with specific color

During simulations, **tGEN** adds new color to tokens that will be deposited in **pBUFF**. The new color will be 'type-A' 10% of the time, 'type-B' 30% of the time and 'type-C' 60% of the time. Since **tA** will consume only tokens with color 'type-A', tokens with color 'type-A' are deposited in **pA**; similarly, **pB** and **pC** will have only tokens with color 'type-B' and 'type-C' respectively.

20.1.1 MSF

The main simulation file is given below; it shows that number of initial tokens in **pGEN** is 100:

```
% MSF for Example-17: COLOR Selection EXAMPLE
global_info.ratio_A=0.10;
global_info.ratio_B=0.30;
global_info.ratio_C=0.60;

png = petrinetgraph('select_color_def');
dyn.initial_markings = {'pGEN',30};

[RES] = gpensim(png, dyn, global_info);
plotp(RES, {'pA', 'pB', 'pC'});
print_colormap(RES, {'pA', 'pB', 'pC'});
```

20.1.2 PDF

The Petri net definition file is given below:

```
% PDF for Example-17: COLOR Selection EXAMPLE
function [PN_name, set_of_places, set_of_trans, set_of_arcs] ...
    = select_color_def(global_info)

PN_name = 'SELECT COLOR Example';
set_of_places = {'pGEN', 'pBUFF', 'pA', 'pB', 'pC'};
set_of_trans={'tGEN', 'tA', 'tB', 'tC'};

set_of_arcs={'pGEN','tGEN',1 ,'tGEN','pBUFF',1, ...
    'pBUFF','tA',1,'tA','pA',1,...
    'pBUFF','tB',1,'tB','pB',1,...
    'pBUFF','tC',1,'tC','pC',1};
```

20.1.3 TDF: 'tGEN_def.m'

The first transition definition file is for the transition **tGEN**. The only task of this transition definition file is to produce tokens with a color: either 'type-A' or 'type-B'.

```

function [fire,new_color,over_ride, selected_tokens,global_info] = ...
    tGEN_def (PN, new_color, over_ride, selected_tokens, global_info)

random_number = rand(1);
```

```
if (random_number < global_info.ratio_A),
    new_color = 'type-A';
elseif (random_number < (global_info.ratio_A + global_info.ratio_B)),
    new_color = 'type-B';
else
    new_color = 'type-C';
end;

fire = 1;
```

From the above code, it is visible, that the transition always fire if enabled (fire=1); however, it also add a color (‘type-A’, ‘type-B’ or ‘type-C’) to new tokens deposited into **pBUFF**.

20.1.4 TDFs for tA, tB, and tC

The only task of this transition definition file for **tA**, **tB**, and **tC** is to select tokens with specific color. In the TDF for **tA**, we force the transition **tA** to select ‘type-A’ tokens only:

```
function [fire, new_color, over_ride, selected_tokens,global_info] = ...
    tA_def (PN, new_color, over_ride, selected_tokens, global_info)
%%%% TDF: tA_def

tokID = select_token_color(PN,'pBUFF',1,'type-A');

selected_tokens = tokID; % this token must be removed, none other
fire = (selected_tokens); % FIRE ONLY IF 'Selected_tokens' IS NOT EMPTY
```

First, tokens from input place **pBUFF** with color ‘type-A’ is selected by using the function **select_token_color**. The third parameter - ‘1’ - is the number of tokens needed. If selection is successful, then the identity number of the selected token (**tokID**) is returned as the output parameter. By copying tokID to `selected_tokens`, we inform the system that this token must be consumed by the transition. Finally, we allow the transition to fire only if tokID is not empty, meaning that there exist a token with ‘type-A’ color.

20.1.5 Simulation results

Figure-23 shows the plot of tokens in **pA, pB,** and **pC**. Since ‘type-C’ is produced 60% of the time, there will about 6 times more tokens in **pC** than in **pA** and **pB**. The results shown in figure-23 agrees.

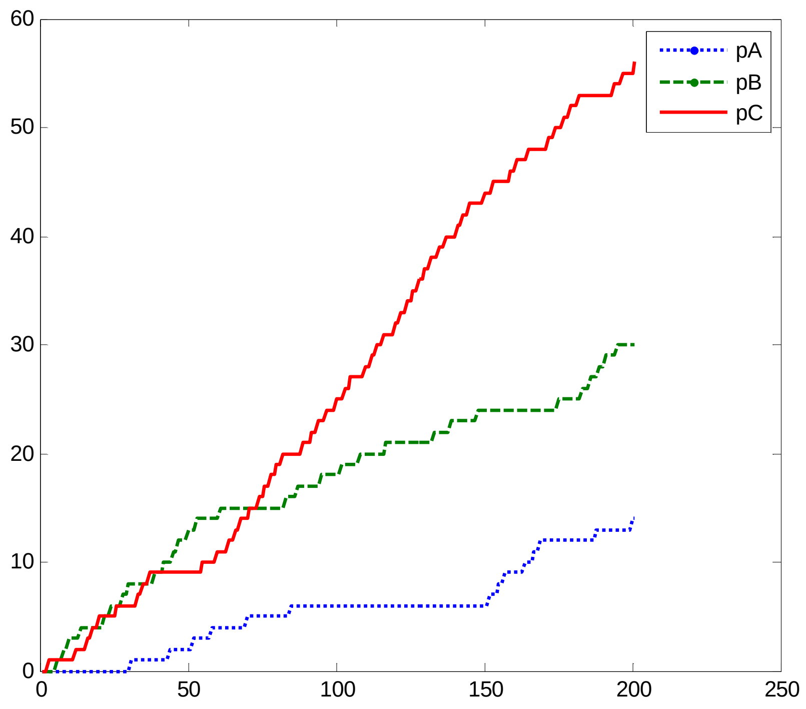


Figure-30. Simulation results of ‘select_color’ demo.

In addition, we can also inspect the colormap. In **pA**, the only color of any token is ‘type-A’.

```
Color Map for place: pA
Time: 0
    'type-A'


Color Map for place: pB
Time: 0
    'type-B'


Color Map for place: pC
Time: 0
    'type-C'

>>
```

20.2 Required or Preferred Color?

This is an important issue. With a very small change, we can allow a transition to prefer (‘may’) a color than require (‘must’) a color.

In the example given above, we forced the transition **tA** to select a token with color ‘type-A’. This is done first by selecting a token with ‘type-A’ color. Function **`select_token_color`** will return tokID if a token is with ‘type-A’ color is available or else returned tokID value will be empty (‘[]’). And then we forced the transition to fire only if tokID is not zero, meaning there is at least one token with the required color, so that the transition can fire.

However, we may also allow transition to ***prefer*** ‘type-A’ tokens. This means, if ‘type-A’ tokens are available, they will be consumed; if not, one of the other existing tokens of ‘type-B’ or ‘type-C’will be consumed. The newer TDF given below prefers (rather than forcing) ‘type-A’ tokens:

```
function [fire, new_color, over_ride, selected_tokens,global_info] = ...
    tA_def (PN, new_color, over_ride, selected_tokens, global_info)

selected_tokens = select_token_color(PN,'pBUFF',1,'type-A');

fire = 1;
```

This transition always fires if enabled (because fire=1), regardless of ‘type-A’ tokens are available or not. It will also consume ‘type-A’ tokens if available (if ‘selected_tokens’ list is not empty).

Let us think about a generic case: if a transition needs **`m`** tokens from an input place to fire (arc weight **`m`**), and has obtained **`n`** numbers preferred tokens (**`selected_tokens`** list has **`n`** tokIDs). If **`m`** is greater than **`n`**, then the system consumes (removes) **`n`** number of specific tokens (identified by the tokIDs in the **`selected_tokens`** list) and the rest **`m-n`** tokens will be other arbitrary tokens in the input place.

20.2.1 Simulations

TDFs for **tA**, **tB**, and **tC** are changed so that tokens with specific colors are preferred (not required).

Simulations show that now pA, pB, and pC have tokens with all colors.

```
Color Map for place: pA
Time: 0
    'type-A'     'type-B'     'type-C'



Color Map for place: pB
Time: 0
    'type-A'     'type-B'     'type-C'



Color Map for place: pC
Time: 0
    'type-A'     'type-B'     'type-C'

>>
```

20.2.2 Example-18: Selecting Input Tokens with 2 or more colors

In this example, we make a tiny change to **tA** so that **tA** make select either ‘type-A’ or ‘type-B’ color.

```
function [fire, new_color, over_ride, selected_tokens,global_info] = ...
    tA_def (PN, new_color, over_ride, selected_tokens, global_info)
%%%% TDF: tA_def

tokID1 = select_token_color(PN,'pBUFF',1,'type-A');
tokID2 = select_token_color(PN,'pBUFF',1,'type-B');

selected_tokens = [tokID1 tokID2]; % one of these token must be removed
fire = (selected_tokens); % FIRE ONLY IF 'Selected_tokens' IS NOT EMPTY
```

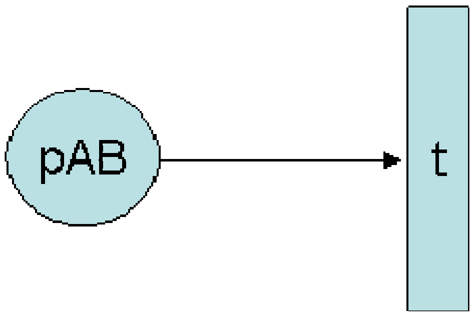
Now we see that tokens in **pA** have both ‘type-A’ and ‘type-B’ colors.

```
Color Map for place: pA
Time: 0
    'type-A'    'type-B'


Color Map for place: pB
Time: 0
    'type-B'


Color Map for place: pC
Time: 0
    'type-C'

>>
```

21. Summary: Token Selection based on Color

21.1 Token Selection From A Single Input Place

Let's say that place **pAB** has tokens with many colors including {'A', 'B', 'X', 'Y', {'A', 'B'}, {'A', 'X'}, {'A', 'Y'}, {'B', 'X'}, …. {'A', 'B', 'X', 'Y'}}.



- Transition **t** selects token with color 'A' from **pAB** (meaning tokens with color {'A'}or {'A', 'B'} or {'A', 'X'} are relevant):

Program code in TDF:

```
     selected_tokens = select_token_color(PN,'pAB',1,'A'); %
     fire = (selected_tokens); % MUST
```

- Transition **t** selects 'A' **<u>or</u>** 'B' from **pAB**:

Program code in TDF:

```
        tokID1 = select_token_color(PN,'pAB',1,'A');
        tokID2 = select_token_color(PN,'pAB',1,'B');
        selected_tokens = [tokID1 tokID2]; % tokens to be removed
        fire = (length(selected_tokens) >= 1); % MUST
```

- Transition **t** **<u>prefers</u>** 'A' or 'B' from **pAB**:

Program code in TDF:

```
        tokID1 = select_token_color(PN,'pAB',1,'A');
        tokID2 = select_token_color(PN,'pAB',1,'B');
        selected_tokens = [tokID1 tokID2]; % tokens to be removed
        fire = 1; %
```

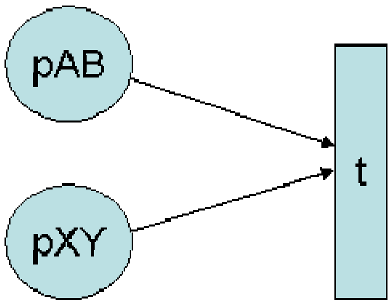
- Transition **t** selects a token with ‘A’ **and** ‘B’ from **pAB**:

Program code in TDF:

```
        selected_tokens = select_token_color(PN,'pAB',1, {'A‘, ‘B’});

        fire = (selected_tokens); % MUST
```

21.2 Token Selection From Multiple Input Places

Let’s say that place **pAB** has tokens with colors {‘’, ‘A’, ‘B’, {‘A’, ‘B’}} and pXY has tokens with colors {‘’, ‘X’, ‘Y’, {‘X’, ‘Y’}}.



- Transition **t** selects ‘A’ from **pAB and** ‘Y’ from **pXY**:

Program code in TDF:

```
          tokID1 = select_token_color(PN,'pAB',1,'A');

          tokID2 = select_token_color(PN,'pXY',1,'X');

          selected_tokens = [tokID1 tokID2]; % tokens to be removed

          fire = (length(selected_tokens) == 2); % MUST
```

- Transition **t** select ‘A’ from **pAB or** ‘X’ from **pXY** (at least one token be ‘A’ or ‘X’):

Program code in TDF:

```
          tokID1 = select_token_color(PN,'pAB',1,'A');

          tokID2 = select_token_color(PN,'pXY',1,'X');

          selected_tokens = [tokID1 tokID2]; % tokens to be removed

          fire = (length(selected_tokens) >= 1); % MUST
```

- Transition **t prefers** ‘A’ from **pAB** or ‘X’ from **pXY**:

Program code in TDF:

```
        tokID1 = select_token_color(PN,'pAB',1,'A');
        tokID2 = select_token_color(PN,'pXY',1,'X');
        selected_tokens = [tokID1 tokID2]; % tokens to be removed
        fire = 1; % may
```

22. Token Selection based on Time

A transition may select input tokens based on time. In the current version GPenSIM 3.0, selection can be done based on two policies: 'FCFS' (First-Come-First-Served) and 'LCFS' (Last-Come-First-Served). Selection of time based token is done by executing the function **`select_token_time.`** There are 4 input parameters to this function: the Petri net structure at run-time, the input place of the transition, number of tokens to be selected, and finally the time-based selection policy ('FCFS' or 'LCFS').

The output parameter of the function is a set of IDs of the selected tokens (set of **tokID**). Of course, the number of returned **tokID** may be not equal to the number originally wanted by the transition, depending on token availability.

Usage example: if a transition wants **4** **oldest** tokens from the input place **pBUFF**, then the transition will execute the following statement:

```
function [fire,new_color,override, selected_tokens,global_info] = ...
    tLR_A_def (PN, new_color, override, selected_tokens, global_info)

selected_tokens = select_token_time(PN,'pBUFF',4, 'FCFS');

fire = 1;
```

If **pBUFF** has more than equal to 4 tokens, then tokIDs of the 4 oldest tokens will be returned in **`selected_tokens`**. Otherwise, if **pBUFF** has less than 4 tokens, then tokIDs of all the tokens will be returned.

Similarly, if a transition wants **2** **youngest** tokens from the input place **pBUFF**, then the transition will execute the following statement:

```
function [fire,new_color,override, selected_tokens,global_info] = ...
    tLR_A_def (PN, new_color, override, selected_tokens, global_info)

selected_tokens = select_token_time(PN,'pBUFF',2, 'LCFS');

fire = 1;
```

22.1 Example-19: Token selection based on time

Figure-24 shows the example for token selection based on time. pSTART has 3 initial tokens (initial tokens are of course colorless). tCOL add colors to the tokens it deposits into pQUEUE. The branch “pDLY-tDLY-pRDY” is a delay, just to keep tSEL wait until all the three tokens are deposited into pQUEUE.

tCOL adds color to tokens followingly:

- Gets current time from the system.
- Converts current time into ASCII string
- Adds the ASCII string as color

This means all the three tokens deposited into pQUEUE will have colors reflecting the time they were made by tSEL.

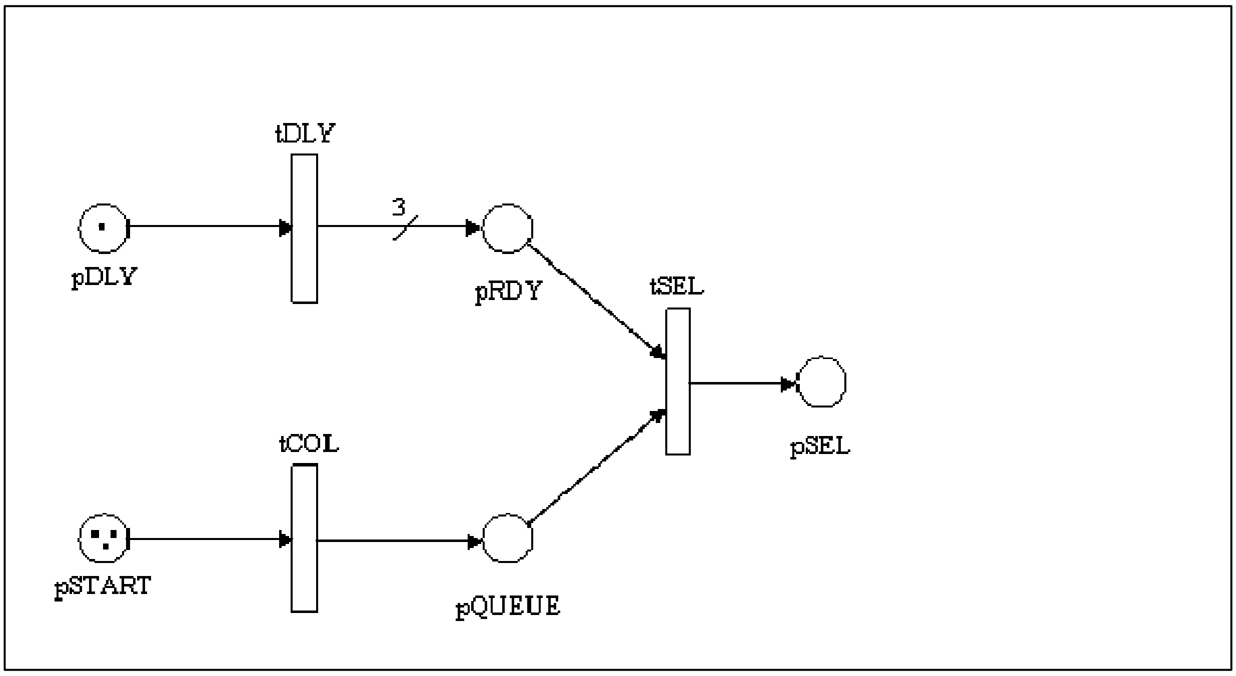


Figure-31. FCFS example

22.1.1 PDF: fcfs_def.m

```
% PDF for Example-19: Token selection based on time
function [PN_name, set_of_places, set_of_trans, set_of_arcs] ...
    = fcfs_def(global_info)

PN_name = 'FCFS - LCFS DEMO';
set_of_places = {'pSTART', 'pQUEUE', 'pDLY', 'pRDY','pSEL'};
set_of_trans={'tCOL', 'tSEL', 'tDLY'};
set_of_arcs={'pSTART','tCOL',1, 'tCOL','pQUEUE',1,...
    'pQUEUE','tSEL',1, 'tSEL','pSEL',1,...
    'pDLY','tDLY',1, 'tDLY','pRDY',3,...
    'pRDY','tSEL',1};
```

22.1.2 MSF: fcfs.m

```
% MSF for Example-19: Token selection based on time
png = petrinetgraph('fcfs_def');

dyn.initial_markings = {'pSTART',3, 'pDLY',1};
dyn.firing_times = {'tCOL',1,'tDLY',100, 'tSEL',10};

RES = gpensim(png, dyn);

print_statespace(RES);
print_colormap(RES, 'pSEL');
```

22.1.3 TDF: tCOL_def.m

```
function [fire, new_color, over_ride, selected_tokens,global_info] =
    ...
    tCOL_def (PN, new_color, over_ride, selected_tokens,
    global_info)
%%%% TDF: tCOL_def

% add color
new_color = num2str(PN.current_time);

fire = 1;
```

22.1.4 TDF: tSEL_def.m

```
function [fire,new_color,override, selected_tokens,global_info] =
    ...
    tSEL_def (PN, new_color, override, selected_tokens, global_info)

selected_tokens = select_token_time(PN,'pQUEUE',1, 'FCFS');
fire = 1;
```

22.1.5 Simulation Results

The simulation result clearly shows that **tSEL** selects tokens on “FCFS” basis. At **pSEL**, 3 tokens arrive; the first token had color ‘0’ then arrive a token with color ‘1’ and finally, come token with color ‘2’.

```
…
…

step:7    Firing event: tSEL     (Starting time: 120)  Finishing Time: 130
Current markings:
pSTART    pQUEUE    pDLY      pRDY      pSEL
 0         0         0         0         3
Completion time: 130
```

```
Displaying token colors (WARNING: processing takes time ....

Color Map for place: pSEL

Time: 110
    '0'

Time: 120
    '0'    '1'

Time: 130
    '0'    '1'    '2'
```

22.1.6 Simulation results for LCFS

Let's change selection policy to LCFS:

```
function [fire,new_color,override, selected_tokens,global_info] =
    ...
    tSEL_def (PN, new_color, override, selected_tokens, global_info)

selected_tokens = select_token_time(PN,'pQUEUE',1, 'LCFS');
fire = 1;
```

Then the simulation result also depicts LCFS selection by **tSEL**:

```
…
…

step:7    Firing event: tSEL     (Starting time: 120)  Finishing Time: 130
Current markings:
pSTART    pQUEUE    pDLY      pRDY      pSEL
 0                     0                0            0             3
Completion time: 130


Displaying token colors (WARNING: processing takes time ....

Color Map for place: pSEL

Time: 110
    '2'

Time: 120
    '1'    '2'

Time: 130
    '0'    '1'    '2'
```

Part-II: Applications

23. Modeling a Single Runway Airport

This project is to model a single runway airport. The aim is to propose a simple dynamic Petri net model that describes the traffic flow of a single runway (RWY) due to schedule (i.e. estimated times of arrivals and departures).

23.1 Description of the Model

Though the runway to be modeled is simple, it consists of the important elements of the runway dynamics.

23.1.1 Assumptions

In order to obtain a relatively simple model for simulation and dynamic analysis purposes, the following modeling assumptions are made:

- There are only three types of aircrafts (A/C) handled by the airport.
- The three types of A/Cs use pre-calculated runway length

23.1.2 Model elements

The important elements of the model are:

- Runway (RWY)
- Four taxiways (TWY)
- Aircrafts (A/Cs), arriving, taxing, engaged in terminals, and departing
- Rules that govern the interaction between A/C and use of the RWY

The characteristic properties of each of the model elements are as follows.

23.1.3 Runway (RWY) and taxiways (TWY)

A single 2500 m runway is considered with two 90^0 TWY on both end and two rapid exit taxiways (RETs) located at approx. 1000 m and 1500 m from approach end threshold (see figure 2).

23.1.4 The three categories of A/Cs

The difference between aircraft is based on International Civil Aviation Organization (ICAO) threshold speed categories (A to E). Only aircraft with categories A, B and C are considered. The selected traffic mix contains the following types of aircraft with percentage:

1. Category-A (e.g. lighter Cessna A/C): 30%,
2. Category-B (e.g. Medium Business Jets): 10%
3. Category-C (General Passenger Traffic): 60%

Category-A, B, and C A/Cs occupy 1500, 2000, 2500 meters of the RWY for landing and take-off, respectively.

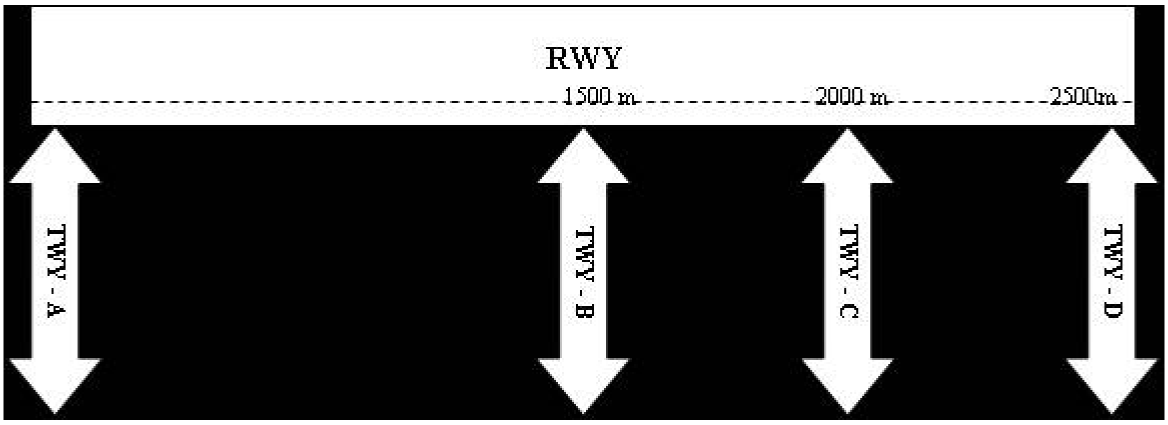


Figure-32. Elements of the runway

23.1.5 Governing rules

The following rules are used to control the interactions between A/C and the use of the runway.

1. Arrivals have priority on departures
2. A landing aircraft will not normally be permitted to cross the runway threshold on its final approach until the preceding departing A/C has crossed the end of the runway, or has started a turn, or until all preceding landing A/C are clear off the RWY. That is, the model is governed by elementary air traffic control (ATC) principles, such as, only one aircraft at a time on RWY, and arrivals have priority over departures.

23.1.6 Timing for simulations

Runway occupancy times (ROT) for landing and departures are assumed to be equal for a specific category A/C:

- Category-A A/Cs take 5 minutes (and first 1500 m of the RWY)
- Category-B A/Cs take 7 minutes (and first 2000 m of the RWY)
- Category-C A/Cs take 9 minutes (and the whole 2500 m of the RWY)

Besides:

- For arriving A/Cs, taxiing through any TWYs takes 5 minutes;
- For departing A/Cs, lineup time for take-off is same taxiing time for arriving A/Cs
- A/Cs arrive at a rate of 15-60 minutes (assume random timing)
- Arrived A/C take service time (offloading and on-boarding passengers and goods) of about 45 minutes
- Initially, there may be some A/Cs parked on turf or terminals (assume any number of A/Cs)
- YOU MAY ASSUME ANY OTHER TIMING

23.2 The Petri net Model

23.2.1 The Elements

- Air crafts
- Runway
- Exit ways (for taxiing)
- Terminal, and
- Control tower

23.2.2 Process Modules

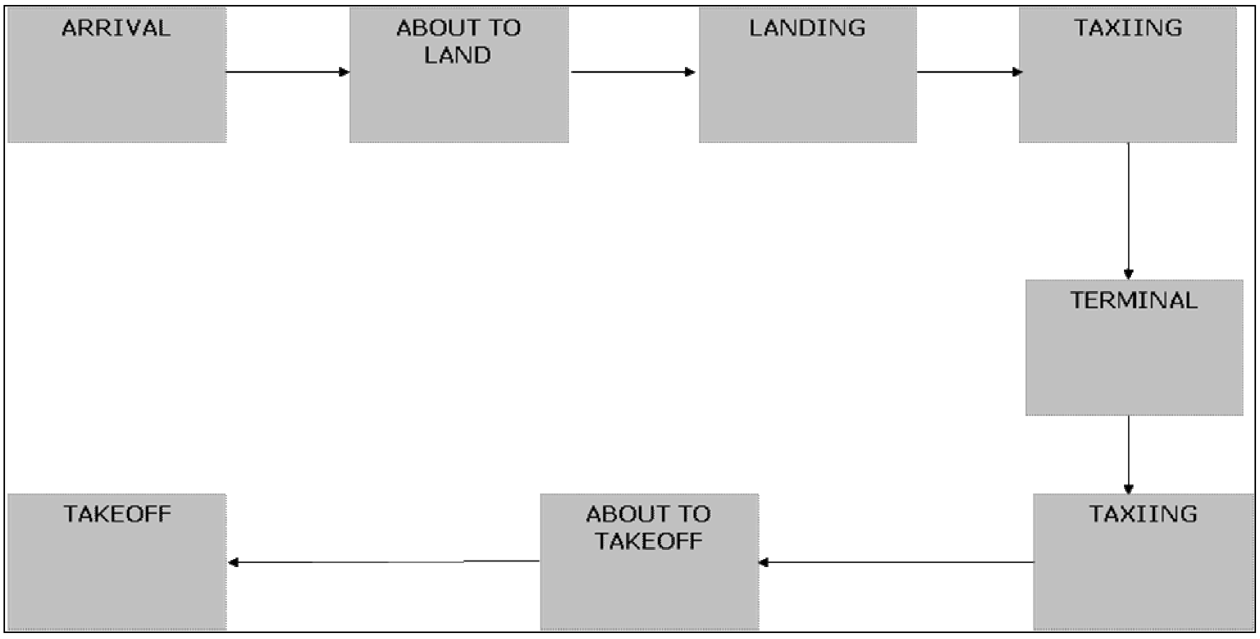


Figure-33. Elements of the runway

23.2.3 The Petri net Model

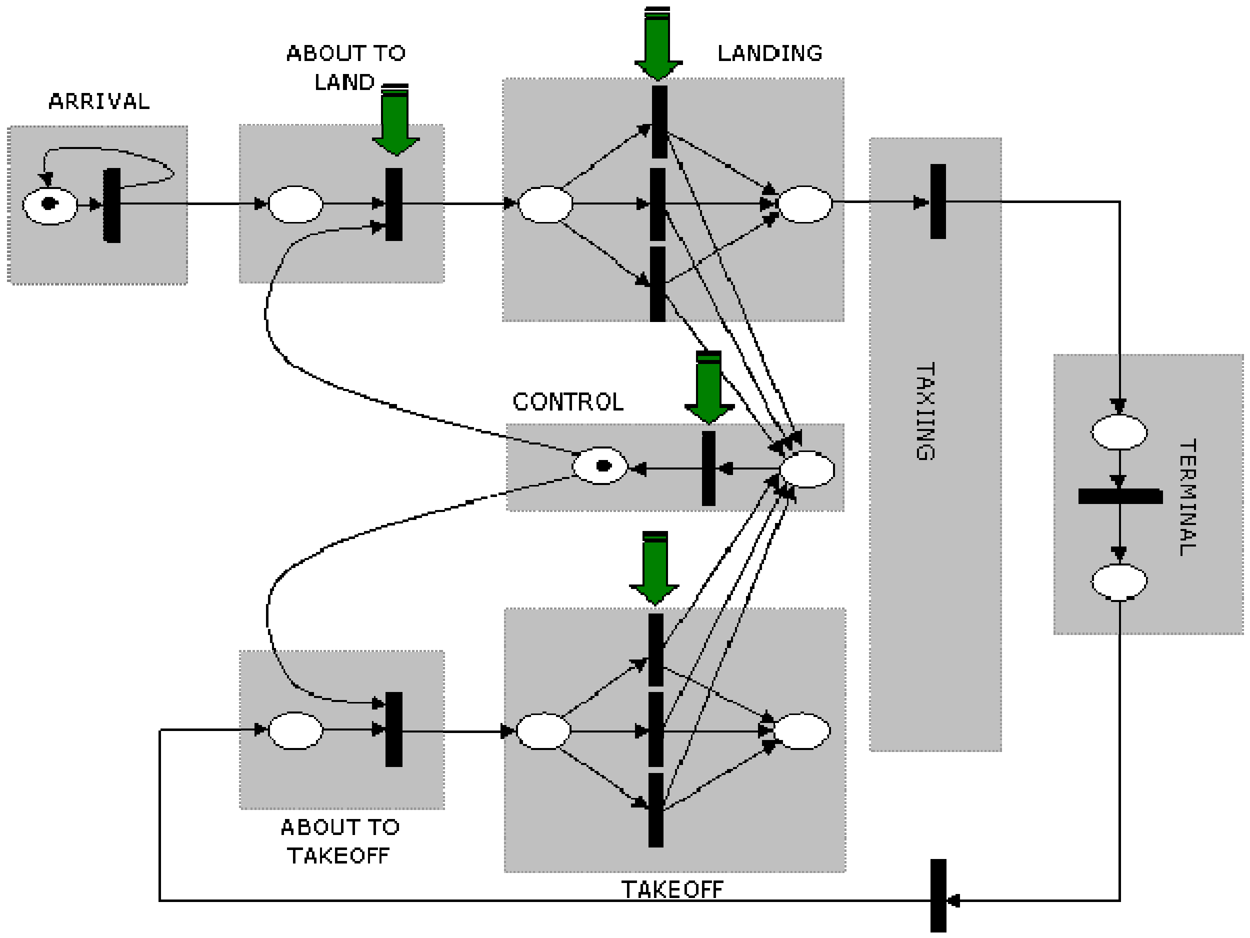


Figure-34. The Petri net model showing only one terminal

23.2.4 Places and transitions

- Module-1: ARRIVAL: pARR, tARR
- Module-2: ABOUT TO LAND: pW4L: Wait for landing
  tGPL: Granting Permission for landing
- Module-3: LANDING: pR2L: Ready to Land;
  tLR1: Landing RWY length-1; tLR2: Landing RWY length-2;
  tLR3: Landing RWY length-3; pACL: A/C Landed
- Module-4: TAXIING: tT2T: Taxiing to Terminal;
  tT2R: Taxiing to RWY
- Module-5: TERMINAL: pR2B: Ready to Board;
  tBRD: Boarding; pR2D: Ready to depart
- Module-6: ABOUT TO TAKEOFF: pW4T: Wait for Takeoff;

tGPT: Granting Permission for Takeoff

- Module-7: TAKEOFF: pR2T: Ready to Takeoff;
  tTR1: Takeoff RWY length-1; tTR2: Takeoff RWY length-2;
  tTR3: Takeoff RWY length-3; pACD: A/C Departed
- Module-8: CONTROL: pCTR1: Runway to Control Tower,
  pCTR2: Control Tower 2 Runway
  tCLC: clear token color

23.3 Program Code: MSF

23.3.1 MSF

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  NARVIK; modeling a single runway airport
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
clear; clc;
global_info.ratio_A=0.30;
global_info.ratio_B=0.10;
global_info.ratio_C=0.60;

global_info.MAX_LOOP = 200;
global_info.LOOP_NUMBER = 1;

ARRIVAL_FREQUENCY = 30;      % the main variable !!!

%%%% STATIC DETAILS %%%%
png = petrinetgraph('single_rwy_def');

%%%% DYNAMIC DETAILS %%%%
dyn.initial_markings = {'pARR',100, 'pCTR2', 1};
dyn.firing_times = {'tARR', ARRIVAL_FREQUENCY, 'tGPL', 0,...
    'tLRA',5, 'tLRB',7, 'tLRC',9,...
    'tT2T',5, 'tBRD',45, 'tT2R',5, 'tGPT',0,...
    'tTRA',5, 'tTRB',7, 'tTRC',9};

%%%% SIMULATE %%%%%
[RES, global_info] = gpensim(png, dyn, global_info);
print_statespace(RES);
plotp(RES, {'pW4L', 'pR2B', 'pW4T'});
```

23.4 Program Code: PDF

```
function [PN_name, set_of_places, set_of_trans, set_of_arcs] ...
    = single_rwy_def(global_info)
% PDF: single_rwy_def

PN_name = 'SINGLE RWY';

set_of_places = {'pARR', 'pW4L', 'pR2L', 'pACL', 'pR2B', ...
    'pR2D', 'pW4T', 'pR2T', 'pACD','pCTR1','pCTR2'};
```

```
set_of_trans =  {'tARR', 'tGPL', 'tLRA','tLRB','tLRC',...
    'tT2T', 'tBRD', 'tT2R', 'tGPT', ...
    'tTRA','tTRB','tTRC', 'tCLC'};

set_of_arcs = {...
    'pARR','tARR',1, 'tARR','pARR',1, 'tARR','pW4L',1,...
    'pW4L','tGPL',1, 'tGPL','pR2L',1, ...
    'pR2L','tLRA',1, 'pR2L','tLRB',1, 'pR2L','tLRC',1, ...
    'tLRA','pACL',1, 'tLRB','pACL',1,'tLRC','pACL',1,...
    'pACL','tT2T',1, 'tT2T','pR2B',1,...
    'pR2B','tBRD',1, 'tBRD','pR2D',1,...
    'pR2D','tT2R',1, 'tT2R','pW4T',1,...
    'pW4T','tGPT',1, 'tGPT','pR2T',1,...
    'pR2T','tTRA',1, 'pR2T','tTRB',1, 'pR2T','tTRC',1, ...
    'tTRA','pACD',1, 'tTRB','pACD',1,'tTRC','pACD',1,...
    'tLRA','pCTR1',1, 'tLRB','pCTR1',1,'tLRC','pCTR1',1,...
    'tTRA','pCTR1',1, 'tTRB','pCTR1',1,'tTRC','pCTR1',1,...
    'pCTR1','tCLC',1, 'tCLC','pCTR2',1,...
    'pCTR2','tGPL',1, 'pCTR2','tGPT',1,...
    };
```

23.5 Program Code: TDFs

23.5.1 TDF for tGPL (Adding Color)

```
function [fire,new_color,over_ride, selected_tokens,global_info] = ...
    tGPL_def (PN, new_color, over_ride, selected_tokens, global_info)

% function [fire,new_color,selected_tokens,global_info] = t2_def (PN,...
%     new_color, selected_tokens, global_info)

over_ride = 1;

random_number = rand(1);
if (random_number < global_info.ratio_A),
    new_color = 'CAT-A';
    global_info.A_count = global_info.A_count + 1;
elseif and ((random_number >= global_info.ratio_A),...
        (random_number < (global_info.ratio_A + global_info.ratio_B))),
    new_color = 'CAT-B';
    global_info.B_count = global_info.B_count + 1;
else
    new_color = 'CAT-C';
    global_info.C_count = global_info.C_count + 1;
end;

fire = 1;
```

23.5.2 TDF for tLRA (Landing A-type AC)

```
function [fire, new_color, over_ride, selected_tokens,global_info] = ...
```

```
    tLRA_def (PN,new_color, over_ride, selected_tokens, global_info)

% function [fire,new_color,selected_tokens,global_info] = ...
%   tLRA_def (PN,new_color, selected_tokens, global_info)

selected_tokens = select_token_with_colors(PN,'pR2L',1,'CAT-A');

if ~isempty(selected_tokens),
    global_info.tLRA_count = global_info.tLRA_count + 1;
    fire = 1;
else
    fire = 0;
end;
```

23.5.3 TDF for tLRB (Landing B-type AC)

```
function [fire, new_color, over_ride, selected_tokens,global_info] = ...
    tLRB_def (PN,new_color, over_ride, selected_tokens, global_info)

% function [fire,new_color,selected_tokens,global_info] = ...
%   tLRB_def (PN,new_color, selected_tokens, global_info)

selected_tokens = select_token_with_colors(PN,'pR2L',1,'CAT-B');

if ~isempty(selected_tokens),
    global_info.tLRB_count = global_info.tLRB_count + 1;
    fire = 1;
else
    fire = 0;
end;
```

23.5.4 TDF for tLRC (landing C-type AC)

```
function [fire, new_color, over_ride, selected_tokens,global_info] = ...
    tLRC_def (PN,new_color, over_ride, selected_tokens, global_info)

% function [fire,new_color,selected_tokens,global_info] = ...
%   tLRC_def (PN,new_color, selected_tokens, global_info)

selected_tokens = select_token_with_colors(PN,'pR2L',1,'CAT-C');

if ~isempty(selected_tokens),
    global_info.tLRC_count = global_info.tLRC_count + 1;
    fire = 1;
else
    fire = 0;
end;
```

23.5.5 TDF for tTRA (Take Off, A-type AC)

```
function [fire, new_color, over_ride, selected_tokens,global_info] = ...
    tTRA_def (PN,new_color, over_ride, selected_tokens, global_info)

% function [fire,new_color,selected_tokens,global_info] = ...
%   tTRA_def (PN,new_color, selected_tokens, global_info)

selected_tokens = select_token_with_colors(PN,'pR2T',1,'CAT-A');

if ~isempty(selected_tokens),
    fire = 1;
else
    fire = 0;
end;
```

23.5.6 TDF for tTRB (Take Off, B-type AC)

```
function [fire, new_color, over_ride, selected_tokens,global_info] = ...
    tTRB_def (PN,new_color, over_ride, selected_tokens, global_info)

% function [fire,new_color,selected_tokens,global_info] = ...
%   tTRB_def (PN,new_color, selected_tokens, global_info)

selected_tokens = select_token_with_colors(PN,'pR2T',1,'CAT-B');

if ~isempty(selected_tokens),
    fire = 1;
else
    fire = 0;
end;
```

23.5.7 TDF for tTRC (Take Off, C-type AC)

```
function [fire, new_color, over_ride, selected_tokens,global_info] = ...
    tTRC_def (PN,new_color, over_ride, selected_tokens, global_info)

% function [fire,new_color,selected_tokens,global_info] = ...
%   tTRC_def (PN,new_color, selected_tokens, global_info)

selected_tokens = select_token_with_colors(PN,'pR2T',1,'CAT-C');

if ~isempty(selected_tokens),
    fire = 1;
else
    fire = 0;
end;
```

23.5.8 TDF for tCLC (removing color in tokens)

```
function [fire,new_color,over_ride, selected_tokens,global_info] = ...
    tCLC_def (PN, new_color, over_ride, selected_tokens, global_info)

% function [fire,new_color,selected_tokens,global_info] = ...
%   tCLC_def (PN,new_color, selected_tokens, global_info)

over_ride = 1;
fire = 1;
```

23.6 Simulation Results

Finding the Bottleneck for varying arrival rate:

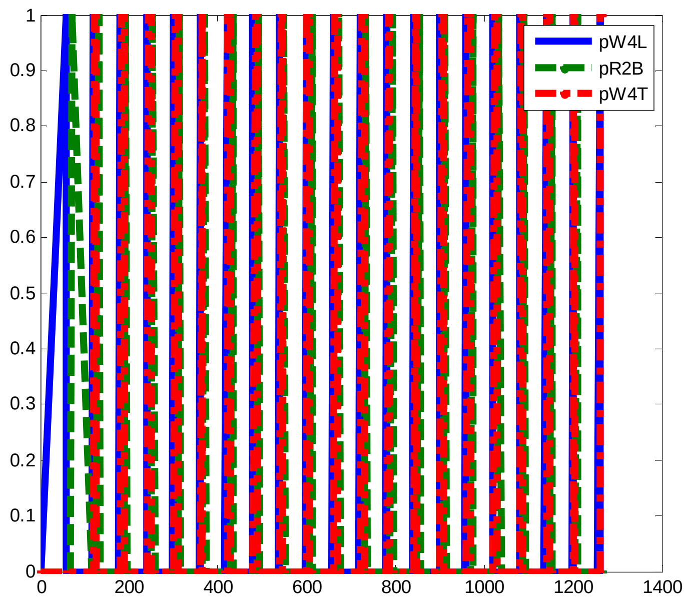


Figure-35. Arrival of ACs: every 60 min

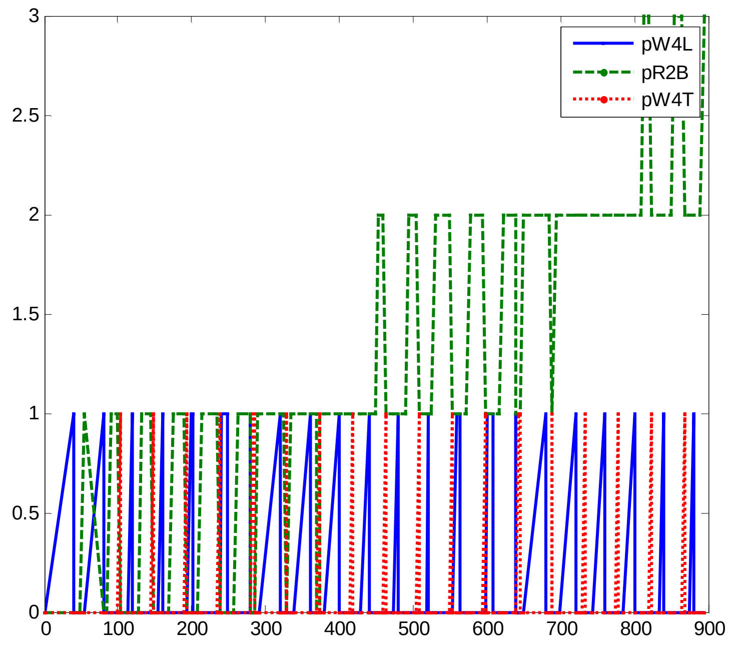


Figure-36. Arrival of ACs: every 40 min

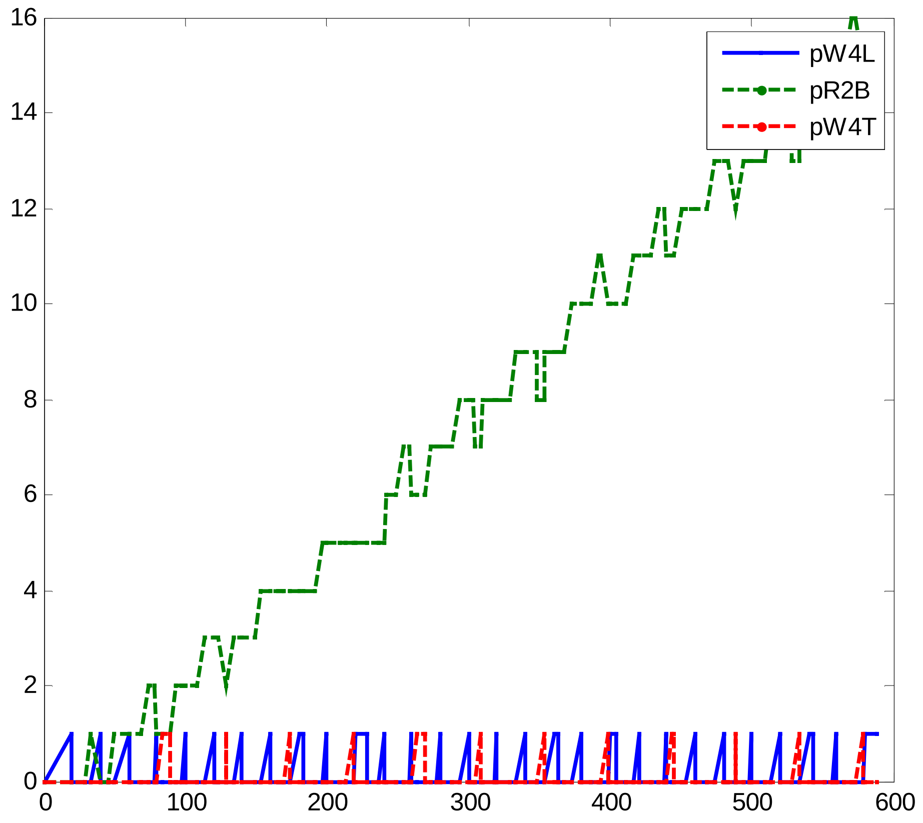


Figure-37. Arrival of ACs: every 20 min

23.7 Discussion

- For all frequencies (like flights every 60 min, 40 min, and 20 min), maximum number of flights waiting in the air (‘pW4L’) is 1. Therefore RWY is not the bottleneck.
- Condition-1 (at any time, only one AC in RWY) is satisfied structurally.
- How to satisfy ATC Condition-2: Landing has priority over takeoff?
- Only one gate is used in the model. Thus, Gate is the bottleneck in simulations (‘pR2B’)
- However, single RWY is obviously a problem considering close-down for maintenance and for fault-tolerance
- How can the Petri net model easily modified for Stavanger-Sola (Double RWY)

23.8 Improvement to simulation model – job arrival in predefined times

In the Petri net model shown in figure-30, the aircraft arrival generator (or generally, job arrival generator) is given as a loop that will create aircraft arrivals with specific intervals; this could be slightly improved by using a stochastic value e.g. ‘normrnd(45, 5)’ meaning that aircraft arrives at about every 45 minutes with STD 5 minutes. But, still this will not help we have to generate arrivals at specific (or predefined) times. Generating arrivals at predefined times can be elegantly done with the help of global_info, as shown in the following example.

23.9 Example-26: Arrivals at predefined times

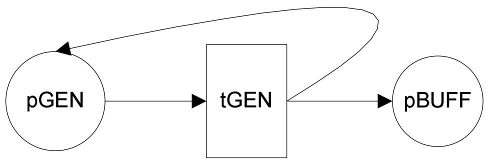


Figure-38. Arrival at predefined times

Let us assume that jobs arrive at pre-defined times, e.g. at the following time: 4, 10, 22, 34, 36, and 75.

23.9.1 MSF

```
% Example-26: A Example for pre-defined arrival times
% file: profile_pn_def.m:
clear, clc;

global_info.MAX_LOOP = 500;
global_info.Arrival_Times = [4, 10, 22, 34, 36, 75];

png = petrinetgraph({'arrivals_def'});
dynamic.initial_markings = {'pGEN',1};
sim = gpensim(png, dynamic, global_info);

print_statespace(sim);
plotp(sim, {'pBUFF'});
```

23.9.2 PDF

```
% Example-26: A Example for pre-defined arrival times
% file: arrivals_def.m:

function [PN_name, set_of_places, set_of_trans, set_of_arcs]...
    = predefined_def(global_info)

PN_name = 'Demo for pre-assigned arrival times';
set_of_places = {'pGEN', 'pBUFF'};
set_of_trans = {'tGEN'};
set_of_arcs =  {'pGEN','tGEN',1, 'tGEN','pGEN',1, 'tGEN', 'pBUFF', 1};
```

23.9.3 TDF ‘tGEN_def.m’

```
function [fire,new_color,override, selected_tokens,global_info] = ...
    tGEN_def (PN, new_color, override, selected_tokens, global_info)

fire = 0; % to start with

if ~isempty(global_info.Arrival_Times),
    Current_AT = global_info.Arrival_Times(1);

    if le(Current_AT, PN.current_time),  % less than or equal
            global_info.Arrival_Times = global_info.Arrival_Times(2:end);
            fire = 1;
    end;
end;
```

23.9.4 Simulation Results

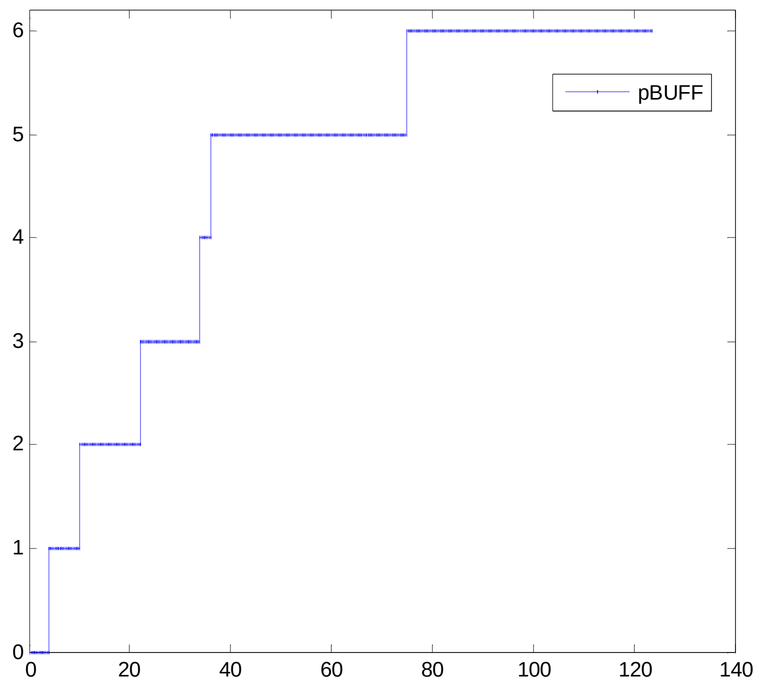


Figure-39. Jobs generation at predefined times

24. Scheduling

We present two examples in this section. Example-xx is a warm up example. In example-xx, we go through the “better-intended, worst-happened” phenomena normally associated with scheduling. Problems stated in the examples are taken from Stein (2008).

24.1 Example-81: Minimizing completion time

Figure-34, a digraph, shows the tasks to be done to complete a work. The figure shows the order in which the tasks to be done and the time required to complete each task. E.g. Task T1 requires 4 time units and tasks T1 and T2 must be completed before task T4.

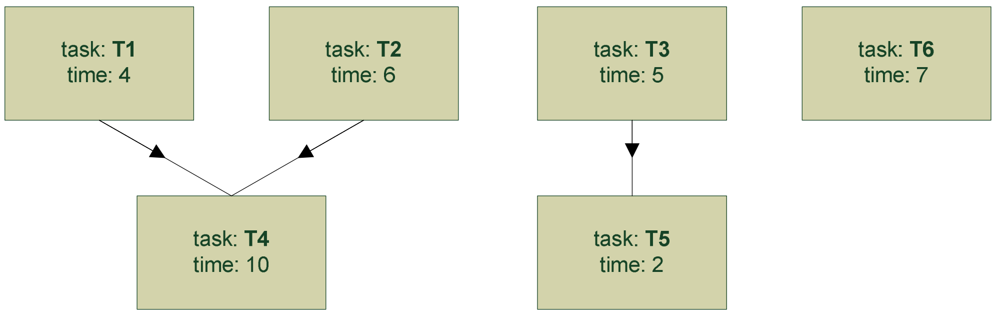


Figure-40. Digraph showing order of tasks to be completed

Note that it will take a minimum of 16 time units to complete all the tasks, as task T2 followed by T4, which requires 16 time units, is the critical path – the path of longest duration.

The algorithm used for simulations is the priority-list scheduling. The order of priority (high to low) is assumed to be T1, T2, … , and T6. finally, we assume two human resources, generic and can do any task, named ‘Al, and ‘Bob’.

24.1.1 Petri net model

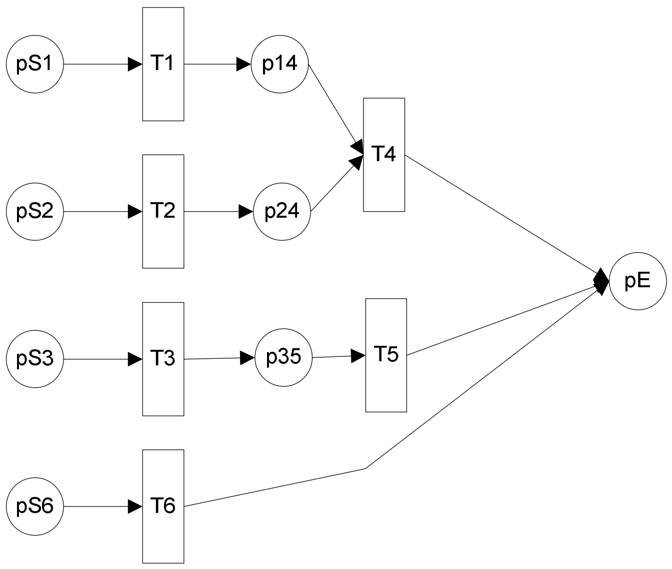


Figure-41.   Petri Net model of the scheduling digraph

The PDF for the Petri net model shown in figure –XX2 is given below:

PDF (‘schedule01_def.m’):

```
% Example-81: Scheduling-01
% file: schedule01_def.m: PDF

function [PN_name, set_of_places, set_of_trans, set_of_arcs]...
                    = schedule01_def(global_info)

PN_name='Scheduling example 01';

set_of_places={'pS1','pS2','pS3','pS6', 'pE', ...
               'p14','p24', 'p35'};
set_of_trans={'T1', 'T2', 'T3', 'T4', 'T5', 'T6'};

set_of_arcs={'pS1','T1',1, 'pS2','T2',1, 'pS3','T3',1, 'pS6','T6',1, ...
             'T1','p14',1, 'T2','p24',1, 'T3','p35',1, ...
             'p14','T4',1, 'p24','T4',1, 'p35','T5',1, ...
             'T4','pE',1, 'T5','pE',1, 'T6','pE',1, };
```

24.2 Programs

In the preprocessor of each task, we will try to grab a resource that is available; the resources are implemented as a semafors.

The pre-processor for task T1 (‘**T1_pre.m**’) is given below; other pre-processors for other tasks are similar – the only change is the ***task_nr***, which is underlined in the code snippet given below:

```
function [fire, new_color,override,selected_tokens,global_info] = ...
    T1_pre(PN, new_color,override,selected_tokens,global_info)

% T1_pre

task_nr = 1;    % TASK-1

occu_semafor = global_info.semafor;
semafor = ~occu_semafor;

[row, cols] = find(semafor);    % find any available semafor (value ~= 0)

if ~isempty(cols),
    sema = cols(1); % which is the first avialble semafor
    global_info.my_semafor(task_nr) = sema; % that will be mine
    global_info.semafor(sema) = task_nr;  % then reserve it

    % pack results
    global_info.timing(task_nr, 1) = sema;   % task handler
    global_info.timing(task_nr, 2) = PN.current_time; % task starting time
    fire = 1;
else
    fire = 0;
end;
```

In the post-processor of each task, we will release the semafor after use. The post-processor for task T1 (‘**T1_post.m**’) is given below; again, the post-processors for the other task are similar, we only need to change the ***task_nr***.

```
function [global_info] = ...
    T1_post(transition, PN, global_info)
% function T1_post
%

task_nr = 1;    % TASK-1

my_semafor = global_info.my_semafor(task_nr);  % which is my semafor
global_info.semafor(my_semafor) = 0;      % release that

% Pack results: task completion time
global_info.timing(task_nr, 3) = PN.current_time; % task completion time
```

Finally, the MSF (‘**schedule01.m**’) is given below:

```
% Example-81:
% MSF: scheule01.m
clear, clc;

no_of_employees = 2;
no_of_tasks = 6;

global_info.semafor    = zeros(1, no_of_employees); % employees available
global_info.my_semafor = zeros(1, no_of_tasks);

global_info.PRIORITY = {'T1', 'T2', 'T3', 'T4', 'T5', 'T6'};

global_info.timing = zeros(no_of_tasks, 3);

png = petrinetgraph('schedule01_def');

dynamicpart.initial_markings = {'pS1',1, 'pS2',1, 'pS3',1, 'pS6',1};
dynamicpart.firing_times = {'T1',4, 'T2',6, 'T3',5, 'T4',10, 'T5',2,
'T6',7};

[sim, global_info] = gpensim(png, dynamicpart, global_info);

timing = global_info.timing;
print_schedule(timing, {'Al', 'Bob'});
```

In the MSF, we are using a print function called ‘**print_schedule.m**’, to make better printout. This function is given below:

```
function print_schedule(timing, list_of_names)
% function print_schedule(timing, list_of_names)

no_of_employees = length(list_of_names);

[timing_rows, timing_cols] = size(timing);

for employee = 1:no_of_employees,
    disp('  ');
    disp([' *** ', list_of_names{employee}, ' ***']);

    for i=1:timing_rows,
        if eq(timing(i,1), employee),
            disp(['Task', num2str(i), ':   [',...
                num2str(timing(i,2)), ', ', num2str(timing(i,3)),']']);
        end;
    end;
end;
disp('  ');
```

24.3 Results

When we use only one resource (‘Al’), the time taken will be summation of all the time for individual tasks, 34 time units.

```
% Example-81:
% MSF: scheule01.m
no_of_employees = 1;
…
…
…
print_schedule(timing, {'Al'});
```

The result of simulation is:

```
*** Al ***
Task1:   [0, 4]
Task2:   [4, 10]
Task3:   [10, 15]
Task4:   [15, 25]
Task5:   [25, 27]
Task6:   [27, 34]
```

When we use two resources (‘Al’ and ‘Bob’), the time taken is 18 time units to complete all the tasks:

```
% Example-81:
% MSF: scheule01.m
no_of_employees = 2;
…
…
print_schedule(timing, {'Al', 'Bob'});
```

```
 *** Al ***
Task1:   [0, 4]
Task3:   [4, 9]
Task5:   [9, 11]
Task6:   [11, 18]

 *** Bob ***
Task2:   [0, 6]
Task4:   [6, 16]
```

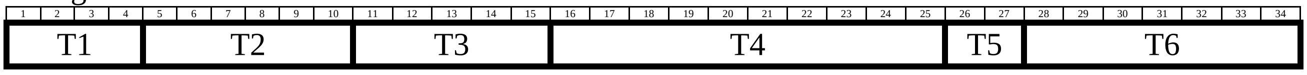
However, if we use three resources (‘Al’, ‘Bob’, and ‘Carter’), then the maximum time needed is the critical path time, that is 16 time units.

```
% Example-81:
% MSF: scheule01.m
no_of_employees = 3;
…
…
print_schedule(timing, {'Al', 'Bob', 'Carter'});
```

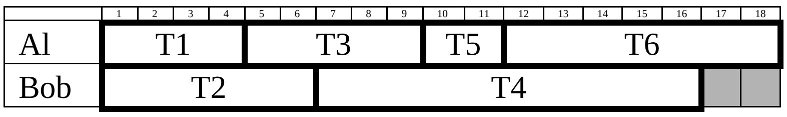
```
 *** Al ***
Task1:   [0, 4]
Task6:   [4, 11]

 *** Bob ***
Task2:   [0, 6]
Task4:   [6, 16]

 *** Carter ***
Task3:   [0, 5]
Task5:   [5, 7]
```

24.3.1 In Summary:

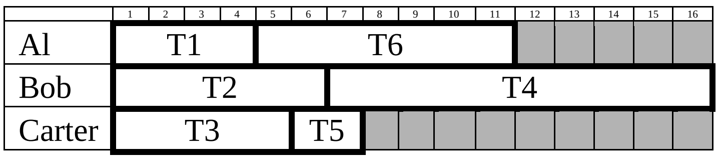
When only one resource (‘Al’) is used:
Completion time: 34 time units
Usage of resources = 100%



When two resources (‘Al’ and ‘Bob’) are used:
Completion time: 18 time units
Idle time: Bob: 2 time units



When three resources (‘Al’, ‘Bob’, and ‘Carter’) are used:
Completion time: 16 time units
Idle time:
Al: 5 time units
Carter: 9 time units



24.4 Example-82: Scheduling – II

Figure-36 shows another example.

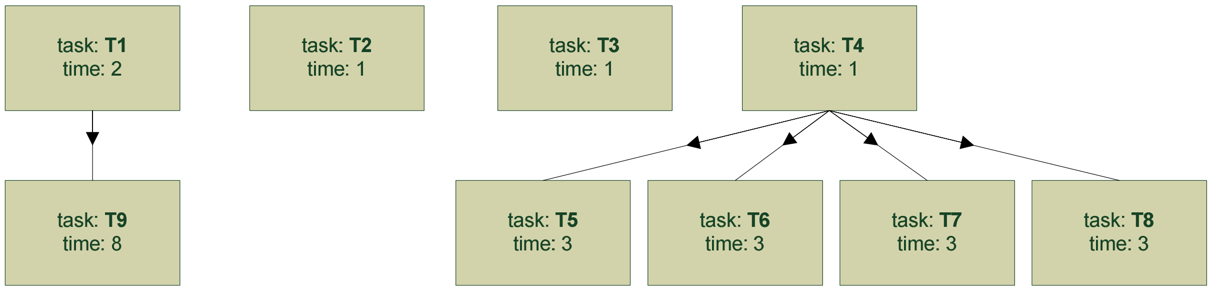


Figure-42. Digraph for example-82

In this example too, the priority of tasks are assumed as previously (top to bottom): T1, T2, …, T9

When three resources (‘Al’, ‘Bob’, ‘Carter’) are used, the completion time is found to be 12 time units. This is a “perfect storm” scenario, finishing the job by the time of the critical path (T1, T3), which is 12 time units.

	1	2	3	4	5	6	7	8	9	10	11	12
Al	T1			T9								
Bob	T2		T4		T5				T7			
Carter	T3		Idle						T8			

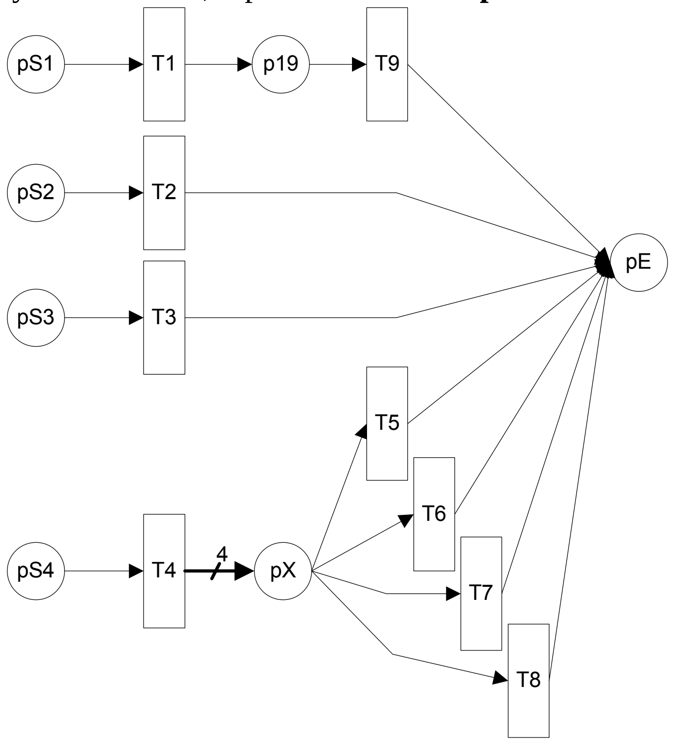
Let’s add another resource (‘Don’) and see how much the completion times are reduced.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Al	T1			T8				▨	▨	▨	▨	▨	▨	▨	▨
Bob	T2		T5				T9								
Carter	T3		T6				▨	▨	▨	▨	▨	▨	▨	▨	▨
Don	T4		T7				▨	▨	▨	▨	▨	▨	▨	▨	▨

The results above shows that when we add more resources, we make things worse as completion time is now increased. Now the completion time is 15 time units.

24.4.1 Petri Net Model

Figure given below shows the Petri net model. Note that the weight of arc between **T4** and **pX** is 4. This means, every time **T4** fires, it puts 4 tokens into **pX**.



This means, we have to make sure that these 4 tokens are consumed by the 4 transitions T5, T6, T7 and T8, one token for each transition.

24.4.2 Programming

PDF (‘schedule02_def.m’):

```
% Example-82: Scheduling-02
% file: schedule02_def.m: PDF

function [PN_name, set_of_places, set_of_trans, set_of_arcs]...
                    = schedule02_def(global_info)

PN_name='Scheduling example 02';

set_of_places={'pS1','pS2','pS3','pS4', 'pE', ...
                'p19','pX'};
set_of_trans={'T1', 'T2', 'T3', 'T4', 'T5', 'T6', 'T7', 'T8', 'T9'};

set_of_arcs={'pS1','T1',1, 'pS2','T2',1, 'pS3','T3',1, 'pS4','T4',1, ...
             'T1','p19',1, 'p19','T9',1, ...
             'T9','pE',1, 'T2','pE',1, 'T3','pE',1, ...
             'T4','pX',4, ...
             'pX','T5',1, 'pX','T6',1, 'pX','T7',1, 'pX','T8',1, ...
             'T5','pE',1, 'T6','pE',1, 'T7','pE',1, 'T8','pE',1};
```

MSF (‘schedule02.m’):

```
% Example-82:
% MSF: scheule02.m
clear, clc;

no_of_employees = 4;
no_of_tasks = 9;

global_info.semafor    = zeros(1, no_of_employees); % employees available
global_info.my_semafor = zeros(1, no_of_tasks);
global_info.PRIORITY = {'T1','T2','T3','T4','T5','T6','T7','T8','T9'};

global_info.timing = zeros(no_of_tasks, 3);

png = petrinetgraph('schedule02_def');

dynamicpart.initial_markings = {'pS1',1, 'pS2',1, 'pS3',1, 'pS4',1};
dynamicpart.firing_times = {'T1',3, 'T2',2, 'T3',2, 'T4',2, ...
                'T5',4, 'T6',4, 'T7',4, 'T8',4, 'T9',9};

[sim, global_info] = gpensim(png, dynamicpart, global_info);
%grid on, plotp(sim, {'p14', 'p24','p35','pE'});


timing = global_info.timing;
three_chaps = {'Al', 'Bob', 'Chuck'};
four_chaps  = {'Al', 'Bob', 'Chuck', 'Don'};

if (no_of_employees==3),
    print_schedule(timing, three_chaps);
else
    print_schedule(timing, four_chaps);
end;
```

24.4.3 Pre-processor for T1, T2, T3, T4 and T9:

The only job of the preprocessors T1_pre to T4_pre, and T9_pre is to grab an available so that they can start. However, the preprocessors for T5-T8 have one more job to do, that is to make sure that they fire only once (or consume only one token after T4 has fired).

Pre-processor for T1, T2, T3, T4 and T9 are similar:

```
function [fire, new_color,override,selected_tokens,global_info] = ...
    T1_pre(PN, new_color,override,selected_tokens,global_info)

% T1_pre

task_nr = 1;   % TASK-1

occu_semafor = global_info.semafor;
semafor = ~occu_semafor;

[row, cols] = find(semafor);    % find any available semafor (value ~= 0)

if ~isempty(cols),
    sema = cols(1); % which is the first avialble semafor
    global_info.my_semafor(task_nr) = sema; % that will be mine
    global_info.semafor(sema) = task_nr;  % then reserve it

    % pack results
    global_info.timing(task_nr, 1) = sema;   % task handler
    global_info.timing(task_nr, 2) = PN.current_time; % task starting time

    fire = 1;
else
    fire = 0;
end;
```

Pre-processor for T5, T6, T7, and T8 are similar; they first check whether the transition is already fired once. If yes, then no more firing. Other wise, they try to grab a semafor.

```
function [fire, new_color,override,selected_tokens,global_info] = ...
    T5_pre(PN, new_color,override,selected_tokens,global_info)

% T5_pre

task_nr = 5;   % TASK-5

occu_semafor = global_info.semafor;
semafor = ~occu_semafor;

[row, cols] = find(semafor);    % find any available semafor (value ~= 0)

tx = get_trans(PN, 'T5');
```

```
if (tx.times_fired), %if T5 has already fired once, then dont fire anymore
    fire = 0;
    return;
end;

if ~isempty(cols),
    sema = cols(1); % which is the first avialble semafor
    global_info.my_semafor(task_nr) = sema; % that will be mine
    global_info.semafor(sema) = task_nr;  % then reserve it

    % pack results
    global_info.timing(task_nr, 1) = sema;   % task handler
    global_info.timing(task_nr, 2) = PN.current_time; % task starting time

    fire = 1;
else
    fire = 0;
end;
```

24.4.4 Post-processors

Post-processors for all the transition are similar; they just release the semafor the transitions were holding. The post-processor for T1 ('T1_post.m'):

```
function [global_info] = ...
    T1_post(transition, PN, global_info)
% function t1_post
%

task_nr = 1;   % TASK-1

my_semafor = global_info.my_semafor(task_nr);  % which is my semafor
global_info.semafor(my_semafor) = 0;     % release that

% Pack results: task completion time
global_info.timing(task_nr, 3) = PN.current_time; % task completion time
```

25. Stochastic Timer

This is an advanced topic, dealing with discretizing of continuous systems. We know that Petri net is for discrete event simulations only. However, if we could discretize continuous systems then these systems can also be modeled with Petri nets. However, this is not easy and needs some understanding of Petri net formalism and matrix representation. Interest reader is referred to a good book on this topic, Darren J. Wilkinson, "Stochastic Modelling for Systems Biology", Chapman & Hall/CRC, NY, 2006. ISBN-10 1-58488-540-8. Read especially about Gillespi's algorithm in chapter 06.

Stochastic timer: So far, we have been using inbuilt global timer for simulations. We did not use any user-defined timer or time series for advancing the clock. Sometimes, we do need to use special timers to advance the simulation time by ourselves. In this case, we use stochastic timer.

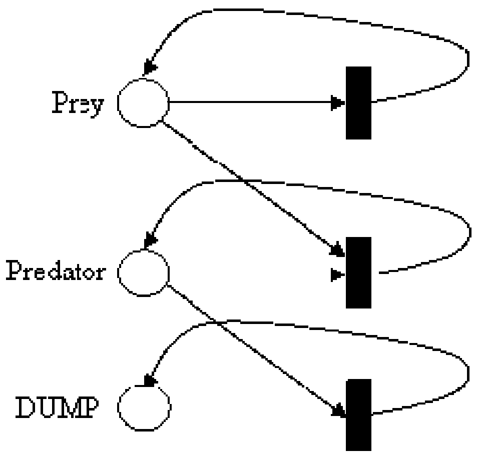


Figure-43. Petri net model of the Prey-Predator interaction

25.1 Example-25: The Prey-Predator ecological equilibrium

The equilibrium is stated by 2 simple differential equations (known as Lotka & Volterra equation):

- The specimen prey (e.g. rabbit - r) mutates by itself and depleted by predators (e.g. foxes - f):

$$\frac{dr}{dt} = (\alpha \cdot r) - (\beta \cdot r \cdot f)$$

- The specimen predator (e.g. fox) grows due to rabbits (access to food) and depleted by its own population (competition for food):

$$\frac{df}{dt} = -(\gamma \cdot f) + (\delta \cdot r \cdot f)$$

- $\alpha, \beta, \gamma,$ and δ are parameters representing the interaction of the two species.

25.2 Converting the dynamics to Petri nets

Of course, the equilibrium is determined by classical (partial) differential equations. Without using mathematical solutions, which demands high mathematical skills for higher order systems when many specimen types are involved, we go for the analytical reasoning using Petri nets. Equivalent Petri net model for the interaction is given below:

25.3 Simulation files

The program snippets using GPenSIM is given below:

- First, in the main simulation file, we have to set the flag for ‘stochastic timer’ (`global_info.STOCHASTIC = 1;`)
- Second, we have to define the stochastic timer in the file ‘time_advancement.m’

25.3.1 The Main Simulation File

```
% MSF file for Example-25: Predator-Pey example
% THIS EXAMPLE USES STOCHASTIC TIMER !!!
global_info.MAX_LOOP = 10000;
global_info.c = [1 .005 .6];

global_info.STOCHASTIC = 1; % set the flag for stochastic timer
global_info.LOOP_NUMBER = 1; % set this flag as MAX_LOOP is large

pn = petrinetgraph('predator_prey_def');
dynamicpart.initial_markings = {'Prey',50, 'Predator', 100};

sim = gpensim(pn, dynamicpart, global_info);
% NOTE: !!!
%   print function ‘print_statespace’
%   can not be used applications using stochastic timer !!!!!
%   !!!!!
plotp(sim, {'Prey','Predator'});    %%%% figure 28
plot(sim.LOG(:,2), sim.LOG(:,3));   %%%% figure 29
```

25.3.2 Petri net Definition File

```
%% PDF for Example-25: predator_prey_def.m:

function [PN_name, set_of_places, set_of_trans, set_of_arcs]...
                    = predator_prey_def(global_info)
PN_name='predator-prey p/151';
set_of_places={'Prey', 'Predator', 'DUMP'};
set_of_trans={'t1','t2','t3'};
set_of_arcs={'Prey','t1',1, 't1','Prey',2,...
    'Prey','t2',1, 'Predator','t2',1, 't2','Predator',2,...
    'Predator','t3',1, 't3','DUMP',1};
```

25.3.3 Definition of stochastic timer (‘time_advancement.m’)

```
%%%% !!!!!!!! CHANGING GLOBAL TIME !!!
%%%% time_advancement is for CHANGING GLOBAL TIME !!!
%%% this time series is a realization of “Gilespi algorithm”

function [pn, global_info] = time_advancement(pn, global_info)

c1=global_info.c(1);  c2=global_info.c(2);  c3=global_info.c(3);
```

```
Prey = get_place(pn, 'Prey');
PRED = get_place(pn, 'Predator');

h1 = c1 * Prey.tokens;
h2 = c2 * Prey.tokens * PRED.tokens;
h3 = c3 * PRED.tokens;
H = h1 + h2 + h3;

%%%%  probabilities
global_info.pro1 = (h1/H);
global_info.pro2 = (h2/H);
global_info.pro3 = (h3/H);

delta_T = 1-exp(-1/H);
pn.current_time = pn.current_time + delta_T ; %%%%  CHANGING GLOBAL TIME
!!!
```

25.3.4 Transition Definition File: t1_def.m

```
function [fire, new_color, override, selected_tokens,global_info] = ...
           t1_def (pn, new_color, override, selected_tokens,global_info)
% function t1_def

c1=global_info.c(1); c2=global_info.c(2); c3=global_info.c(3);

Prey = get_place(pn, 'Prey');
PRED = get_place(pn, 'Predator');

h1 = c1 * Prey.tokens;
h2 = c2 * Prey.tokens * PRED.tokens;
h3 = c3 * PRED.tokens; H = h1 + h2 + h3;

%%%%  probabilities
pro1=(h1/H); pro2=(h2/H); pro3=(h3/H);

R = rand*(1);
fire =  (R <= pro1);
```

25.3.5 Transition Definition File: t2_def.m

```
function [fire, new_color, override, selected_tokens,global_info] = ...
    t2_def (pn, new_color, override, selected_tokens,global_info)
% function fire = t2_def(pn, global_info)

c1=global_info.c(1);  c2=global_info.c(2);  c3=global_info.c(3);

Prey = get_place(pn, 'Prey');
PRED = get_place(pn, 'Predator');

h1 = c1 * Prey.tokens;
h2 = c2 * Prey.tokens * PRED.tokens;
h3 = c3 * PRED.tokens;
H = h1 + h2 + h3;

%%%%  probabilities
pro1=(h1/H);  pro2=(h2/H);  pro3=(h3/H);

R = rand*(1);
fire =  (R <= pro2);
```

25.3.6 Transition Definition File: t3_def.m

```
function [fire, new_color, override, selected_tokens,global_info] = ...
    t3_def (pn, new_color, override, selected_tokens,global_info)
% function fire = t3_def(pn, global_info)
% tRES_implementation

c1=global_info.c(1);  c2=global_info.c(2);  c3=global_info.c(3);

Prey = get_place(pn, 'Prey');
PRED = get_place(pn, 'Predator');

h1 = c1 * Prey.tokens;
h2 = c2 * Prey.tokens * PRED.tokens;
h3 = c3 * PRED.tokens;
H = h1 + h2 + h3;

%%%%  probabilities
pro1=(h1/H);  pro2=(h2/H);  pro3=(h3/H);

R = rand*(1);
fire =  (R <= pro3);
```

CAUTION! CAUTION!
Print functions 'print_statespace' can not be used for applications that use stochastic timer.

This is the reason for manipulating simulation results log file directly, as done in the example above. We give below code snippet from MSF for prey-predator example:

```
% NOTE: !!!
%   print function ‘print_statespace’
%   can not be used applications using stochastic timer !!!!!
%   !!!!!
plotp(sim, {'Prey','Predator'});    %%%% figure 28
plot(sim.LOG(:,2), sim.LOG(:,3));   %%%% figure 29
```

25.4 The Simulation Results

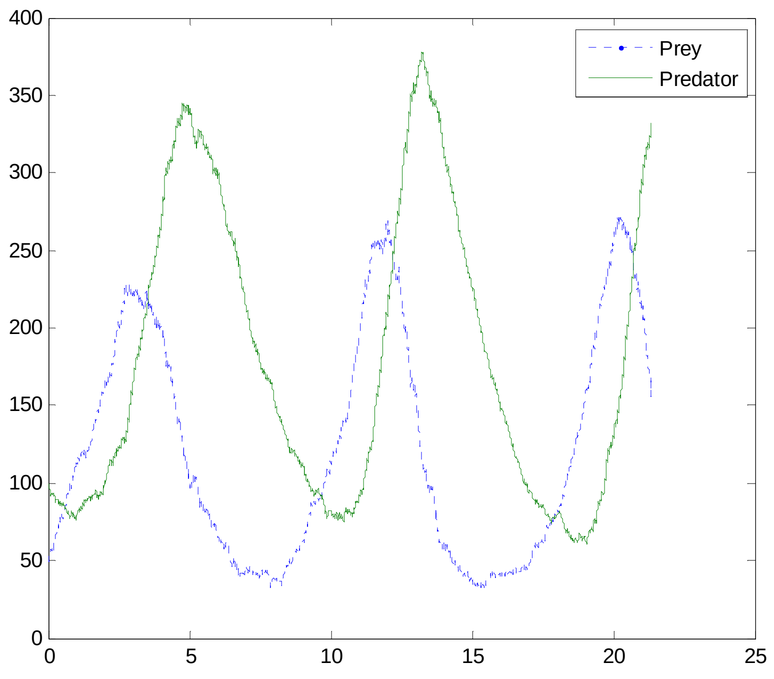


Figure-44. Composition of specimens Prey-Predator with time

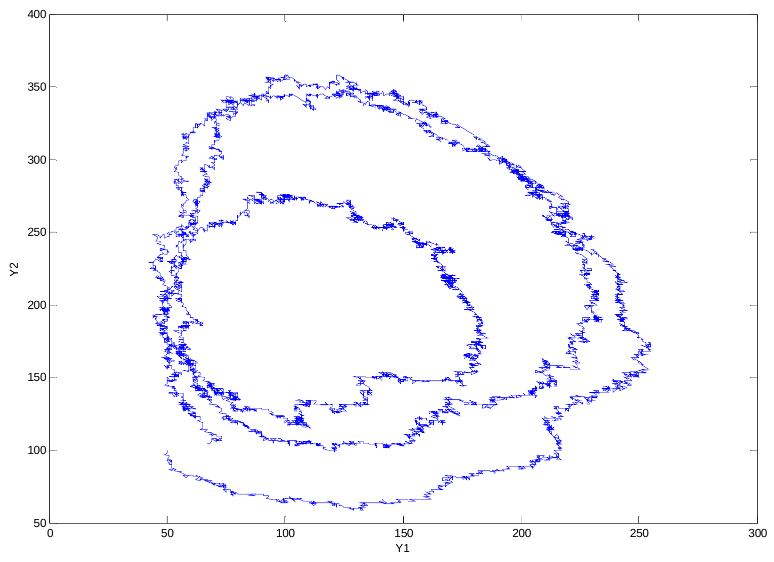


Figure-45. Prey-Predator Equilibrium

26. Measuring Robot Usage

The flexible manufacturing cell at the Narvik Institute of Technology (NIT), Norway, consists of a CNC vertical machining center (Mori Seiki), a CNC horizontal machining center (Mori Seiki), an ABB IRB2000 robot, and a conveyor belt; figure 12 shows the system.

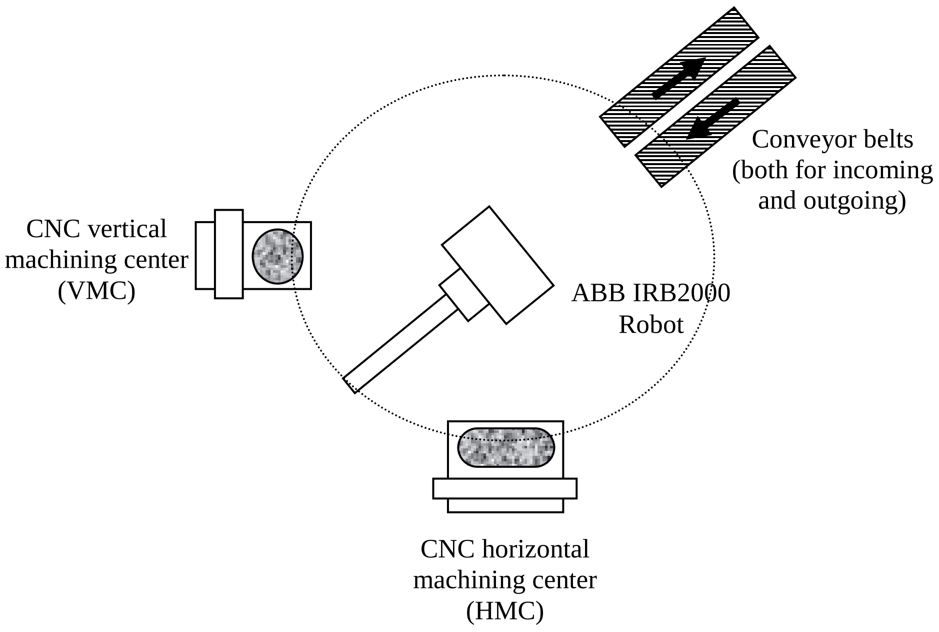


Figure-46. Flexible Manufacturing Cell at Narvik Institute of Technology (NIT)

Here is the operational specification of the system, somewhat simplified for our modeling purposes:

1. To start a cycle, a raw part must be available on the incoming conveyor belt, and the robot is also available.
2. The robot moves a raw part from the conveyor and loads it at the horizontal machining center (HMC).
3. The milling operation is performed at HMC while the robot backs off (returns).
4. The robot unloads the work piece from HMC, loads it to the vertical machining center (VMC) and returns.
5. The drilling operation is performed at VMC, and simultaneously the robot perform step 2.
6. The robot unloads the finished part from VMC, deposits it on the conveyor and returns.

In steady-state steps 2-6 repeat. Note that the specifications are very similar to the one given in Zhou and Robbi (1994). Well, it has to be similar, considering the simple systems we and they have, there is only one way to do it.

26.1.1 The Petri net model

The Petri net model for flexible manufacturing cell at NIT is given in figure 13. It is possible that one could come up with a slightly different model for the same system than the one shown in figure 13.

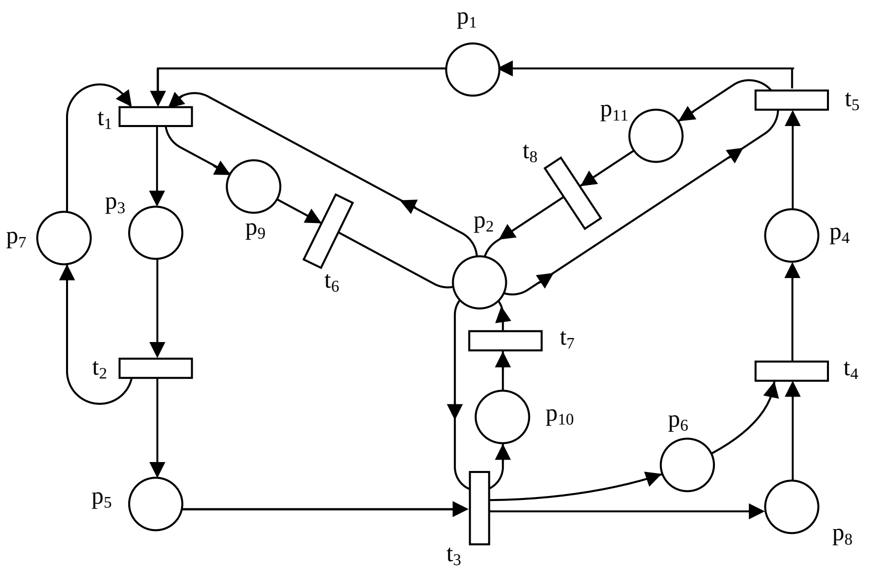


Figure-47. Timed PN model for flexible manufacturing cell at NIT.

26.1.2 The Petri net model

The upper arm of the model consisting place p_1 is the start mode. The left arm of the model is for the milling operation at HMC, the right arm is for the drilling operation at VMC, the bottom arm is for the transition between these two operations, and finally the central part is for the robot movements. Table-III shows the meaning of the different places and transitions.

Table-III: Meanings of places and transitions for the PN model.

place	*interpretation*	*trans*	*interpretation*	*time*
p_1	Raw parts	t_1	Robot/part to HMC	1
p_2	Robot available	t_2	Milling operation	*vary*
p_3	Part loaded to HMC	t_3	Robot/part to VMC	1
p_4	Out buffer VMC	t_4	Drilling operation	*vary*
p_5	Out buffer HMC	t_5	Robot/part to output	1
p_6	Part loaded to VMC	t_6	Robot returns	0.5
p_7	HMC available	t_7	Robot returns	0.5
p_8	VMC available	t_8	Robot returns	0.5
p_9	Robot ready return		(specimen operation times are given in minutes)	
p_{10}	Robot ready return			
p_{11}	Robot ready return			

It must be noted that there are potentials for parallel operations. For example, after loading a part into the HMC, while the milling operation is going on, the robot can retreat to its ready position, and also load a part from the output buffer of HMC into VMC(t_1 and t_2 are parallel).

26.1.3 Simulations

Lets vary the machining times of both milling and drilling operations and see for what combination of operations robot is overloaded (a second robot should be commissioned).

	Milling operation			
Drilling op.	0.3	0.5	1.0	5.0
0.3	100%	100%	90%	50%
0.5	100%	100%	90%	50%
1.0	90%	90%	82%	47%
5.0	50%	50%	47%	33%

Table-IV: Robot usage for different operation times.

!!!!!

27. Norwegian Traffic Lights

As shown in the figure below, Norwegian traffic lights have 4 states:
Red -> Red & Yellow -> Green -> Yellow

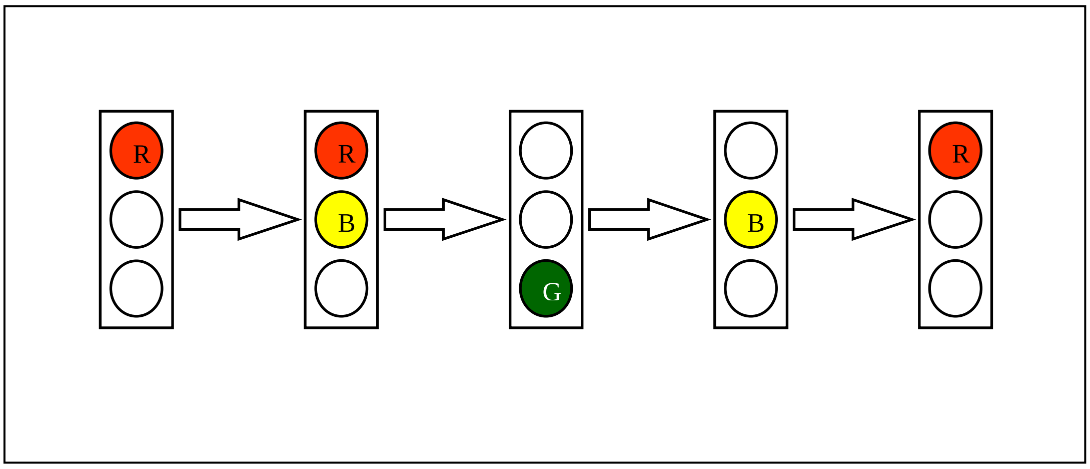
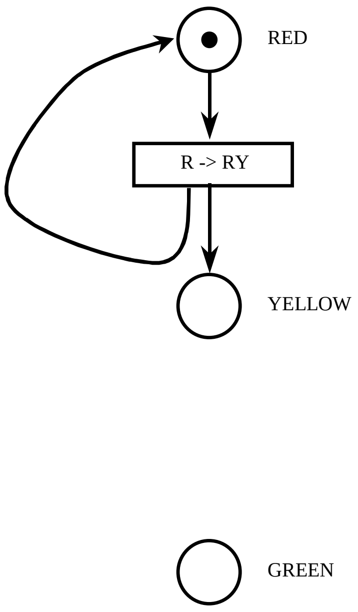


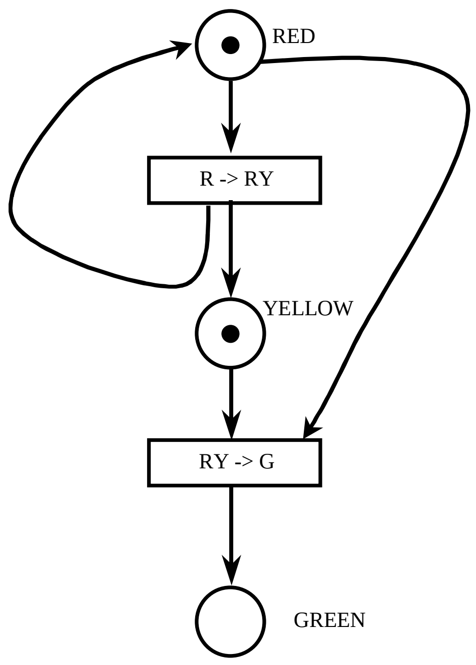
Figure-48. Norwegian Traffic Lights

27.1 Developing a Petri Net Model for Norwegian Traffic Light

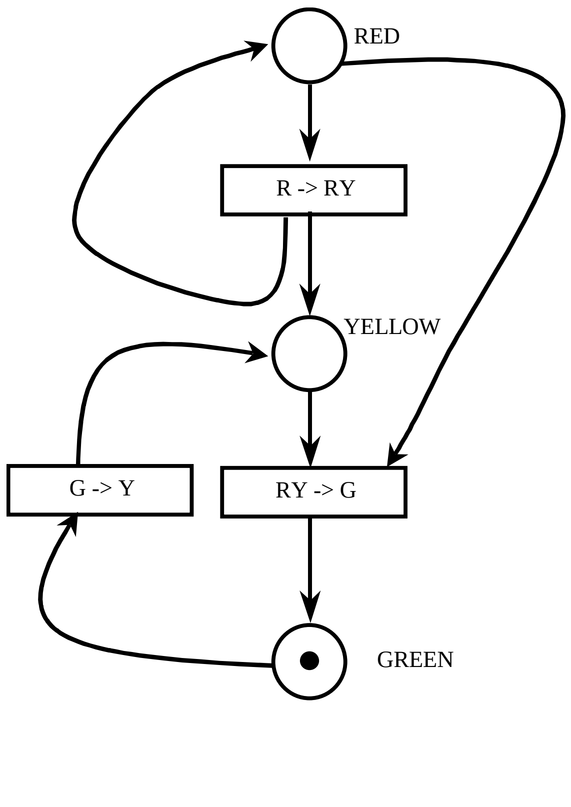
27.1.1 State-1 (RED) to State-2 (RED & YELLOW)



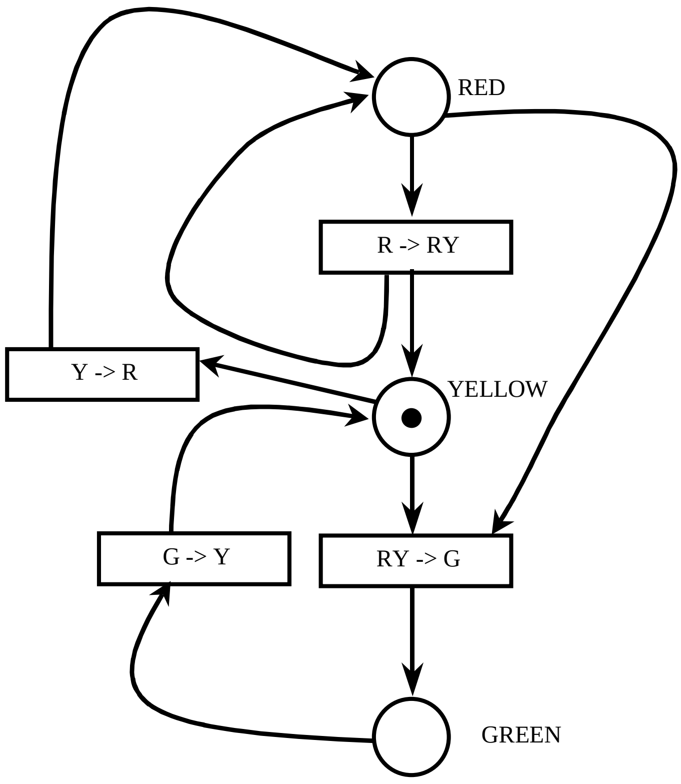
27.1.2 State-2 (RED & YELLOW) to State-3 (GREEN)



27.1.3 State-3 (GREEN) to State-4 (YELLOW)



27.1.4 State-4 (YELLOW) to State-1 (RED)



27.2 Transition Definitions

State-1 (RED) to state-2 (RED & YELLOW):

Transition tR->RY will fire only if there is a token in place RED and there is no token in place YELLOW (if there are tokens in both places, then tRY->G will fire)

State-4 (YELLOW) to state-1 (RED):

Transition tY->R will fire only if there is a token in place YELLOW and there is no token in place RED

27.3 Program Code for the Petri Net Model

27.3.1 Main Simulation File

```
% the main file to run simulation
clear, clc;
global_info.MAX_LOOP = 5; % stop after 5 states (one cycle)

pn = petrinetgraph('NO_light_def');
dynamic_info.initial_markings = {'pRED', 1};

Results = gpensim(pn, dynamic_info, global_info);
print_statespace(Results);
plotp(Results, {'pRED', 'pYELLOW', 'pGREEN'});
```

27.3.2 PDF

```
function [PN_name, set_of_places, set_of_trans, set_of_arcs]...
        = NO_light_def(global_info)
% file: pn_def.m:
% definition of petri net graph for Norwegian trafic lights

PN_name='Pet Net graph for trafic light (NOR)';
set_of_places={'pRED', 'pYELLOW', 'pGREEN'};

set_of_trans={'tR_RY','tRY_G','tG_Y','tY_R'};

set_of_arcs={'pRED','tR_RY',1, 'tR_RY','pRED',1, 'tR_RY','pYELLOW',1,...
    'pRED','tRY_G',1, 'pYELLOW','tRY_G',1, 'tRY_G','pGREEN',1,...
    'pGREEN','tG_Y',1, 'tG_Y','pYELLOW',1,...
    'pYELLOW','tY_R',1, 'tY_R','pRED',1};
```

27.3.3 TDF: tR_RY

```
function [fire, new_color, override, selected_tokens, global_info] = ...
    tR_RY_def(pn, new_color, override, selected_tokens, global_info)
% function fire = tR_RY_def(PN)

pR = get_place(pn, 'pRED');
pY = get_place(pn, 'pYELLOW');

tRRY = get_trans(pn, 'tR_RY');

fire = (pR.tokens) & not(pY.tokens);
```

27.3.4 TDF: tY_R

```
function [fire, new_color, override, selected_tokens, global_info] = ...
    tY_R_def(pn, new_color, override, selected_tokens, global_info)
% function fire = tY_R_def(PN)

pR = get_place(pn, 'pRED');
pY = get_place(pn, 'pYELLOW');
fire = not(pR.tokens) & (pY.tokens);
```

Part-III: Reference Manual

28. Design of the GPenSIM Simulator

In this section, we will look into the internals of GPenSIM simulator. Like any simulators, GPenSIM also has the following two major components: a (global) timer, and a queue to keep firing transitions (active events); in addition, GPenSIM also has mechanisms ('functions') to manipulate these two components - a push function to push firing transitions into queue, and a pop function to eject a firing transition from queue in order to complete (or finish) firing.

28.1 The Main Loop

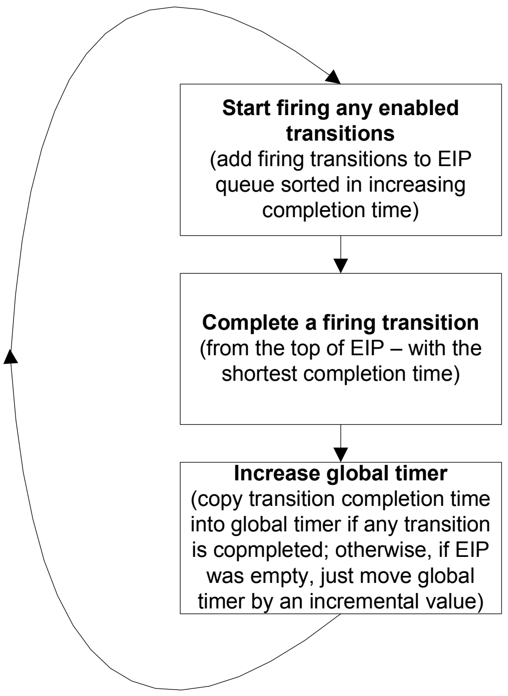
Components in the main loop:

- A Global Timer ("pn.current_time")
- A Queue ("EIP" – events in progress)

Mechanisms (functions) that manipulate the components:

- Pushing firing transition**s** into Queue (function 'start_firing')
- Popping **a** firing transition from Queue, in order to complete it (function 'complete_firing')

The components and the functions are realized in the M-file "**timed_pensim.m**". Figure-37 shown below summarizes the main loop realized in the M-file "timed_pensim.m":



Start firing any enabled transitions
(add firing transitions to EIP queue sorted in increasing completion time)
Complete a firing transition
(from the top of EIP – with the shortest completion time)
Increase global timer
(copy transition completion time into global timer if any transition is copmpleted; otherwise, if EIP was empty, just move global timer by an incremental value)

Figure-49. Simplified main loop of the simulation

However, actual coding of M-file "timed_pensim.m" is little more complicated due to the processing of stochastic systems, as shown in the following figure.

Figure-38 presents the actual loop for simulation, coded in the M-file "timed_gpensim.m".

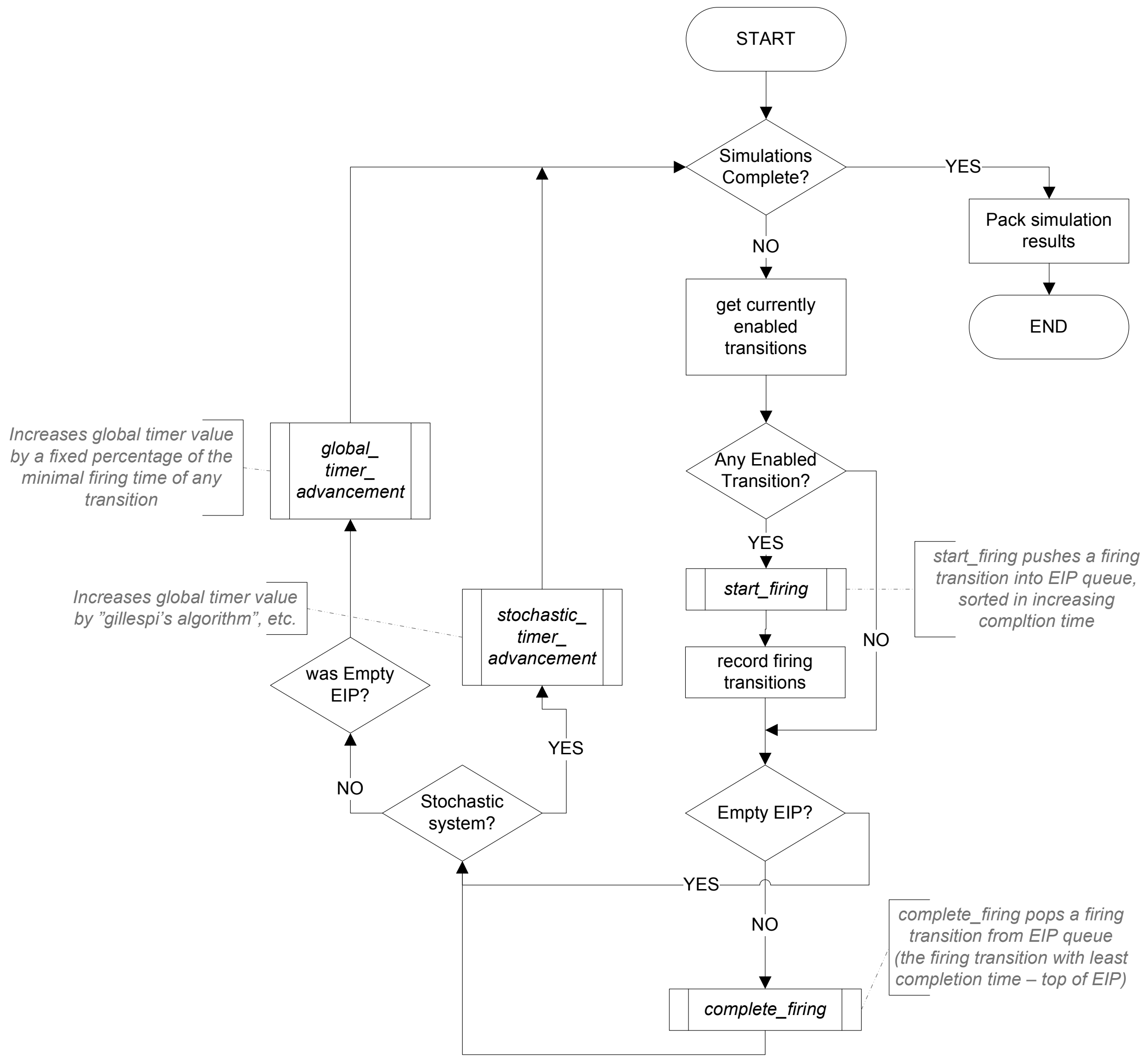

START
Simulations Complete?
YES
Pack simulation results
END
NO
get currently enabled transitions
Any Enabled Transition?
YES
start_firing
start_firing pushes a firing transition into EIP queue, sorted in increasing compltion time
record firing transitions
NO
Empty EIP?
YES
NO
complete_firing
complete_firing pops a firing transition from EIP queue (the firing transition with least completion time – top of EIP)
Stochastic system?
YES
NO
stochastic_ timer_ advancement
Increases global timer value by ”gillespi’s algorithm”, etc.
was Empty EIP?
global_ timer_ advancement
Increases global timer value by a fixed percentage of the minimal firing time of any transition

Figure-50. Main loop for simulation

29. Further Work (Future Extensions)

There are numerous possibilities for extending GPenSIM. We give blow just two:

- Adaptive GPenSIM: a version of GPenSIM in which the arc weights are not fixed and can vary during the simulation run.
    - Self adaptive: In each TDF, the arc weight of the transition can be changed.
    - Forced adaptive: in a specific TDF, arc weights of any transition can be varied

- Real-time ("soft PLC") simulator: Instead of global timer, the real-time clock of the computer can be used. In this case, the GPenSIM is no longer just a simulator, but it becomes a soft Programmable Logic Controller.

30. Data structures in GPenSIM

GPenSIM uses data structures to pass information between the functions. Some of the structures:

1. Structure for Petri net (PN): there are two Petri net structures
   a. Static PN structure is created by the function **petrinetgraph**
   b. Run-time PN structure is available during simulation; copy of run-time PN structure is available in TDFs.
2. Structure for Place: this structure is created by the function **place**
3. Structure for Transition: this structure is created by the function **transition**
4. Structure for Arc: this structure is created by the function **arc**
5. Structure for Token: tokens are removed (consumed) and added (deposited) during simulations
6. Structure for simulation results: this structure is created by the function **gpensim**
7. Structure for Co-Tree: this structure is created by the function **cotree**
8. Structure for Co-Tree: this structure is created by the function **gpensim**

30.1 Static Structures for Petri net and its elements

In order to inspect these structures, let us visit the example given in section-3 again. The program code snippet given below shows the main simulation file:

```
pn = petrinetgraph('simple_pn_def');
dynamic_info.initial_markings = {'Place-1',1, 'Place-2',2};
dynamic_info.firing_times = {'Transition-1', 10};

Sim_Results = gpensim(pn, dynamic_info);
print_statespace(Sim_Results);
```

After execution of the first line of the program snippet given above, the function **gpensim** returns the Petri net structure as an output variable called ‘pn’. Lets inspect this variable:

```
>> pn

pn =
                  name: 'A Simple Petri Net implementation'
         global_places: [1x3 struct]
    global_transitions: [1x1 struct]
           global_arcs: [1x3 struct]
      incidence_matrix: [1.00 2.00 0 0 0 1.00]
```

Screen dump given above shows that the Petri net structure has 7 elements. The elements are:

1) name: the ASCII string identifier of the Petri net
2) global_places: the set of all places in the Petri net
3) global_transitions: the set of all transitions in the Petri net
4) global_arcs: the set of all arcs in the Petri net
5) incidence_matrix: the matrix that depicts how the places and transitions are connected together, and
6) type: (not used)

Let us study the elements and their respective data structures one by one:

30.1.1 name

Name is an ASCII string identifier for the Petri net. From the screen dump given above, we already know the name of the Petri net, which is `'A Simple Petri Net implementation'`. We can also inspect the name anytime by typing Sim_Res.name:

```
>> Sim_Res.name

ans =
A Simple Petri Net implementation
```

30.1.2 global_places

global_places is the set of all places in the Petri net. Let's inspect the **global_places** by typing Sim_res.global_places:

```
>> Sim_Res.global_places

ans =
1x3 struct array with fields:
    type
    name
    tokens
    max_capacity
```

The screen dump given above shows that there are three places inside the **global_places** ([1X3]), and that each place has the following elements: type, name, tokens, and max_capacity. Let's inspect the places individually: The first place:

```
>> Sim_Res.global_places(1)

ans =

            type: 'place'
            name: 'Place-1'
          tokens: 0
    max_capacity: Inf
```

The first place is identified by its name as 'Place-1'. It has no tokens at the simulation end. The element 'max_capacity' is NOT USED.

We can also inspect a place by passing its identifier to the function 'get_place':

```
>> p1 = get_place(Sim_Res, 'Place-1')

p1 =
            type: 'place'
            name: 'Place-1'
          tokens: 0
    max_capacity: Inf
```

30.1.3 global_transitions

global_transitions is the set of all transitions in the Petri net. **global_transitions** can be studied by the same approach applied to inspecting **global_places**.

30.1.4 Global_arcs

global_arcs is the set of all arcs in the Petri net.

```
>> Sim_Res.global_arcs

ans =
1x3 struct array with fields:
    type
    from
    to
    weight
    name
```

Screen dump shows that **global_arcs** consists of three arcs. Let's inspect the first arc of **global_arcs**:

```
>> Sim_Res.global_arcs(1)

ans =

      type: 'arc'
      from: [1x1 struct]
        to: [1x1 struct]
    weight: 1.00
      name: 'Arc.475'
```

The first arc of the set of arcs has 5 elements:

1. type: this element identifies the type ('arc') of the element as an arc
2. from: this element identifies *the source of the arc*
3. to: this element identifies *the destination of the arc*
4. weight: this element identifies *the weight of the arc* (the weight of the arc is 1)
5. name: an ASCII string identifier to the arc (a unique identifier is generated for every arc: the unique identifier for this arc is 'Arc.475')

Further let's inspect the source and the destination of this arc:

```
>> Sim_Res.global_arcs(1).from

ans =

            type: 'place'
            name: 'Place-1'
          tokens: 0
    max_capacity: Inf
```

The source of this arc is the place 'Place-1'. The destination of the arc is:

```
>> Sim_Res.global_arcs(1).to

ans =
              type: 'transition'
              name: 'Transition-1'
       firing_time: 100.00
       firing_cost: 0
  firing_condition: ''
       times_fired: 0
```

The destination of the arc is the transition ‘Transition-1’. Of course, figure 6 verifies the results.

30.1.5 incidence_matrix

The incidence matrix is a matrix that depicts how the places and transitions are connected together. GPenSIM uses a compact and unique format to convey this information. Incidence matrix in GPenSIM is actually two matrices put together:

```
incidence_matrix =  [input_incidence_matrix    output_incidence_matrix]
```

Please refer to any standard text on Petri nets to know the details of incidence matrix.

30.1.6 type

‘type’ identifies a Petri net type. A Petri net can be un-timed (no concern about the firing times of the transitions), timed, or stochastic (firing times are not deterministic).

30.2 Run-time Structures for Petri net and its elements

[Also discussed in the section on “TDF”].
Run-time Petri net structure is available in all TDFs. It consists of the following elements:

1	STATIC	PN.Name:	'TDF Example: Production facility'
2	**Run-time**	**PN.global_places:**	**[1x*n* struct]** A set of sturctures; one structure per place, consisiting the following: **type: 'place'** **name: 'p1'** **tokens: 3.00** **max_capacity: Inf** **token_bank: [1x3 struct]** Token_bank is also a set of structures - one for each token in the place - consisitng the following. **tokID: 1.00** **creation_time: 0** **color: {‘A’, ‘B’}**
3	**Run-time**	**PN.global_transitions:**	**[1x*m* struct]** A set of sturctures; one structure per transition, consisiting the following: **type: 'transition'** **name: 't1'** **firing_time: 10.00** **firing_cost: 0** **times_fired: 0**
4	STATIC	PN.global_arcs:	[1x6 struct]
5	STATIC	PN.incidence_matrix:	[3x8 double]
6	**Run-time**	**PN.current_time:**	**45.00**
7	**Run-time**	**PN.token_serial_number:**	**30.00**
8	**Run-time**	**PN.X:**	**[10.00 3.00 5.00 2.00]** (Current Markings)
9	**Run-time**	**PN.Firing_Transitions:**	**[0 1 1]** Transitions Firing *at the moment*; one bit per transition; 0 – not firing; 1 – firing
10	**Run-time**	**PN.Enabled_Transitions:**	**[1 0 0]** Transitions enabled at the start of the cycle (Apriori); one bit per transition; 0 – not enabled; 1 - enabled)

30.3 Structures for simulation results

Simulation results from the function **gpensim** are kept in a structure that has two elements:

1. type: 'simulation'
2. LOG: a matrix
3. Firing_Transitions: a matrix
4. Enabled_Transitions: a matrix
5. State_Diagram: a matrix
6. Place_Names: Block of strings
7. Transition_names: Block of strings

Matrices LOG, Firing_Transitions, and Enabled_Transitions have same the number of rows. (Exception: for stochastic timer applications, LOG generally has less rows).

The *LOG* matrix can become large as it has all the simulation results. Each raw of *LOG* matrix represents changes due to firing of a transition, and has the following elements:

1. The new markings (the new state)
2. Fired transition
3. Parent state (matrix raw number) from which this state was obtained
4. Firing start time, and
5. Firing completion time

The *Firing_Transitions* matrix contains information about all the firing transitions at each inspection time. Each row of the *Firing_Transitions* starts with inspection time (element 1), and then rest of the elements are represents transitions; if element is '1' then the corresponding transitions was firing at the inspection time.

Similarly, the *Enabled_Transitions* matrix contains information about all the enabled transitions at each inspection time. Each row of the *Enabled_Transitions* starts with inspection time (element 1), and then rest of the elements are represents transitions; if element is '1' then the corresponding transitions was enabled at the inspection time.

State_Diagram represents sequences of states and the transitions that make state changes. State_Diagram is used by the print system (**'print_statespace'**). **NB: State_Diagram is also designed for making off-line graphical simulations; explained in the following subsection.**

State_Diagram consists of three different types of information: Row-1 is the new state; Row-2 is the enabled transitions after the new state; Row-3 is the firing transitions after the new state; This is further explained in example given below.

Places_Names and Transition_Names are names of all the places and the transitions respectively.

30.4 Example-1

In order to inspect the structure for simulation results, let us visit a small example. The program code snippet given below shows the main simulation file:

```
png = petrinetgraph('simple_pn_def');
dynamic.initial_markings = {'p1',3, 'p2',5};
dynamic.firing_times = {'t1', 10.11};

[sim] = gpensim(png, dynamic, global_info);
```

```
print_statespace(sim);
sim.LOG
```

```
function [PN_name, set_of_places, set_of_trans, set_of_arcs]...
    = simple_pn_def(global_info)

PN_name = 'A Simple Petri Net definition';
set_of_places = {'p1', 'p2', 'p3'};
set_of_trans = {'t1', 10};
set_of_arcs =  {'p1', 't1', 1, 'p2', 't1', 2, 't1', 'p3', 1};
```

Let us inspect the structure sim_RESULTS:

```
>> sim

sim =
                   type: 'simulation'
                    LOG: [3x7 double]
     Firing_Transitions: [3x2 double]
    Enabled_Transitions: [3x2 double]
          State_Diagram: [9x6 double]
            Place_Names: [3x2 char]
       Transition_Names: 't1'
```

30.4.1 LOG

Type 'simulation' identifies that the structure was obtained by after simulation run, and was output by the function **gpensim**.

The LOG matrix is a 3 X 10 matrix containing the simulation results. The easiest way to understand the simulation results is to use the function **print_statespace**. However, we can inspect this structure on our own:

```
>> sim.LOG

ans =

   Columns (1:3)         Column 4    Col 5   Col 6   Col 7
 New state (marking)      Firing    Parent  Firing  Firing
                        Transition   state   Start    Stop
                                     (raw    Time     Time
                                     no.)
 3.00    5.00     0         0         0       0        0
 2.00    3.00    1.00      1.00      1.00     0      10.11
 1.00    1.00    2.00      1.00      2.00   10.11    20.22
```

30.4.2 Firing_Transitions and Enabled_transitions

Firing_Transitions represents status (firing or not) of all the transitions at different inspection times. The first element in each row is the inspection time, followed by the status of the transitions.

```
>> sim.Enabled_Transitions

ans =

            0          1.00
        10.00          1.00
        20.00             0
```

Row-1: at time 0, **t1** was enabled.
Row-2: at time 10, **t1** was also enabled.
Row-3: at time 20, **t1** was NOT enabled.

30.4.3 State_Diagram

```
>> sim.State_Diagram

ans =

0           0           3           5           0           0
0           0           0           0           0           1
0           0           0           0           0           1
10          1           2           3           1           0
10          0           0           0           0           1
10          0           0           0           0           1
```

```
20          1          1          1          2          0
20          0          0          0          0          0
20          0          0          0          0          0
```

EXPLANATION:

row no.1 (state info)

Time	**NOT USED**	**Initial State**			**NOT USED** number of cells = number of transitions
0	**0**	**3**	**5**	**0**	0

row no.2 (enabled transitions)

Time	**NOT USED** number of cells = (number of places + 1)				**Enabled Transitions**
0	0	0	0	0	**1**

row no.3 (firing transitions)

Time	**NOT USED** number of cells = (number of places + 1)				**Enabled Transitions**
0	0	0	0	0	**1**

Row no. 4 (state info)

Time	**Fired Transitions** (Transition that created the new state)	**New State**			**(not used)**
10	**1**	**2**	**3**	**1**	0

row no.5 (enabled transitions)

Time	**NOT USED** number of cells = (number of places + 1)				**Enabled Transitions**
10	0	0	0	0	**1**

row no.6 (firing transitions)

Time	**NOT USED** number of cells = (number of places + 1)				**Enabled Transitions**
10	0	0	0	0	**1**

Row no. 7 (state info)

Time	**Fired Transitions** (Transition that created the new state)	**New State**			**(not used)**
20	**1**	**1**	**1**	**2**	0

row no.8 (enabled transitions)

Time	NOT USED number of cells = (number of places + 1)				Enabled Transitions
20	0	0	0	0	**0**

row no.9 (firing transitions)

Time	NOT USED number of cells = (number of places + 1)				Enabled Transitions
20	0	0	0	0	**0**

Function ‘**print_statespace**’ uses the matrix **State_Diagram** to print out simulation results:

```
State:0    Time: 0
Initial State:
p1        p2        p3
 3         5         0
At time: 0  enabled transitions are:
 t1

At time: 0  firing transitions are:
 t1

State: 1    Time: 10
Fired Transition: t1
Current State:
p1        p2        p3
 2         3         1
At time: 10  enabled transitions are:
 t1
At time: 10  firing transitions are:
 t1

State: 2    Time: 20
Fired Transition: t1
Current State:
p1        p2        p3
 1         1         2
At time: 20  enabled transitions are:
At time: 20  firing transitions are:
```

Explanation:

Print_statespace lines	Equivalent row of the matrix ’State_Diagram’

`State:0    Time: 0`
`Initial State:`
`p1        p2        p3`
` 3         5         0`
	Row-1

`At time: 0  enabled transitions are:`
` t1`
	Row-2

`At time: 0  firing transitions are:`	**Row-3**

t1	
State: 1 Time: 10 Fired Transition: t1 Current State: p1 p2 p3 2 3 1	**Row-4**
At time: 10 enabled transitions are: t1	**Row-5**
At time: 10 firing transitions are: t1	**Row-6**
State: 2 Time: 20 Fired Transition: t1 Current State: p1 p2 p3 1 1 2	**Row-7**
At time: 20 enabled transitions are:	**Row-8**
At time: 20 firing transitions are:	**Row-9**

30.4.4 Place_Names and Transition_Names

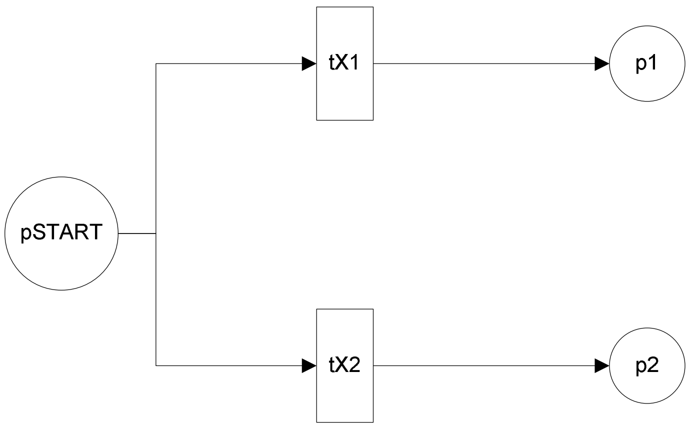
```
>> sim.Place_Names

ans =

p1
p2
p3
```

Since there is only 1 transition is the system,

```
>> sim.Transition_Names

ans =
```

30.5 Example-2 for State_Diagram

Figure shown below depicts a web server consisting of two server machines (**tX1** and **tX2**) that will fire alternatively. To allow alternative firing, we can implement a binary semaphore that can be read and manipulated by the definition files of both transitions.



MSF:

```
global_info.semafor = 1;    % GLOBAL DATA: binary semafor

png = petrinetgraph('loadbalance_def');
dynamicpart.initial_markings = {'pSTART', 10};
dynamicpart.firing_times = {'tX1', 10, 'tX2', 15};

sim = gpensim(png, dynamicpart, global_info);

plotp(sim, {'p1', 'p2'});
print_statespace(sim);
```

Let’s inspect the ‘**State_Diagram**’ element of the simulation results ‘**sim**’

```
>> sim.State_Diagram

ans =

      0      0     10      0      0      0      0
      0      0      0      0      0      1      1
      0      0      0      0      0      1      0
     10      1      9      1      0      0      0
     10      0      0      0      0      1      1
     10      0      0      0      0      0      1
     25      2      8      1      1      0      0
     25      0      0      0      0      1      1
     25      0      0      0      0      1      0
     35      1      7      2      1      0      0
     35      0      0      0      0      1      1
     35      0      0      0      0      0      1
     50      2      6      2      2      0      0
     50      0      0      0      0      1      1
     50      0      0      0      0      1      0
     60      1      5      3      2      0      0
     60      0      0      0      0      1      1
     60      0      0      0      0      0      1
     75      2      4      3      3      0      0
     75      0      0      0      0      1      1
     75      0      0      0      0      1      0
     85      1      3      4      3      0      0
     85      0      0      0      0      1      1
     85      0      0      0      0      0      1
```

```
    100     2     2     4     4     0     0
    100     0     0     0     0     1     1
    100     0     0     0     0     1     0
    110     1     1     5     4     0     0
    110     0     0     0     0     1     1
    110     0     0     0     0     0     1
    125     2     0     5     5     0     0
    125     0     0     0     0     0     0
    125     0     0     0     0     0     0

>>
```

Explanation:

Row-1: `[0    0    10    0    0    0    0]`
At time=0, the initial row shows the initial markings (at time 0)

Row-2: `[0    0    0    0    0    1    1]`
At time=0,, both **tX1** and **tX2** are enabled.

Row-3: `[0    0    0    0    0    1    0]`
At time=0, only tX1 is allowed to fire.

Row-4: `[10    1    9    1    0    0    0]`
tX1 (1) takes 10 minutes to fire. After **tX1** is fired, new state is [9 1 0]

Row-5: `[10    0    0    0    0    1    1]`
At time = 10, both **tX1** and **tX2** are enabled.

Row-6: `[10    0    0    0    0    0    1]`
At time = 10, only **tX2** is allowed to fire.

Row-7: `[25    2    8    1    1    0    0]`
When tX2 (2) completes firing, time moves from 10 to 25 seconds. The new state is [8 1 1].

Row-8: `[25    0    0    0    0    1    1]`
At time = 25, both **tX1** and **tX2** are enabled.

Row-9: `[25    0    0    0    0    1    0]`
At time = 25, only **tX1** is allowed to fire.

Row-10: `[35    1    7    2    1    0    0]`
When tX1 (1) completes firing, time moves from 25 to 35 seconds. The new state is [7 2 1].

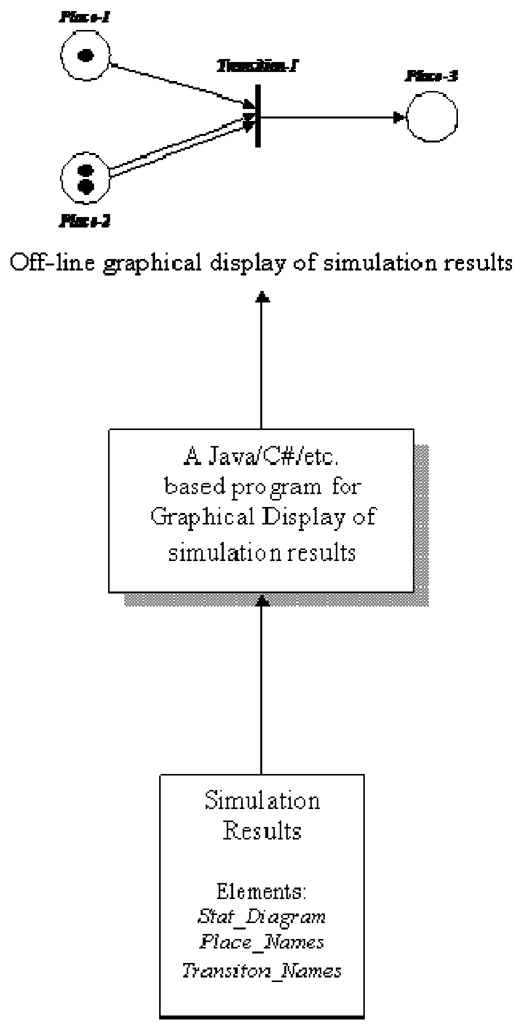
Row-11: `[35    0    0    0    0    1    1]`
At time = 35, both **tX1** and **tX2** are enabled.

Row-12: `[35    0    0    0    0    0    1]`
At time = 35, only **tX2** is allowed to fire.

…

30.6 Off-line Graphical Display

After simulations by the function ‘gpensim’, the simulation results has all the necessary information for off-line graphical display. The simulation results, lets call it ‘Sim_Results’, has three elements that can be used for graphical display (figure-32):



Place-1
Transition-1
Place-3
Place-2
Off-line graphical display of simulation results
A Java/C#/etc. based program for Graphical Display of simulation results
Simulation Results
Elements:
Stat_Diagram
Place_Names
Transiton_Names

Figure- 16: Off-line (after gpensim simulation) graphical display of simulation results step-by-step

Figure-51. Off-line graphical display

1. State_Diagram: a matrix
2. Place_Names: Block of strings
3. Transition_names: Block of strings

30.7 Structure for co-tree

Section 7.1 discusses obtaining co-tree of a Petri net. The program is given below:

```
% the main file to get the co-tree
png = petrinetgraph('fig_8_def');
sources = {'p1',1};
CT = cotree(png, sources);
print_cotree(CT); %
```

Execution of line 4 gives a structure called CT for the co-tree. Let us inspect this structure:

```
>> CT

CT =

    type: 'COTREE'
     LOG: [3x6 double]
```

The structure has two elements, element ‘type’ identifies that this structure is for co-tree, and the element ‘LOG’ has the rows of data for co-tree.

```
>> size(CT.LOG)

ans =

          3.00          6.00
```

The above screen dump shows that the LOG element is a 3 X 6 matrix. Only way to see co-tree properly is to feed the structure (CT) to function **print_cotree**.

30.8 Structure for colormap

Section 12.1 discusses colormap of a Petri net. The program is given below:

```
clear, clc;
pn = petrinetgraph('simple_adder_def');
dynamicpart.initial_markings = {'p1',1, 'p2',1};

[results, global_info, colormap] = gpensim(pn, dynamicpart);
…
```

Execution of line 4 gives a structure called colormap. Let us inspect this structure:

```
>> colormap

colormap =

    type: 'color_map'
     LOG: [1x5 struct]
```

The structure has two elements, element ‘type’ identifies that this structure is for colormap, and the element ‘LOG’ has the rows of data for colormap.

```
>> size(colormap.LOG)

ans =

          7.00          5.00
```

The above screen dump shows that the LOG element is a 7 X 5 matrix, meaning it has colors of 7 tokens. Colormap structure as an output of **gpensim** contains properties (color, creation time, and place) of all the tokens that were existed during simulation run. Let us see what the color of the first token is:

```
>> colormap.LOG(1)

ans =

     time: 0
    place: 4
    color: {'21', '45'}
```

The screen dump shows that the colors of the token were ‘21’ and ‘45’. We can see the colors of all the tokens that were involved during simulation, by feeding colomap structure to the function **print_colormap**.

31. Using MSF and petrinetgraph

Main Simulation File (MSF) calls at least three other GPenSIM functions directly:

- ‘petrinetgraph’
- ‘gpensim’, and
- ‘print_statespace’, ‘print_colormap’, ‘plotp’, etc.

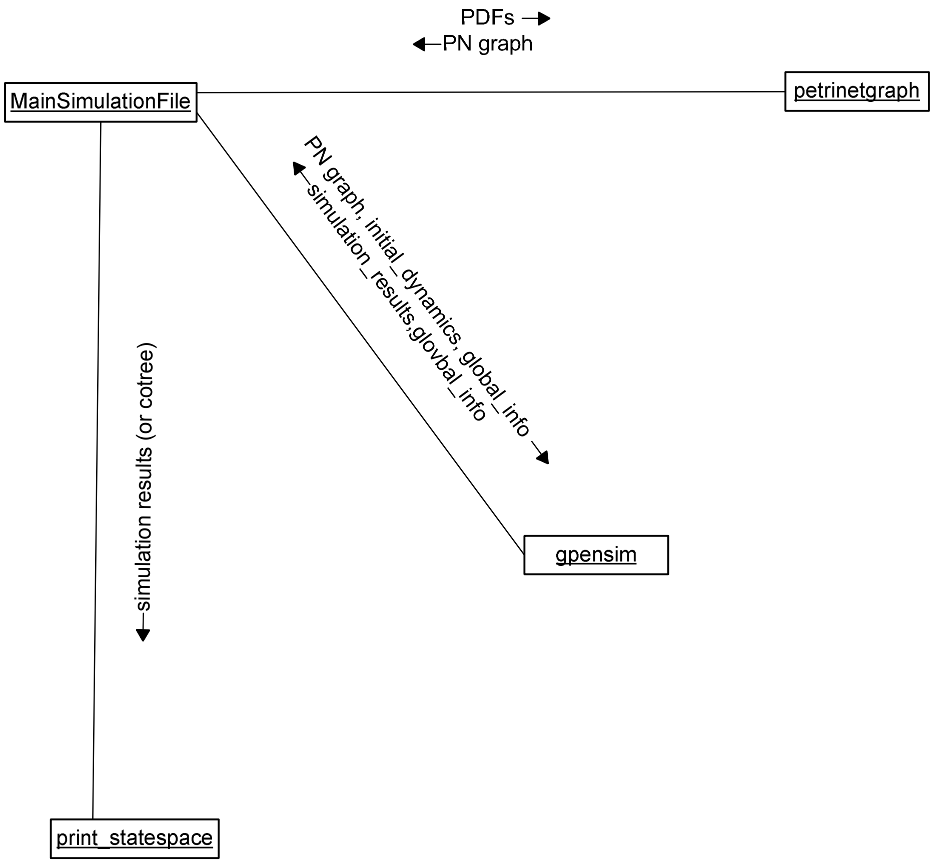

PDFs →
← PN graph
MainSimulationFile
petrinetgraph
PN graph, initial_dynamics, global_info →
← simulation_results,glovbal_info
gpensim
← simulation results (or cotree)
print_statespace

Figure-52. Collaboration Diagram for MSF

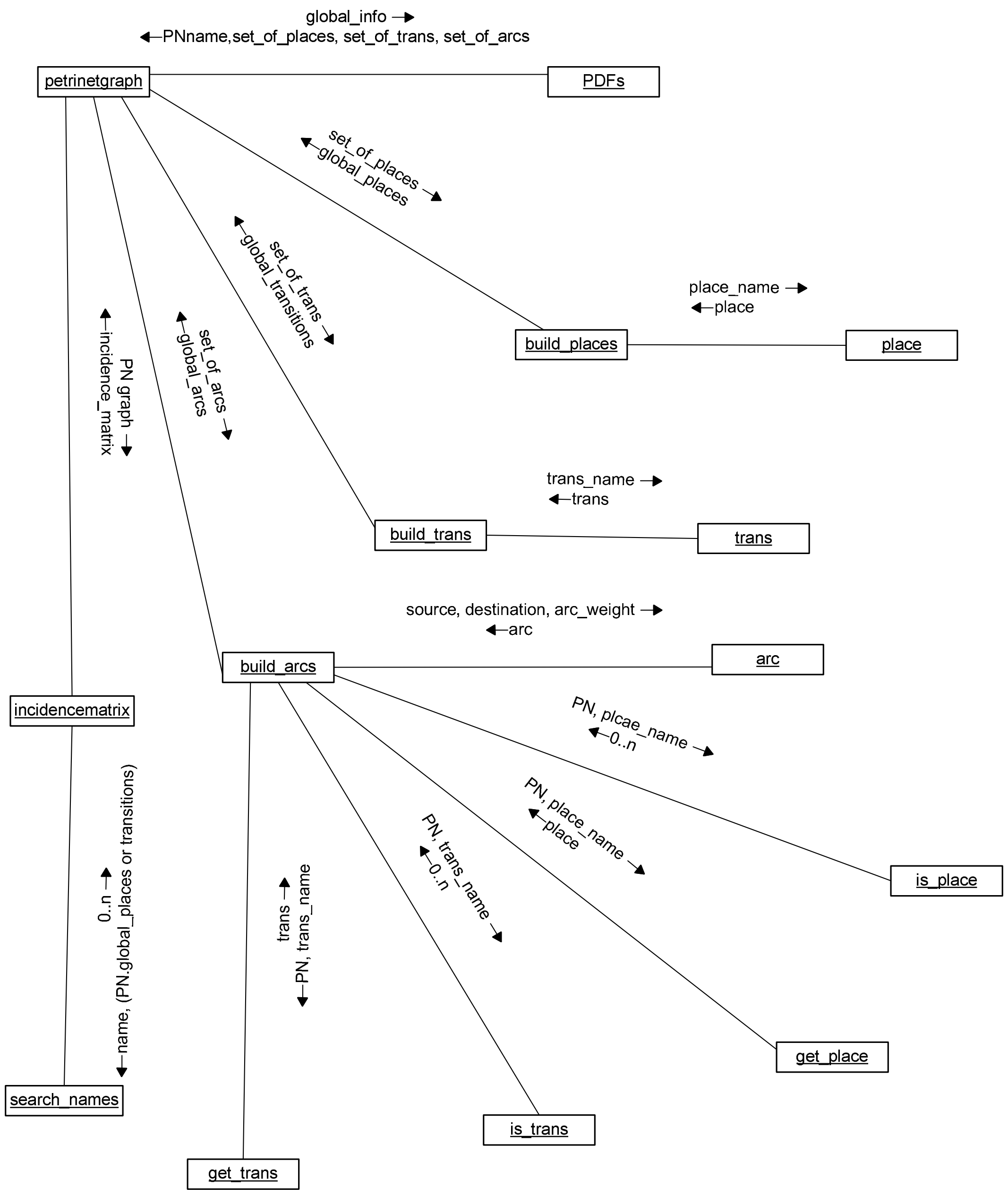


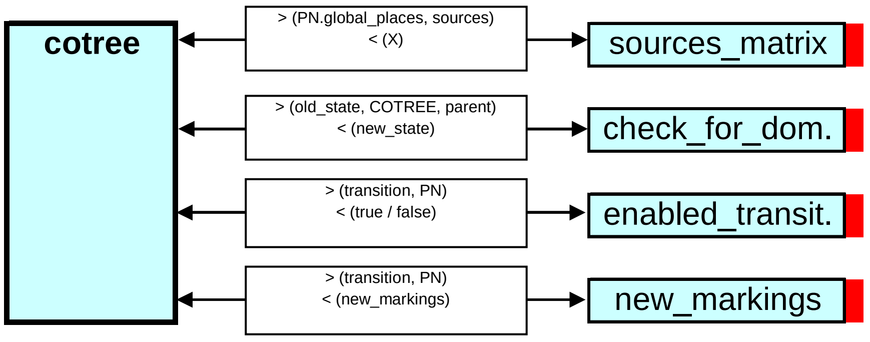
Figure-53. Collaboration Diagram for ‘petrinetgraph’

32. Description of the Main Functions

This section presents detailed description of some of the main GPenSIM functions. The following functions are described in detail: cotree, extractp, gpensim, gpensim_ver, MSF, PDF, petrinetgraph, plotp, print_cotree, print_finalcolors, print_statespace, timed_pensim, TDF.

Name:	**cotree**
Purpose:	Creates the coverability tree of a Petri net
Input parameters:	Static Petri net sturcture (the structure output by 'petrinetgraph') Intial_markings
Out parameters:	Cotree structure
Uses:	sources_matrix enabled_transition new_marking check_for_dominance good_name
Used by:	[main simulation file]
NOTE:	***Cotree algorithm is similar to the one by Cassandras & Lafortune (1998)***
Example:	

```
% in main simulation file
png = petrinetgraph('cotree_example_def');
dyn.initial_markings = {'p1',2, 'p4', 1};
cotree_sturcture = cotree(png, dyn.initial_markings);
print_cotree(cotree_sturcture);
```



Name:	**extractp**
Purpose:	To extract tokens from the Simulation results structure.
Input parameters:	Simulation Results (the structure output by ‘gpensim’) {set_of_place_names}
Out parameters:	TOKEN_MATRIX First row :[0 set_of_place_indices] Second & subsequent rows: [first column is time, other columns are tokens]
Uses:	None
Used by:	[main simulation file], Plotp
Example:	

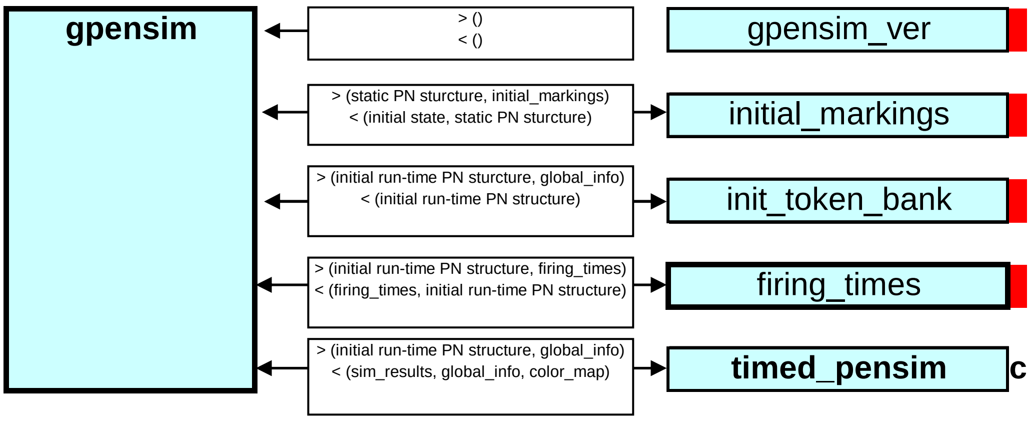
```
% in main simulation file
sim = gpensim(png, dynamic);
plotp(sim, {'p1','p2','p3'});
extractp(sim, {'p1','p2','p3'}) % print the token matrix
```

Name:	**gpensim**
Purpose:	To run simulations and output simulation results When the results are returned, they can be also analyzed (with tools like print_statespace, plotp, extract, occupancy, etc.)
Input parameters:	Static Petri net structure (output from ‘petrinetgraph’) initial dynamics global_info
Out parameters:	Simulation results global info
Uses:	gpensim_ver, initial_markings, init_token_bank, firing_times, state_space, **timed_gpensim**
Used by:	[main simulation file]
Example:	

```
% in main simulation file
[simualtion_Results, global_info] = gpensim(png, dyn, global_info);
print_statespace(simualtion_Results);
```

Name:	**gpensim_ver**
Purpose:	Prints the current version of gpensim
Input parameters:	None
Out parameters:	None
Uses:	None
Used by:	gpensim, [main simulation file]
Example:	

```
% in main simulation file
gpensim
% equivalently,
gpensim_ver
```


gpensim
> ()
< ()
gpensim_ver
> (static PN sturcture, initial_markings)
< (initial state, static PN sturcture)
initial_markings
> (initial run-time PN sturcture, global_info)
< (initial run-time PN structure)
init_token_bank
> (initial run-time PN structure, firing_times)
< (firing_times, initial run-time PN structure)
firing_times
> (initial run-time PN structure, global_info)
< (sim_results, global_info, color_map)
timed_pensim
c

Name:	**Main Simulation File (MSF)**
Purpose:	1. To declare global variables (global_info), 2. To load Petri net graphs (PDFs), and to create a static Petri net graph with the function 'petrinetgraph' 3. To assign initial dynamics, and 4. To start the simulation (with 'gpensim'). When the results are returned, they can be also analyzed (with tools like 'print_statespace', 'plotp', 'extractp', 'occupancy', etc.)
Input parameters:	-
Out parameters:	-
Uses:	petrinetgraph, gpensim, etc. tools like plotp, print_statespace, etc.
Used by:	-
Example:	

```
%%% FILE: MSF for MIC (mic_new.m)
global_info.LOOP_NUMBER = 1; %% print loop number during simulation

%%%% COMPOSE %%%%%%%
png = petrinetgraph({'client_def', 'internet_def',...
  'sil_def','conn_pro', 'iterate_def', 'strategy_def',...
  'tactic_def'});  %% 7 modules

%%%% DYNAMIC DETAILS %%%%
dyn.initial_markings = {'pSR',1, 'pNOI', round(unifrnd(2,4)), 'pB3',1};
dyn.firing_times = {'tCS','normrnd(5000,50)', 'tSC','normrnd(5000,50)',...
    'tINIT','unifrnd(280,320)',...
    'tRES','unifrnd(1, 10)', 'tSD','unifrnd(80, 100)',...
    'tTD','unifrnd(25, 35)', 'tSUB1','unifrnd(10, 15)',...
    'tSUB2','unifrnd(10, 15)', 'tSUB3','unifrnd(10, 15)',...
    'tSUB4','unifrnd(10, 15)'};
%%%% SUIMULATE %%%%%
RES = gpensim(png, dyn);
print_statespace(RES);
```

Name:	**Petri net Definition File (PDF)**
Purpose:	To define a static Petri net graph
Input parameters:	Optional: global_info
Out parameters:	PN_name: a text string of text, set_of_places: array of place structures set_of_trans: array of transition structures set_of_arcs: array of arc structures
Uses:	-
Used by:	Petrinetgraph
Example:	

```
function [PN_name, set_of_places, set_of_trans, set_of_arcs]...
                    = simple_adder_def(global_info)
%% PDF: simple_adder_def.m:

PN_name='Color example: Simple Adder';
set_of_places={'p1', 'p2', 'pNUM1', 'pNUM2', 'pADDED','pRESULT'};
set_of_trans={'tGET_NUM1','tGET_NUM2','tADD','tCONVERT'};
set_of_arcs={'p1','tGET_NUM1',1, 'tGET_NUM1','pNUM1',1,...
             'p2','tGET_NUM2',1, 'tGET_NUM2','pNUM2',1,...
             'pNUM1','tADD',1, 'pNUM2','tADD',1,...
             'tADD','pADDED',1, 'pADDED','tCONVERT',1, ...
             'tCONVERT','pRESULT',1};
```

Name:	**petrinetgraph**
Purpose:	To make a static Petri net structure from the Petri net definition file(s) (PDF(s))
Input parameters:	{ Names of One or more PDFs }
Out parameters:	Static Petri net structure
Uses:	build_places, build_trans, build_arcs, incidencematrix
Used by:	[main simulation file]
Example:	

```
% in main simulation file

% one PDF file
png = petrinetgraph('simple_pn_def');

% multiple PDF files
png = petrinetgraph({'client_def', 'internet_def',...
            'sil_def','conn_pro',...
            'iterate_def', 'strategy_def', 'tactic_def'});
```

Name:	**plotp**
Purpose:	To plot simulation results; to plot how tokens change with time
Input parameters:	Simulation Results (the structure output by ‘gpensim’) {set_of_place_names} global_info (optional)
Out parameters:	TOKEN_MATRIX (contains tokens of places with time)
Uses:	extractp (extracts tokens from the SIM results structure)
Used by:	[main simulation file]
Example:	

```
% in main simulation file
sim = gpensim(png, dynamic);
plotp(sim, {'p1','p2','p3'});
```

Name:	**print_statespace**
Purpose:	To print simulation results
Input parameters:	Simulation Results (the structure output by ‘gpensim’)
Out parameters:	None
Uses:	print_markings, print_statespace_enabled_trans, print_statespace_firing_trans print_statespace_state
Used by:	[main simulation file]
NOTE:	***Not for use with simulations using stochastic timer***
Example:	

```
% in main simulation file
Simulation_results = gpensim(png, dynamic);
print_statespace(Simulation_results);
```

Name:	**print_colormap**
Purpose:	To print colors of the tokens
Input parameters:	Simulation Results (the structure output by ‘gpensim’) {set_of_place_names}
Out parameters:	None
Uses:	print_colormap_for_place
Used by:	[main simulation file]
Example:	

```
% in main simulation file
results = gpensim(pn, dynamicpart);
print_colormap(results, {'pNUM1','pADDED', 'pRESULT'});
```

Name:	**print_finalcolors**
Purpose:	To print colors of the final state (colors of the tokens that are left in the system when the simulations are complete)
Input parameters:	Simulation Results (the structure output by 'gpensim')
Out parameters:	None
Uses:	None
Used by:	[main simulation file]
Example:	

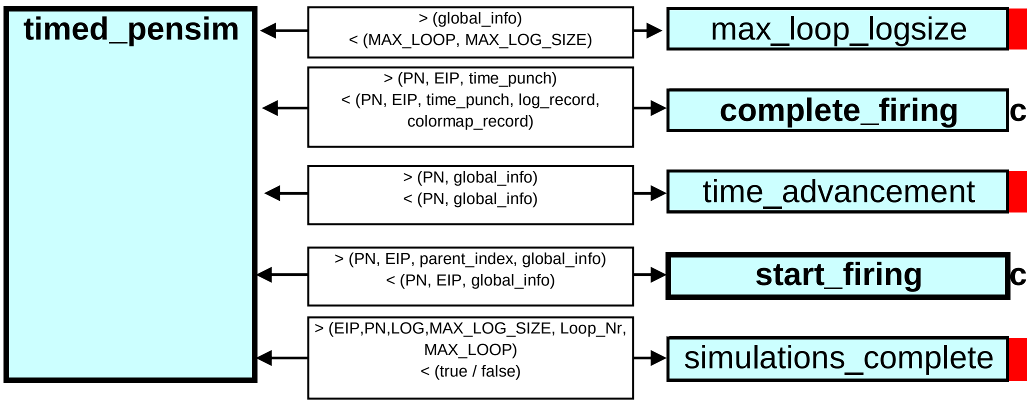
```
% in main simulation file
results = gpensim(pn, dynamicpart);
print_finalcolors(results);
```

Name:	**print_cotree**
Purpose:	To print cotree structure
Input parameters:	Cotree structure (the structure output by 'cotree')
Out parameters:	None
Uses:	print_markings
Used by:	[main simulation file]
Example:	

```
% in main simulation file
cotree_structure = cotree(png, dyn.initial_markings);
print_cotree(cotree_ structure);
```

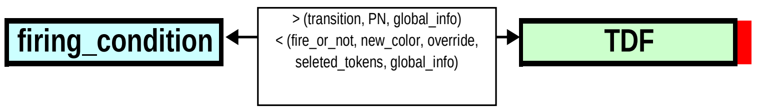
Name:	**timed_pensim**
Purpose:	This is the main M-function for Petri net simulation. Inside the main loop, transitions are randomly chosen and checked whether they are enabled or not. If they are enabled, the token removal and deposition in respective places happens. Then the happenings are recorded in the simulation results LOG.
Input parameters:	Static Petri net structure (output from 'petrinetgraph') global_info
Out parameters:	Simulation results global info
Uses:	max_loop, print_loop_nr, simulations_complete enabled_transition start_firing complete_firing stochastic_timer_advancement, global_timer_advancement pack_sim_results
Used by:	gpensim
Note:	***This is one of the most important M-files, as it realizes the main simulation loop***
Example:	

```
% inside gpensim
[sim_results, global_info] = timed_pensim(png, global_info);
```



Name:	**Transition Definition File (TDF)**
Purpose:	To run user-defined conditions, and to test probe simulation
Input parameters:	PN: run-time Petri net structure global_info : global info packet (Dummy variables: new_color = {}, override=false, selected_tokens=[])
Out parameters:	fire_or_not: fire (≠ 0), don't fire (=0) new_color: colors assigned by transition, override: override (≠ 0), don't override (=0), selected_tokens: tokIDs of any selected tokens for removal (consumption), global_info: updated (if updated by the transition) global info packet
Uses:	-
Used by:	Firing_conditions
Example:	

```
function [fire, new_color, override, selected_tokens,global_info] = ...
    tCONVERT_def (pn, new_color, override, selected_tokens,global_info)
%% TDF: tCONVERT_def

% first, select an token
tokID = select_token(pn, 'pADDED', 1);

% second, get the colors of the selected token
colors = get_color(pn, tokID);
num1 = str2num(colors{1}); % convert color into number
num2 = str2num(colors{2}); % convert color into number
sum = num1 + num2;
new_color = num2str(sum);
override = 1; % only sum as color - NO inheritance
global_info.sum = sum; %%% sum is added to global_info
fire=1;  %always fire
```


firing_condition
> (transition, PN, global_info)
< (fire_or_not, new_color, override, seleted_tokens, global_info)
TDF

REFERENCES

- C. G. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems*. Boston, MA: Springer Science+Business Media, LLC, 2007.

- **GPenSIM web page: http://www.davidrajuh.net/gpensim/**

- Darren J. Wilkinson, "Stochastic Modelling for Systems Biology", Chapman & Hall/CRC, NY, 2006. ISBN-10 1-58488-540-8. Read especially about Gillespi's algorithm in chapter 06.

- **[James D. Stein]**

- T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE,* vol. 77, pp. 541-580, 1989.

- R. Davidrajuh, "Event-driven simulation, modeling, and analysis with GPenSIM," *Communications of the IIMA (Published by the International Information Management Association),* vol. 3, pp. 53-71, 2003

- C. A. Petri and W. Reisig, "Petri net," *Scholarpedia*, vol. 3, p. 6477, 2008

- R. Davidrajuh and I. Molnar, "Designing a new tool for modeling and simulation of discrete event systems," *Issues in Information Systems*, vol. X, pp. 472-477, 2009

- Stateflow (2010) The MathWorks Inc, "Stateflow 7.4 - Design and simulate state machines and control logic," http://www.mathworks.com/products/stateflow/, 2010.

- K. Jensen, *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use,* 2. ed. vol. 1: Springer, 1997

- Zhou, M.C. and Robbi, A.D., 1994, "Application of Petri net methodology to manufacturing systems", Computer Control of Flexible Manufacturing Systems : Research and Development (Edited by : Joshi, S.B. and Smith, J.S.), **Chapman & Hall**, Hong Hong.

- Davidrajuh, R. (2007). "A Service-Oriented Approach for Developing Adaptive Distribution Chain", International Journal of Services and Standards (ISSN (Online): 1740-8857 - ISSN (Print): 1740-8849), Vol. 3, No.1, pp. 64 – 78.