A Tool for Modeling and Simulation of
Discrete-Event Systems

GPenSIM

General Purpose Petri Net Simulator

Version 9.0 © August 2014

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

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 three specific goals: 1) Modeling, simulation, analysis, and control of discrete event systems, 2) A tool that is easy to use and extend, and 3) allowing Petri net models to be integrated with other MATLAB toolboxes (e.g. Fuzzy Logic, Control systems).

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
January 2014

1. Installing GPenSIM

Installation takes five simple steps:

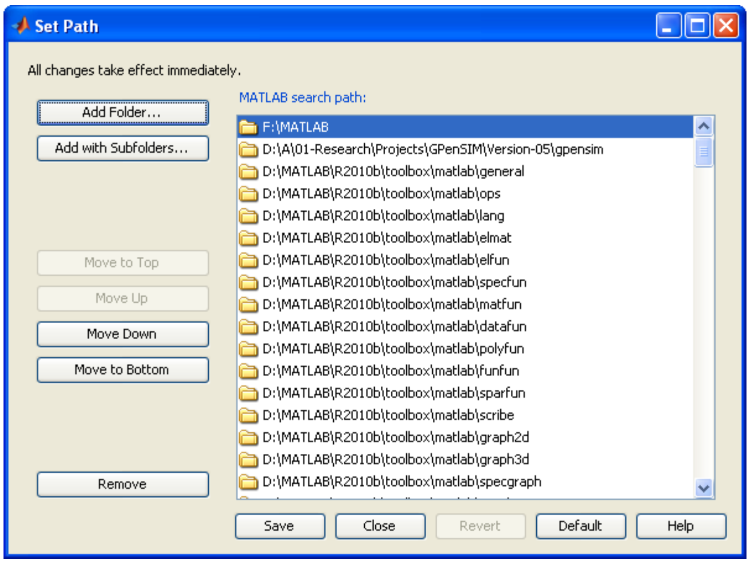
1. Unzip the GPenSIM pack:

Unzip the GPenSIM system M-files (toolbox functions) under a directory, say "d:\GPenSIM\GPenSIM80\". Note: Due to size limitations, there may be one zip file (GPenSIM-v8.0.zip) or two zip files (GPenSIM-v8.0-pack-1.zip and GPenSIM-v8.0-pack-2.zip) zip files.

Similarly, unzip the examples file (Examples-v8.0.zip) under another directory, say "d:\GPenSIM\Examples\"

2. Set MATLAB Path Command:

- Start MATLAB. Go to the file menu in MATLAB, and select "set path" command:



```
F:\MATLAB
D:\A\01-Research\Projects\GPenSIM\Version-05\gpensim
D:\MATLAB\R2010b\toolbox\matlab\general
D:\MATLAB\R2010b\toolbox\matlab\ops
D:\MATLAB\R2010b\toolbox\matlab\lang
D:\MATLAB\R2010b\toolbox\matlab\elmat
D:\MATLAB\R2010b\toolbox\matlab\elfun
D:\MATLAB\R2010b\toolbox\matlab\specfun
D:\MATLAB\R2010b\toolbox\matlab\matfun
D:\MATLAB\R2010b\toolbox\matlab\datafun
D:\MATLAB\R2010b\toolbox\matlab\polyfun
D:\MATLAB\R2010b\toolbox\matlab\funfun
D:\MATLAB\R2010b\toolbox\matlab\sparfun
D:\MATLAB\R2010b\toolbox\matlab\scribe
D:\MATLAB\R2010b\toolbox\matlab\graph2d
D:\MATLAB\R2010b\toolbox\matlab\graph3d
D:\MATLAB\R2010b\toolbox\matlab\specgraph
```

Figure 1-1: Set Path dialog

- Select "Add with Subfolders":

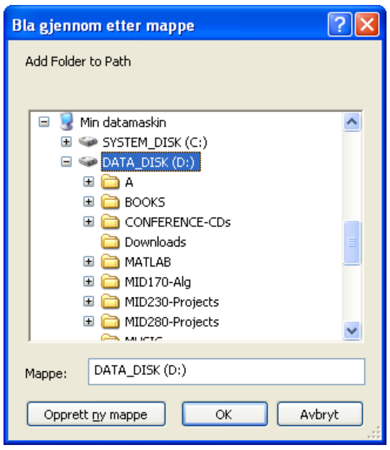


Figure 1-2: Add folder dialog

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.

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 9.0;   Lastupdate: A 2014
(C) Reggie.Davidrajuh@uis.no
http://www.davidrajuh.net/gpensim
--------
>>
```

2. Introducing Classic 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).

2.1 Elements of Classic Petri nets

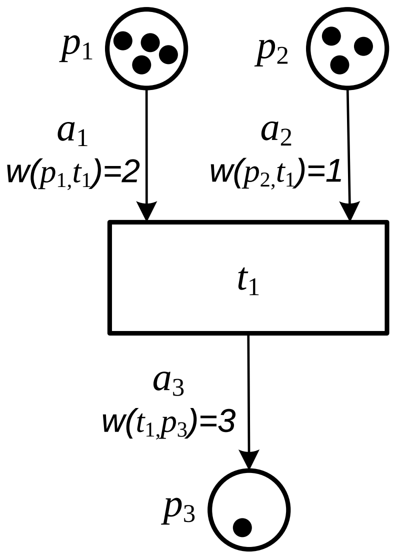


Figure 2-1: Sample Petri Net

A Petri net contain two types of elements: places and transitions; places generally represent passive elements (such as input and out buffers, conveyor belts, etc.) and transitions represent active elements (such as machines, robots, etc.). Petri net is a directed bipartite graph meaning a place can only be connected to transition(s) and a transition to place(s); the conections between places and transitions are termed as arcs.

In addition to places, transitions, and arcs, Petri Net also has tokens. Tokens represent objects that can flow around in a network of nodes, e.g. materials in a material flow system, data (or information) in an information flow. Places hold tokens; tokens move from place to place via the arcs. Tokens are shown as black spots in a Petri net. If a place has a large number of tokens, then it is customary to show the number of tokens with numberals than black spots.

The arcs that connect places to transitions and transitions to places have default weight of one. If an arc has a weight that is greater than unity, then the weight is shown above the arc. The arc weight represents the capacity of the arc to transport a number of tokens simultaneously at a time.

Figure-2.1 shows three places p_1, p_2 and p_3. These three places hold 4, 3 and 1 tokens, respectively. When a transition fire, a number of tokens are taken from the input places and new tokens are deposited into the output places; the arc weights determine the number of tokens taken and deposited. For a transition to be able to fire, the number of tokens in the input places must be equal or larger than the weights of the arcs connecting the input places to the transition. The transition will then becomes able to fire (*enabled transition*). Figure-2.2 shows the state of the sample Petri net from figure 1 after the transition *t1* has fired once.

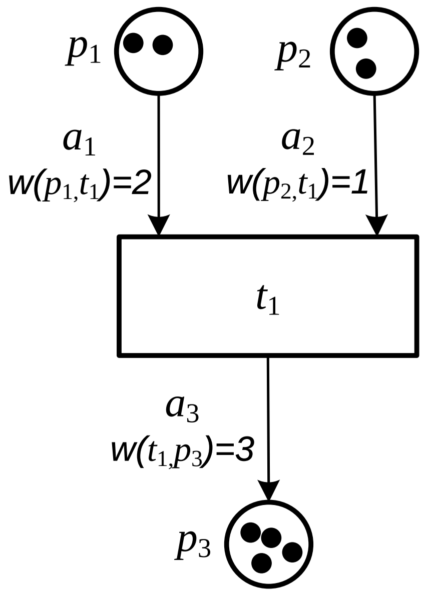


Figure 2-2: Petri Net after one firing

2.2 Formal Definition of Classic (or ordinary or P/T) Petri nets

A Petri net is a four-tuple (P, T, A, m_0)

Where,

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

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

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

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

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

$m = [m(p_1), m(p_2), \dots, m(p_{n1})] \in \mathrm{N}^n$, m_0 is the initial marking.

2.2.1 Input and Output Places of a Transition

In the Petri net in figure-2.2, the places p_1 and p_2 are input places 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): $m = [m(p_1), m(p_2), \dots, m(p_{n1})]$

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, $m(p_i) \geq m(p_i, t_j)$ for all $p_i \in I(t_j)$.
If $f(m, t_j)$ is defined then $m' = f(m, t_j)$, where
$m'(p_i) = m(p_i) - w(p_i, t_j) + w(t_j, p_i), \quad i = 1, \dots, n.$

2.4.1 Time and classic Petri Net

Classic Petri Net (also known as Ordinary, Pure, or P/T Petri Net) is untimed. In classic PN, all the transitions possess zero firing time, like Dirac delta function (impulse). Since transitions in classic Petri Net take zero time to fire, they are called "***primitive transitions***", as they can not represent any real event which always takes time. Though GPenSIM can be used for Classic Petri Nets, it is mainly for Timed Petri Nets. In Timed Petri Nets, ***all the transitions*** are considered to be "***non-primitive***", thus have finite (non-zero) firing times; may be some of the transitons have very small firing times, yet are not zero.

In GPenSIM, if firing times are not assigned to any of the transitions, then the system is assumed as a classic Petri Net. It is not acceptable to assign firing times to some of the transitions and let the other transitions take zero value; in other words, a systems can be either classic Petri Net (all transitions are untimed) or Timed Petri Net where all the transitions are assigned finite firing times (not zero).

2.4.2 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 Simple Petri net Model

The simple Petri net shown in figure-2.3 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.

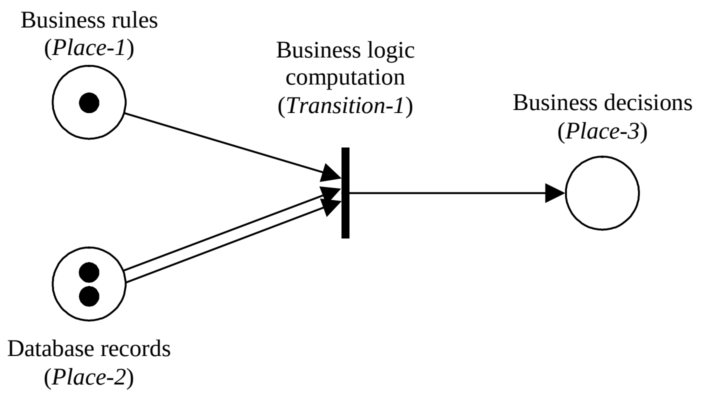


Figure 2-3: Petri Net model for business logic computations

3. Timed Petri Net

In GPenSIM, a discrete system can be either:

- **A classic Petri Net**: firing times are not assigned to any of the transitions, meaning all the transitions are **primitive**, or
- **Timed Petri Net**: firing times are assigned to all the transitions, meaning all the transitions are **non-primitive**.

It is not acceptable to assign firing times to some of the transitions and let the other transitions take zero value; we may assign very small values close to zero, but not zero, to transitions that are very fast compared to the other transitions. GPenSIM interprets Timed Petri Net followingly:

- **No variable duration of event**: the transitions representing events are given fixed timing before hand (timing defined exactly). Though fixed, the timing can be deterministic (e.g. firing time dt = 5 TU) or stochastic (e.g. firing time dt is normally distributed with mean value 10 TU and standard deviation 2 TU (‘normrnd(10,2)’). However, variable firing times are not acceptable.
- ***Maximal-step* firing policy**: just like classic PN, Timed Petri Net also operates with the maximal-step firing policy; maximal-step firing policy means if more than one transition is enabled at a point of time collectively, then all of them fire at the same time.
- *Enabled* transition start *firing* immediately: enabled transition can start firing immediately as there is no (forcibly) induced delay between the time a transiton became enabled and the time it is allowed to fire.

3.1 Atomicity and Virtual Tokens

The figure-3-1 given below explains how non-primitive transition $\boldsymbol{t_i}$ of a Timed Petri Net (firing time of the transition is not zero) can be understood in terms of primitive (firing time is zero) transitions of classic Petri Nets. As figure-3-1 shows, each non-primitive transiton in Timed Petri Net can be considered as a group of two primitive transitions *starter* and *stopper*, and a *virtual place* between them; in addition, there is a place pme with an initial token (pme: place to impose mutual exclusion) in order to make sure that once starter has fired, it will not fire again until stopper is fired.

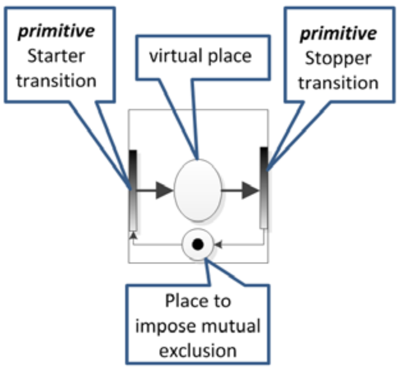
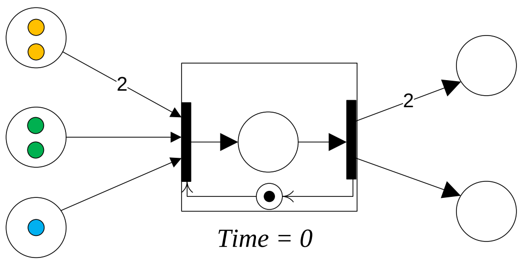
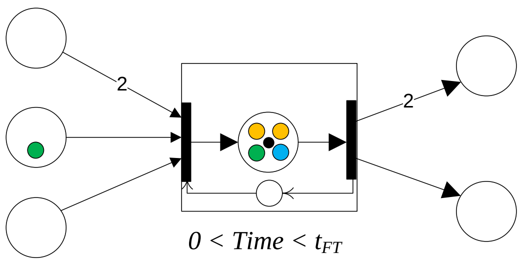

primitive Starter transition
virtual place
primitive Stopper transition
Place to impose mutual exclusion

Figure 3-1: Composition of a non-primitive transition.

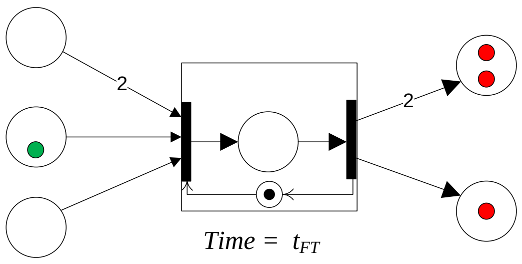
Figure-3-2 explains the firing of $\boldsymbol{t_i}$. Whenever $\boldsymbol{t_i}$ is ready to fire, *starter* fires immediately and passes the input tokens into the virtual place; the input tokens will stay in the virtual place for an amount of time (delay) equal to the firing time of $\boldsymbol{t_i}$. At the completion of the delay, the stopper fires immediately, consuming all the virtual tokens, and depositing newer output tokens into the output places.



2a) Non-primitive transition is enabled and about to fire



2b) Non-primitive transition starts firing: primitive starter fires instantly, removing the tokens from the input place and placing them as virtual tokens inside the virtual place



2c) Non-primitive transition completes firing: primitive stopper transition fires instantly, removing the virtual tokens from the virtual place and depositing output tokens into the output place

Figure 3-2: Maintaining the 'atomicity' property during the firing of a non-primitive transition

The firing mechanism described above makes sure that the tokens are accountable (do not disappear) anytime during the firing of the non-primitive transition $\boldsymbol{t_i}$. Thus, atomicity property is upheld.

3.2 Defining Timed Petri Net (TPN)

Timed Petri Nets (TPN) are classical Petri nets superimposed with time for each transition.

Definition: *A Timed Petri Net is a 6-tuple TPN* = (P, T, A, m_0, D), where:

> *1. PN*(TPN) = (P, T, A, m_0) *is a classic Petri Net and*
> *2. D*: $T \rightarrow R^+$ is the *duration function*, a mapping of each transition into a positive rational number, meaning firing of each transition t_i now takes exactly dt_i time units.

Part-I: GPenSIM Basics

4. Modeling with GPenSIM: The Basics

In GPenSIM, definition of a **Petri net graph** (***static*** details) is given in the **Petri net Definition File**

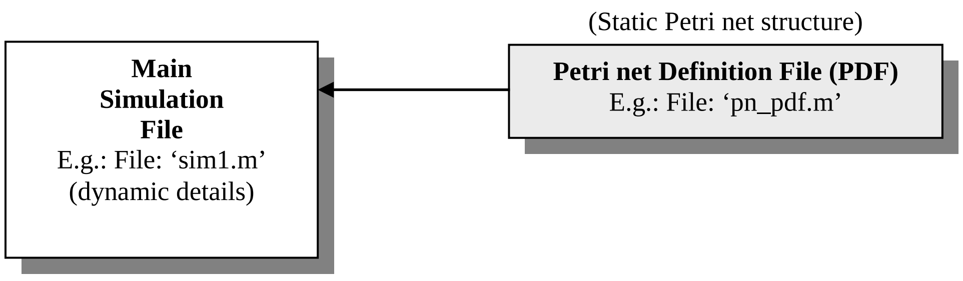


Figure 2: Separating the ***static*** and dynamic Petri net details

(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. Whilst 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 4-1: Separating the static and dynamic Petri net details

4.1 Pre-processors and Post-processors

In addition to these two files (main simulation file - MSF and Petri net definition file - PDF), there can be a number of pre-processors and post-processors.

A pre-processor contains the code for additional conditions to check whether an enabled transition can actually fire; in other words, pre-processor is run before firing a transition, just to make sure that an enabled transition can actually start firing depending upon some other additional conditions ('firing conditions'). Futher, we can write separate pre-processors for each transition or combine them into a signle common pre-processor. Combining individual Pre-processors and common pre-processor is also acceptable.

A post-processor is run after firing of a transition. A post-processor contains code for actions that has to be carried out after a certain transition complets firing. Just like pre-processors, post-processors can be specifically for individual transitions or combined into one common post-processor.

NB: In the older versions of GPenSIM (also in somepalces in this document) pre-processor and post-processor are collectively called as Transiton Definiotion Files (TDF); moreover pre-processor is called TDF_pre and post-processor as TDF_post.

4.1.1 Using pre-processor and post-processor

According to the Petri net theory, a transition can fire ("enabled transition") when there are enough tokens in the input places. However, 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 when both event-1 and event-2 are enabled to fire, and are competing for the same resource. In GPenSIM literature, these additional restrictions are called "**firing conditions**".

The firing conditions for firing a transition are coded in a pre-processor file. *After a transition completes firing*, there may be some book keepings need to be done or some other activities need to be performed; these can be coded into a post-processor file.

CAUTION: Names of the processor files must follow a strict naming policy, as they will be chosen and run automatically: for example, the specific pre-processor for the transition 'trans1' must be named 'trans1_pre.m'; similarly, the specific post-processor for the transition 'trans1' must be named 'trans1_post.m'.

4.1.2 Using pre-processor and post-processer as a test probe

In addition to executing firing conditions, a pre-processor (and post-processor) provides a unique functionality: acting as a probe to simulation engine: Let us explain:

1. The role of **PDF** (Petri net Definition Files): the only use of a PDF is to define a static Petri net graph.
2. The role of **MSF** (Main Simulation File): A PDF will be loaded into memory by MSF right before the simulation start. Thus, an MSF first loads PDF (or PDF**s**, each representing a module) into the workbench and then starts the simulation. MSF will be blocked during the simulation runs; 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 <u>**during**</u> **simulation**.
3. The role of pre- and post-processors: Though MSF does not have any control of what going on <u>**during**</u> **simulation**, however, pre- and post-processors will be called during simulation, *before and after transition firings*: a pre-processor is to check whether firing conditions of a particular transition are met, and a post-processor is to do some post-firing activities if needed, after the particular transition has fired. Since these are called during the simulations, these can be used to inspect the system run-time; more details given in the section on pre- and post-processors.

4.2 Global info

The different files (main simulation file MSF, Petri net definition files PDFs, and pre-processor and post-processor files) can access and exchange global parameters values through a packet called '**global_info**'. If a set of parameters is needed to be passed to different files then these parameters are added to the **global_info** packet. Since **global_info** packet is visible in all the files, the values of the parameters in the packet can be read and even changed in different files.

4.3 Integrating with MATLAB environment

One of the most important reasons 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-4-2. For example, by combining GPenSIM with the Control systems toolbox, we can experiment hybrid discrete-continuous control applications, etc.

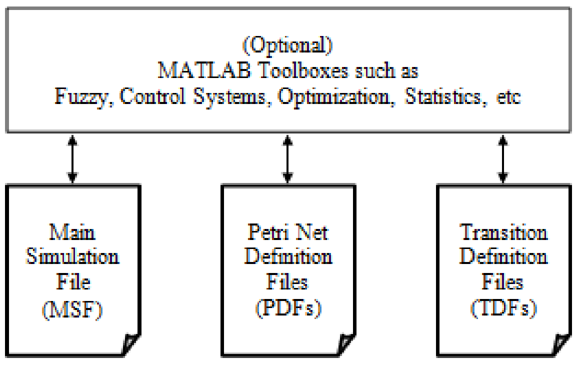

(Optional)
MATLAB Toolboxes such as
Fuzzy, Control Systems, Optimization, Statistics, etc
Main Simulation File (MSF)
Petri Net Definition Files (PDFs)
Transition Definition Files (TDFs)

Figure 4-2: Integrating with MATLAB environment

5. Creating a Classic (Untimed) Petri Net

The methodology for creating a Petri net model consists of three 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 passive elements of a Petri net graph: the places,
b. Identifying the active elements of a Petri net graph: the transitions, and
c. Connecting the these elements with arcs

Step-2. Coding the firing conditions in the relevant pre-processor files and post-firing works in the post-processor files

Step-3. Assigning the initial 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.

5.1 Example-05-01: A Simple Classic (untimed) Petri Net

As the first example, we will create an **untimed** (classic) Petri Net. The two steps are explained below, using the sample Petri net model shown in figure-5-1.

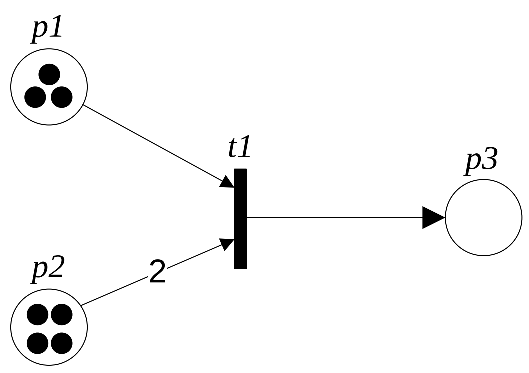


Figure-5-1: A simple Petri Net model

5.1.1 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-5-1 has three places, one transition, and three arcs. The PDF for the graph is given below:

```
% Example-05-01: A Simple Classic Petri Net

function [pns] = simple_pn_pdf()

pns.PN_name = 'Definition of a Simple Classic Petri Net ';
pns.set_of_Ps = {'p1', 'p2', 'p3'};
pns.set_of_Ts = {'t1'};
pns.set_of_As = {'p1', 't1', 1, 'p2', 't1', 2, 't1', 'p3', 1};
```

Explanation:

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

```
> PN_name = 'Definition of a Simple Classic Petri Net';
```

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

```
> set_of_Ps = {'p1', 'p2', 'p3'};
```

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

```
> set_of_Ts = {'t1'};
```

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 'p1' (source), to 't1' (destination) with a unit arc weight:

```
> set_of_As =  {'p1','t1',1, 'p2','t1',2, 't1','p3',1};
```

5.1.2 Step-2: Creating the pre-processor and post-processor files

This example is simple in the sense the transition will always fire if enabled. Thus, there are no additional firing conditions to be coded in the pre-processor file. In addition, there is no need for post-processor file either, as no post-firing work is given.

5.1.3 Step-3: The main simulation file: assigning the initial dynamics

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

```
> pns = pnstruct('simple_pn_pdf');
```

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

```
> dyn.m0 = {'p1',3, 'p2',4};
> pni = initialdynamics(pns, dyn);
```

5.1.4 The Simulations

Function gpensim will do the simulations if the Petri net marked graph (the static part and the initial dynamics) are passed to it:

```
> Sim_Results = gpensim(pni);
```

The output parameter of gpernsim (Sim_Results) is the simulation results.
Sim_Results is a structure for the simulation results. In order to comprehend the simulation results easily, the function '**prnss**' (meaning 'print statespace') could be used.

5.1.5 Viewing the simulation results with 'prnss'

```
> prnss(Sim_Results);
```

The output is given below:

```
======= State Diagram =======
**    Time: 0   **
State:0 (Initial State): 3p1 + 4p2
At start ....
At time: 0,  Enabled transitions are:    t1
At time: 0,  Firing transitions are:     t1

**    Time: 0   **
State: 1
Fired Transition: t1
Current State: 2p1 + 2p2 + p3
Virtual tokens: (no tokens)

Right after new state-1 ....
At time: 0,  Enabled transitions are:    t1
At time: 0,  Firing transitions are:     t1


**    Time: 0   **
State: 2
Fired Transition: t1
Current State: p1 + 2p3
Virtual tokens: (no tokens)

Right after new state-2 ....
At time: 0,  Enabled transitions are:
At time: 0,  Firing transitions are:
```

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

```
> plotp(Sim_Results, {'p1', 'p2', 'p3'});
```

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

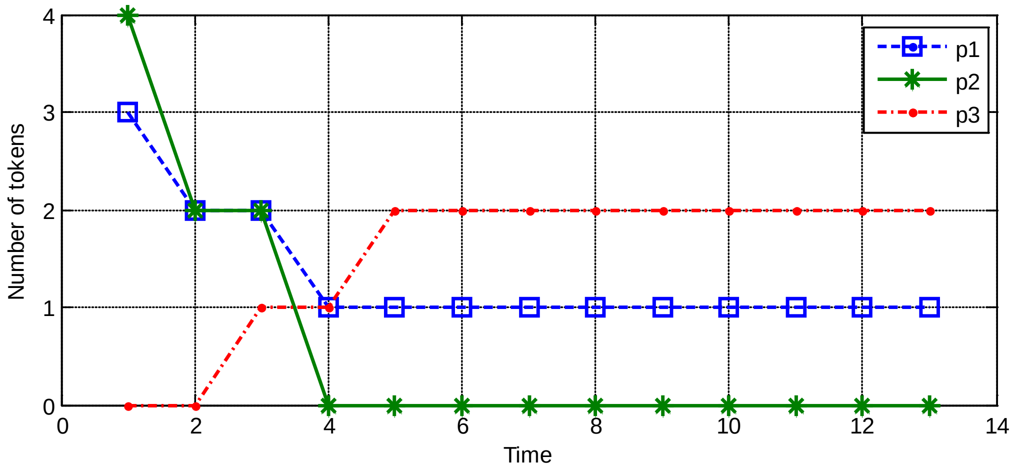


Figure 5-2: Simulation results of business logic computation

5.2 Summary

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

```
% Example-05-01: A Simple Classic Petri Net
% file: 'simple_pn_pdf.m'

function [pns] = simple_pn_pdf()

pns.PN_name = 'Definition of a Simple Classic Petri Net';
pns.set_of_Ps = {'p1', 'p2', 'p3'};
pns.set_of_Ts = {'t1'};
pns.set_of_As = {'p1','t1',1, 'p2','t1',2, 't1','p3',1};
```

Step-2: In this example, step-2 is missing as there is no need for pre-processor and post-processor files.

Step-3: Step-3 is for assigning the initial dynamics (initial markings) 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-5-1 is repeated below:

```
% Example-05-01: A Simple Classic Petri Net
% the main file to run simulation

pns = pnstruct('simple_pn_pdf'); % create petri net structure

dynamic_info.m0 = {'p1',3, 'p2',4};
pni = initialdynamics(pns, dynamic_info);

Sim_Results = gpensim(pn1); % perform simulation runs
prnss(Sim_Results); % print the simulation results
plotp(Sim_Results, {'p1','p2','p3'}); % plot the results
```

5.3 Static PN structure

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

```
      pns = pnstruct('simple_pn_pdf');
```

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

```
                  name: 'A Simple Petri Net'
         global_places: [1x3 struct]
          No_of_places: 3
    global_transitions: [1x1 struct]
     No_of_transitions: 1
        global_Vplaces: [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_Vplaces: the tokens that are residing inside firing transitions, 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.

5.4 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 in this case.

5.5 GPenSIM Reserved Words

CAUTION! There are a few reserved words in GPenSIM. Using these words as variable names should be avoided.

Reserved Words:

'PN' (meaning Petri Net)	PN represents the whole Petri net static model and is visible as a global variable in all GPenSIM system files.
'**global_info**'	global_info is also a global variable that carries user defined variables as well as global OPTIONS too all the systems files.

Avoid using these words for your variables: 1) PN, 2) global_info

6. Creating a Timed Petri Net

As the second example, we will create a simple **Timed** Petri Net; this example is almost the same as the previous example-05-01 (figure-5-1) except the fact that the transiton(s) are assigned firing time to make a Timed Petri Net.

6.1 Example-06-01: A Simple Timed Petri Net

6.1.1 PDF

The PDF will be the same as before as there is no change in the Petri net graph (figure-5-1):

```
% Example-06-01: A Simple Timed Petri Net

function [pns] = simple_pn_pdf()

pns.PN_name = 'Definition of a Simple Timed Petri Net ';
pns.set_of_Ps = {'p1', 'p2', 'p3'};
pns.set_of_Ts = {'t1'};
pns.set_of_As = {'p1', 't1', 1, 'p2', 't1', 2, 't1', 'p3', 1};
```

6.1.2 MSF

In the MSF, the only change from the previous example is the assignment of firing time to the only ttansition t1.

```
> dynamic_info.m0 = {'p1',3, 'p2',4};
> dynamic_info.ft = {'t1',10};    % firing time of 't1' is 10 TU
> pni = initialdynamics(pns, dyn);
```

6.1.3 The Simulations

Function ‘gpensim’ will do the simulations if the Petri net marked graph is passed to it and the results can be echoed (ascii output) on the screen by ‘prnss’; in addition to the ASCII output, we can also view the output graphically with ‘plotp’.

```
> Sim_Results = gpensim(pni);
> prnss(Sim_Results);
> plotp(Sim_Results, {'p1', 'p2', 'p3'});
```

The output is given below:

6.1.4 Simulation Results

Of course, ‘Transition-1’ takes 10 TU (e.g. milliseconds) to produce a token on ‘Place-3’, after removing 1 and 2 tokens from ‘Place-1’ and ‘Place-2’ respectively.

```
======= State Diagram =======
**    Time: 0    **
State:0 (Initial State): 3p1 + 4p2
At start ....
At time: 0,  Enabled transitions are:    t1
At time: 0,  Firing transitions are:     t1
```

```
**    Time: 10   **
State: 1
Fired Transition: t1
Current State: 2p1 + 2p2 + p3
Virtual tokens: (no tokens)

Right after new state-1 ....
At time: 10,  Enabled transitions are:    t1
At time: 10,  Firing transitions are:     t1


**    Time: 20   **
State: 2
Fired Transition: t1
Current State: p1 + 2p3
Virtual tokens: (no tokens)

Right after new state-2 ....
At time: 20,  Enabled transitions are:
At time: 20,  Firing transitions are:
```

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

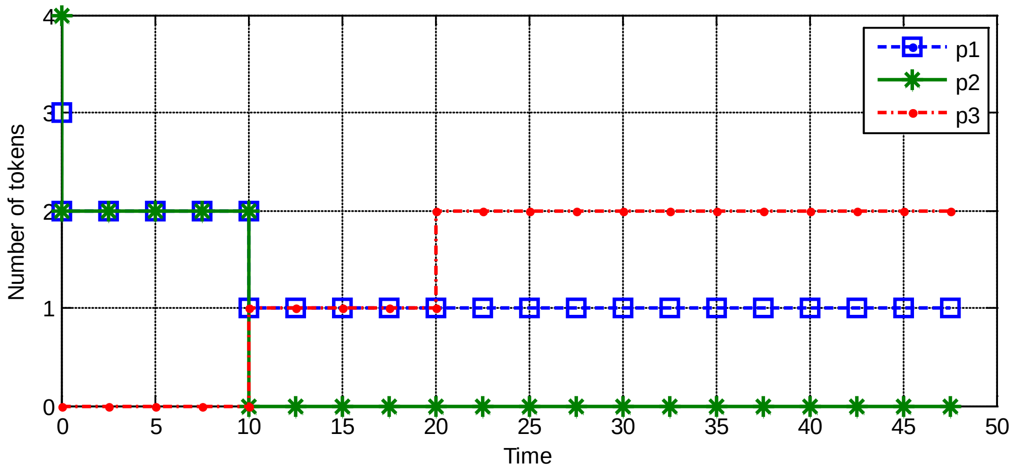


Figure 6-1: Simulation results of business logic computation

7. PRE-PROCESSORS AND POST-PROCESSORS

The previous section explained the methodology for modeling and simulation with GPenSIM consisting of three steps. However, in the previous section, the step-2 was omitted as there was no need for pre-processors and post-processors.

There are four types of processor (earlier known as 'TDF') files:

- **Specific pre-processor** file**:**
  This file is run **before** firing a specific transition. This file is coded with firing conditions for a **specific** transition; thus, when a transition is enabled, the corresponding pre-processor file will be checked, if it exist, to make sure all the firing conditions coded in this file is met; only if all the firing conditions coded in the pre-processor file is satisfied (this file returns '1') then the transition can fire.
  Let's say that if **t1** is enabled; if the file **'t1_pre'** exists then it will be run. Only if **'t1_pre'** returns logic 1 value, then the enabled **t1** is allowed fot fire immediately.

- **Specific post-processor** file**:**
  This file is coded with cleanup work (or accounting work) that may be necessary **after** firing of a specific transition.
  Let's say that **t1** fires; right after the firing, if the file **'t1_post'** exists then it will be also run.

- **COMMON_PRE** file:
  Just like the individual pre-processor files that are specific for individual transitions, COMMON_PRE file, if exists, will be run **before** firing of **every** transition. Thus, COMM_PRE is **common** (one and only one for all transitions), and contains firing conditions that for all the enabled transitions must satisfy in order to fire.
  Let's say that if **t1** is enabled; if the file **t1_pre** exists then it will be run. Also, if the file **COMMON_PRE** exists, it will also fun. Only if **both** **t1_pre** and COMMON_PRE return logic 1 value, then the enabled **t1** is allowed to fire.

- **COMMON_POST** file:
  This file is coded with cleanup work or accounting work needed to be done after firing of every transition. This means, there is only one COMMON_POST possible.
  Let's say that **t1** fires; right after the firing, if the file **t1_post** exists then it will be run.
  In addition, if the file **COMMON_POST** exists then it will be also run.

In section-7-2, we discuss about the pre-processor files by working through the example shown in figure-7-1.

7.1 Structure of a pre-processor file

Given below is a pre-processor file:

```
function [fire, transition] = tRobot_1_pre(transition)
% function [fire, transition] = tRobot_1_pre(transition)
…
…
fire = ;
```

The **input parameter is**:

1) **transition:** input parameter transition is a structure representing the enabled transition. This strucrure has many fields including "transition.name" which is the name of the enabled transition.

The **output parameters** are:

1) **fire: must be logic 'true' for firing**; if fire is equal to logical false, then the enabled transition can not fire – it is blocked.
2) **transition**: a structure representing the enabled transition that may or not fire. trans is a structure (packet) that will carry the following information back to the calling function:
    a. **transition.new_color**: (explained in the section on Coloring Tokens)
    b. **transition.override**: (explained in the section on Coloring Tokens)
    c. **transition.slected_tokens** (explained in the section on Coloring Tokens)

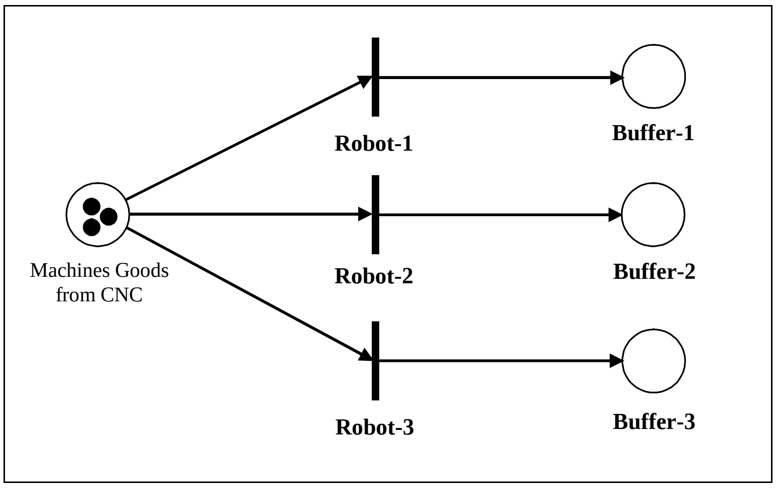


Figure 7-1: Petri net model of a production facility

7.2 Example-07-01: Pre-processor Example

Figure-7-1 shows a Petri net model of a production facility where three robots are involved in sorting products from an input buffer to output buffers. The three robots are represented by transitions and the buffers by places.

The firing conditions:

1. The output buffers have limited capacity: Buffer-1, buffer-2, and buffer-3 can accommodate a maximum of 2, 3, and 1, machined parts respectively.
2. 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 firing conditions stated above shall be coded in the pre-processor files.

7.2.1 Creating M-Files

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

- Step-1: Creating the **PDF** file
- Step-2: Creating the **pre-processors for the three transitions**. This is because, there are firing conditions attached to the transitions.
- Step-3: creating the **MSF**: assigning the initial dynamics (initial markings and firing times) and running the simulations.

7.3 The M-files

7.3.1 Step-1: PDF: Defining the Petri net graph

Let's call the PDF for the Petri net in figure-7-1 as 'pre_processor_pdf.m':

```
% Example-07-01: Pre-processor Example
% file: pre_processor_pdf.m: definition of petri net

function [pns] = pre_processor_pdf()

pns.PN_name = 'Pre-processor Example: PN for production facility';

pns.set_of_Ps = {'pFrom_CNC', 'pBuffer_1', 'pBuffer_2', 'pBuffer_3'};
pns.set_of_Ts = {'tRobot_1','tRobot_2','tRobot_3'};
pns.set_of_As = {'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};
```

7.3.2 Step-2: Pre-processor files

We will need 3 pre-processor files – one for each transition:

- Pre-processor **'tRobot_1_pre': tRobot_1** will fire only if the number of tokens (machined parts) already put in output **pBuffer_1** is less than 2. In addition, number of tokens in **pBuffer_1** should be less than that of **pBuffer_2**; coding these two firing conditions into the Pre-processor for **tRobot-1** is given below. As the name of the transition is '**tRobot_1**', this Pre-processor must be named '**tRobot_1_pre.m**'.

```
function [fire, transition] = tRobot_1_pre(transition)

b1 = get_place('pBuffer_1');
b2 = get_place('pBuffer_2');
fire = and((b1.tokens < b2.tokens), (b1.tokens < 2));
```

- Similarly, the Pre-processor files for **tRobot_2** and **tRobot_3** are created, satisfying the given firing conditions:

```
function [fire, transition] = tRobot_2_pre(transition)

b2 = get_place('pBuffer_2');
fire = lt(b2.tokens, 3);
```

```
function [fire, transition] = tRobot_3_pre(transition)

b1 = get_place('pBuffer_1');
b3 = get_place('pBuffer_3');
fire = and(gt(b1.tokens, b3.tokens), lt(b3.tokens, 1));
```

7.3.3 Step-3: MSF: Assigning the initial dynamics & running simulations

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

```
% Example-07-01: Pre-processor Example

global global_info;
global_info.STOP_AT = 60; % stop after 60 time units

pns = pnstruct('pre_processor_pdf');
dyn.m0 = {'pFrom_CNC',8}; % tokens initially
dyn.ft = {'tRobot_1',10, 'tRobot_2',5, 'tRobot_3',15};
pni = initialdynamics(pns, dyn);

sim = gpensim(pni);

prnss(sim);
plotp(sim, {'pBuffer_1', 'pBuffer_2', 'pBuffer_3'});
```

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

7.3.4 Outcome-1:

```
    Time: 0
State:0 (Initial State)
pBuffer_1 pBuffer_2 pBuffer_3 pFrom_CNC
 0         0         0         8
At time: 0
  Enabled transitions are:
 tRobot_1   tRobot_2   tRobot_3
At time: 0
  Firing transitions are:
 tRobot_2

    Time: 5
State: 1
Fired Transition: tRobot_2
Current State:
pBuffer_1 pBuffer_2 pBuffer_3 pFrom_CNC
 0         1         0         7
At time: 5
  Enabled transitions are:
 tRobot_1   tRobot_2   tRobot_3
At time: 5
  Firing transitions 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         5
At time: 10
  Enabled transitions are:
 tRobot_1   tRobot_2   tRobot_3
At time: 10
  Firing transitions 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         4
At time: 15
  Enabled transitions are:
 tRobot_1   tRobot_2   tRobot_3
At time: 15
  Firing transitions are:
 tRobot_1

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

    Time: 25
State: 5
Fired Transition: tRobot_1
Current State:
pBuffer_1 pBuffer_2 pBuffer_3 pFrom_CNC
 2         3         0         2
At time: 25
  Enabled transitions are:
 tRobot_1   tRobot_2   tRobot_3
At time: 25
  Firing transitions are:
 tRobot_3

    Time: 30
State: 6
Fired Transition: tRobot_3
Current State:
pBuffer_1 pBuffer_2 pBuffer_3 pFrom_CNC
 2         3         1         2
At time: 30
  Enabled transitions are:
 tRobot_1   tRobot_2   tRobot_3
At time: 31.25
  Enabled transitions are:
 tRobot_1   tRobot_2   tRobot_3
>>
```

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

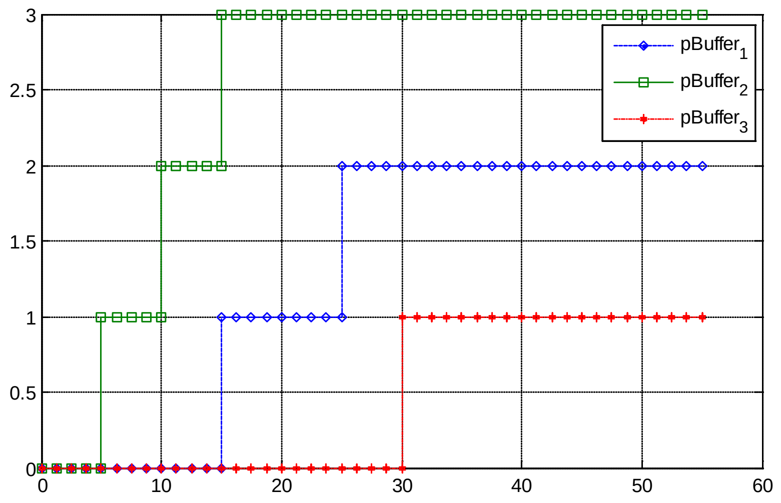


Figure 7-2: Simulation results

7.4 Example-07-02: COMMON_PRE Example

In the previous example, we used three pre-processor files (namely **tRobot_1_pre**, **tRobot_2_pre**, and **tRobot_3_pre**) to check whether the enabled transitions can fire. If we had ten robots then we have to create ten pre-processor files – this may become tiresome. Instead of separate pre-processor files we can create one and only one COMMON_PRE file. Section on "**COMMON PROCESSORS**" presents the details. But, just to taste the technique, COMMON_PRE file is given below that will **replace** the three individual pre-processor files.

```
function [fire, transition] = COMMON_PRE(transition)
% function [fire, trans] = COMMON_PRE(trans)

b1 = get_place('pBuffer_1');
b2 = get_place('pBuffer_2');
b3 = get_place('pBuffer_3');

if (strcmp(transition.name, 'tRobot_1')),
    fire = and(lt(b1.tokens, b2.tokens), lt(b1.tokens, 2);

elseif (strcmp(transition.name, 'tRobot_2')),
    fire = lt(b2.tokens, 3);

elseif (strcmp(transition.name, 'tRobot_3')),
    fire = and(gt(b1.tokens, b3.tokens), lt(b3.tokens, 1);

else
    error('transition name is neither of the three robots ...')
end;
```

8. Implementing Preference through Pre-processors

When more than one enabled transition compete for common resource or even common input tokens, some times it is better to have a preference so that the firing is deterministic rather than allowing an arbitrary transition to fire. In such a situation, pre-processor and COMMON_PRE can be used to block some transitions so that some others can be allowed to fire. A better way to do this is to assign priorities to the transitions so that GPenSIM automatically choose (no coding necessary) the transition with the highest priority when there is completion. However, in this section, we will see how we can achieve preference through pre-processor/COMMON_PRE.

8.1 Example-08-01: Implementing Preference through pre-processors

In this example (figure-8-1), transitions **t1** and **t2** both competes for tokens in **pS;** we prefer **t1** over **t2**. Preference to **t1** can be imposed by increasing **priority** of **t1** – this will be discussed later. In the example shown below, we show how pre-processor can be used to prefer **t1**, or rather prevent **t2** whenever **t1** is also enabled.

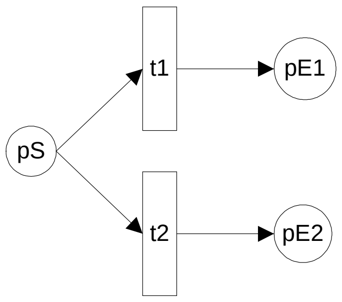


Figure 8-1: Petri Net model of a production facility

MSF:

```
% MSF: Example-08-01: prefer.m
global global_info;
global_info.STOP_AT = 60;   % Stop after 60 time units

% CASE = 1: t1 is preferred;
% CASE = 2: t2 is preferred
%   otherwise, no preference
global_info.CASE = 1;

pns = pnstruct('prefer_pdf');

dyn.ft = {'allothers',10};  % firing times for all the transition is 10 TU
dyn.m0 = {'pS',4};
pni = initialdynamics(pns, dyn);
sim_results = gpensim(pni);
plotp(sim_results, {'pE1', 'pE2'});
```

PDF:

```
% Example-08-01: prefer_pdf.m
function [pns] = prefer_pdf()
pns.PN_name = 'Preference example';
pns.set_of_Ps = {'pS', 'pE1', 'pE2'};
pns.set_of_Ts = {'t1','t2'};
pns.set_of_As = {'pS','t1',1, 't1','pE1',1, 'pS','t2',1, 't2','pE2',1};
```

Pre-processor t1_pre

```
function [fire, transition] = t1_pre (transition)

global global_info;
if eq(global_info.CASE, 2),
    % Case = 2: t2 should fire if enabled
    fire = not(is_enabled('t2'));
else
    fire = 1;
end;
```

Pre-processor: t2_pre

```
function [fire, transition] = t2_pre (transition)

global global_info;
if eq(global_info.CASE, 1),
    % Case = 1: t1 should fire if enabled
    fire = not(is_enabled('t1'));
else
    fire = 1;
end;
```

Simulation results:

CASE = 1

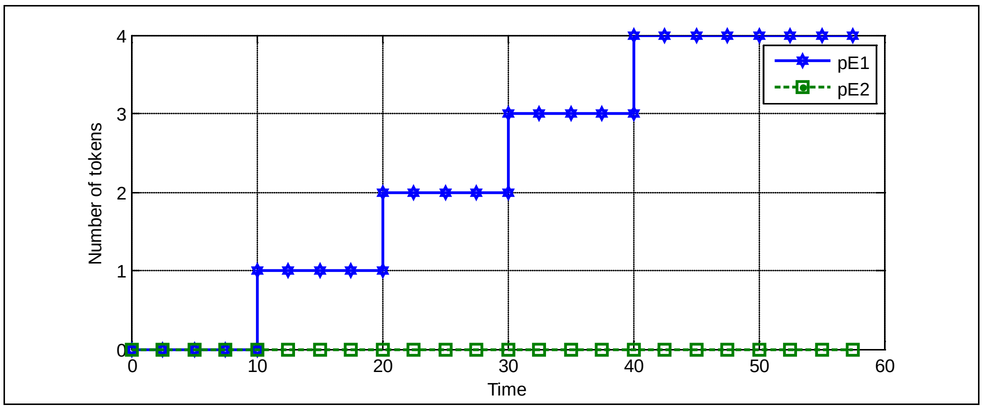


Figure 8-2: Simulation results

CASE = 2

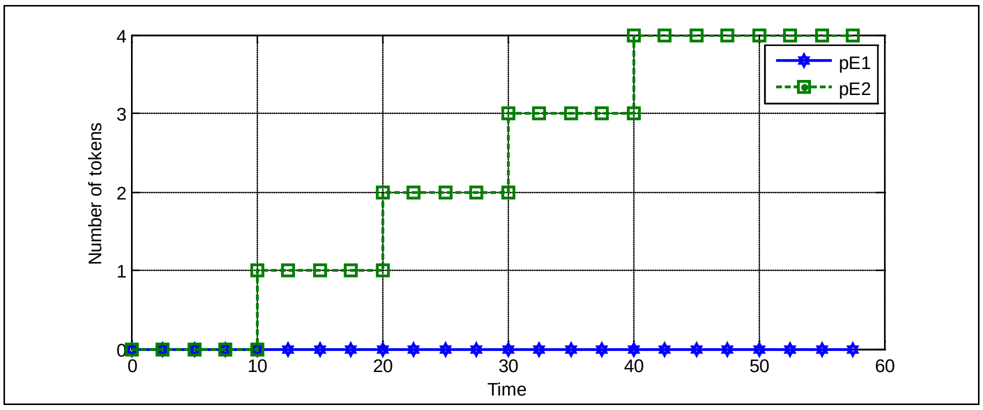


Figure 8-3: Simulation results

9. Post-processor

As stated in the earlier sections, there are two types of processors:

- **Pre-processor**, which are run ***before*** firing a transition, just to check whether an enabled transition can start firing, by checking all the firing conditions given in the _pre file.
- **Post-processor**, which are run ***after*** firing of a transition, in order to perform any post-firing activities.

Given below is a post-processor file for the transition '***tX1***':

```
function [] = tX1_post(transition)
% function tX1_post
...
```

The **input parameter** is: **transition: this is a structure containing <u>name</u> of the <u>fired</u> transition**
The **output parameters** are: **<u>NONE</u>**

<u>Note</u>: If we are going to use any variables in **global_info** packet, then we must declare global_info as a global variable. We can also access the run-time PN structure by declaring PN as a global variable too.

9.1 Example-09-01: Binary Semaphore

Figure-9-1 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.

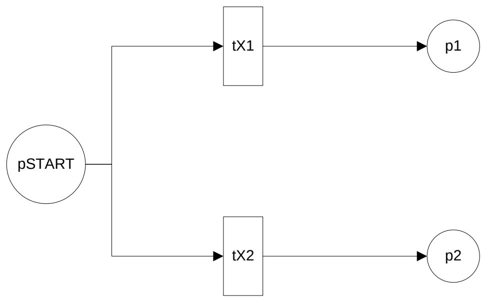


Figure 9-1: Load balancing by alternative firing

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

9.1.1 PDF: ('loadbalance_pdf.m'):

```
% Example-09-01: Load balance with Binary semaphore
function [pns] = loadbalance_pdf()
pns.PN_name = 'Load Balancer with binary semaphore';
pns.set_of_Ps = {'pSTART', 'p1', 'p2'};
pns.set_of_Ts = {'tX1','tX2'};
pns.set_of_As = {'pSTART','tX1',1, 'tX1','p1',1,...
                 'pSTART','tX2',1, 'tX2','p2',1};
```

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

```
% Example-09-01: Load balancing with Binary semaphore

global global_info;
global_info.semafor = 1;    % GLOBAL DATA: binary semafor
global_info.STOP_AT = 100;  % stop after 150 TU

pns = pnstruct('loadbalance_pdf');
dynamicpart.m0 = {'pSTART', 10};
dynamicpart.ft = {'tX1', 10, 'tX2', 20};
pni = initialdynamics(pns, dynamicpart);
sim = gpensim(pns);
plotp(sim, {'p1', 'p2'});
```

Note: the parameter **semafor** with an initial value of 1 is added to the global info packet. The initial value of ‘1’ for semafor means the transition **tX1** is to fire first.

Pre-processor for tX1 (‘tX1_pre.m’):

```
function [fire, transition] = tX1_pre(transition)

global global_info;
fire =  eq(global_info.semafor,1); % fire only if value of semaphore = 1
```

Post-processor for tX1 (‘tX1_post.m’):

```
function [] = tX1_post(transition)

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

Pre-processor for tX2 (‘tX2_pre.m’):

```
function [fire, transition] = tX2_pre(transition)

global global_info;
fire = eq(global_info.semafor,2); % fire only if value of semaphore = 2
```

Post-processor for tX2 (‘tX2_post.m’):

```
function [] = tX2_post(transition)

global global_info;
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 firing alternatively.

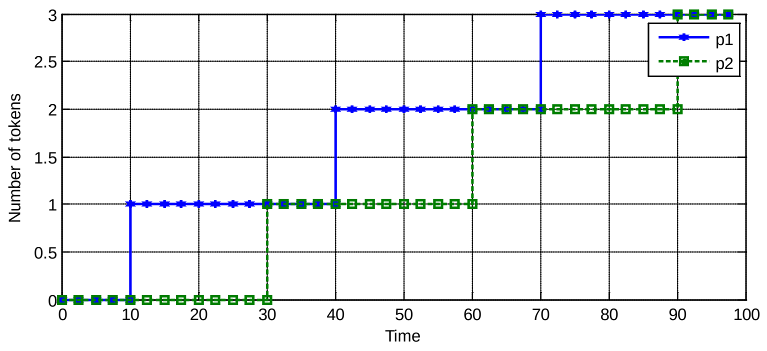


Figure 9-2: Binary semaphore in action

10. COMMON PROCESSORS

In the examples shown up to now, the processors are specific for individual transitions. This means, if there are n transitions, we may have to code a maximum of n pre-processor files and n post- processor files, which may make up to many processor files. In most cases, these files are almost equal, except for the name of the firing transitions. GPenSIM allows common pre-processor and post- processor files for all the transitions. In this way, if we code a flexible COMMON_PRE and COMMON_POST files, we need just 4 files (MSF, PDF, COMMON_PRE, and COMMON_POST) for simulation of a system.

Note:
Naming of the two files, must be all capitalized: 'COMMON_PRE' and 'COMMON_POST'

10.1 Structure of COMMON_PRE and COMMON_POST files

Given below is a COMMON_PRE file:

```
function [fire, transition] = COMMON_PRE(transition)
...
...
fire =
```

The **input parameters** are:

1) **transition: a structure that posses the <u>name</u> of the <u>enabled</u> transition (transition.name)**

The **output parameters** are:

1) **fire: must be logical true for firing**; if fire is equal to logical false, then the enabled transition can not fire – it is blocked.
2) **transition**: a structure to which we can add values like **new_color**, **override**, and **slected_tokens**

Given below is a **COMMON_POST** file:

```
function [] = COMMON_POST(transition)
...
...
```

The **input parameters** are:

1) **transition: a structure that posses <u>name</u> of the <u>fired</u> transition**

The **output parameters** are: **<u>NONE</u>**

10.2 SUMMARY: COMMON Processors versus Specific Processors

In a system with *n* transitions:

- *On one extreme*, we can code the system with a maximum of *n* specific pre-processor files and *n* specific post-processor files. In addition, we can have COMMON_PRE and

COMMON_POST files as well; this makes (2*n*+2) processor files, in addition to MSF and PDF.
- *On the other extreme*, we can code the system with just 2 processor files: COMMON_PRE and COMMON_POST files.

So, which is better? Perhaps, a combination of these two should be used; common actions can be put naturally in the common files, and actions that are specific and heavy (in terms of coding) can be put in the specific files. **In real-time environments, it is not a good idea to overload the common files; in real-time environment, a light common files supplemented by specific processor files is ideal**. In Part-III "Applications" of this manul, we will work thorugh som applications where a mixture of common and specific files is used.

	COMMON PROCESSOR files	**Specific processor files**
Advantage	**Fewer files (just 2 files: COMMON_PRE and COMMON_POST)**	**Easy to understand as it uses specific names and variables**
Disadvantage	**Hard to understand; very generic and can become large**	**Many files (2*n*)**

10.3 Mixing COMMON and Specific processors:

Let us imagine a Petri Net that consists of three transitions, **tA**, **tB**, and **tC**.

11. Inhibitor Arcs

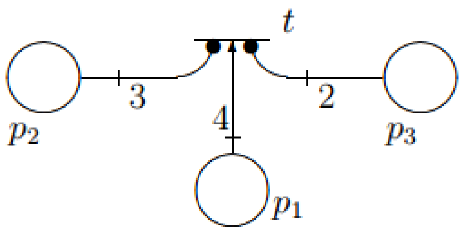
Petri Nets with inhibitor arcs are more powerful than the “basic” P/T Petri Net. Petri Nets with inhibitor arcs have the same expressive power as Turing Machines; Turing Machines have the ability of modeling any discrete event systems. Thus, Petri Nets with inhibitor arcs can model and simulate the functioning of any discrete event systems.
In the Petri Net models, inhibiting arc are drawn with a circle at the end of the arc, instead of pointing arrow.

11.1 Firing rule inhibitor arcs are used:

A transition is enabled if:

a) As with Ordinary P/T Petri nets: the number of tokens in each input place is at least equal to the weight of the input arc from that place.
b) Special for inhibitor arcs: the number of tokens in each input place with an inhibitor arc is less than the weight of the input inhibitor arc from that place.

Example: In the following Petri Net (taken from Ciardo, 1987), the transition t is enabled if and only if the input place p1 has greater than or equal to 4 tokens (usual rule) and the inhibiting input places p2 and p3 has less than 3 and 2 tokens, respectively.



t enabled if: $|p_1| \geq 4$ and $|p_2| < 3$ and $|p_3| < 2$

t disabled if: $|p_1| < 4$ or $|p_2| \geq 3$ or $|p_3| \geq 2$

Figure 11-1: Transition t with two inhibiting arcs p2 and p3 (source: Ciardo, 1987)

In the implementation of this Petri Net, the only change will be in the Petri net Definition File (PDF), where there will be now two types of arcs: the (normal) input arcs, and the inhibitor arcs:

```
function [pns] = simple_inhib_pdf()
pns.PN_name = 'PDF for: Simple Inhibitor arc Example';
pns.set_of_Ps = {'p1', 'p2', 'p3'};
pns.set_of_Ts = {'t'};
pns.set_of_As = {'p1','t',4};                  % normal arc
pns.set_of_Is = {'p2','t',3, 'p3','t',2}; % Inhibitor arcs
```

11.2 Example-11-01: Batch processing

The following model shows a simple mechanism for batch processing.

- Producer ***tS*** produces one item at a time and deposits the products into the buffer ***pS***.
- Buffer ***pS*** has a capacity for holding a maximum of 10 products only; hence, producer ***tS*** stops when the number of products in the buffer ***pS*** becomes 10.
- ***tE*** is the packing machine that picks up 5 products at a time from the buffer ***pS***, and pack them as one packet and places it to the output buffer ***pE.***

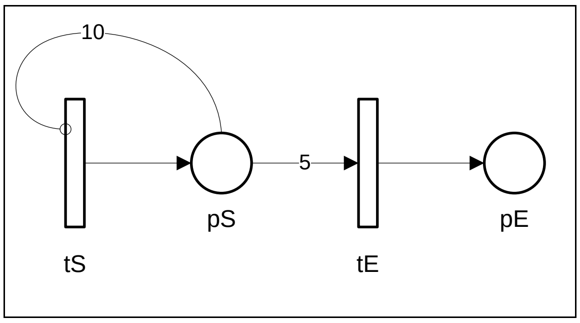


Figure 11-2: Petri Net model of batch processing

PDF:

```
% Example-11-01: A simple Inhibitor Arc example
function [pns] = simple_in_pdf()
pns.PN_name = 'PDF for: Simple Inhibitor arc Example';
pns.set_of_Ps = {'pS', 'pE'};
pns.set_of_Ts = {'tS', 'tE'};
pns.set_of_As = {'tS','pS',1, 'pS','tE',5, 'tE','pE',1}; % input arcs
pns.set_of_Is = {'pS','tS',10};  % inhibitor arc
```

MSF:

```
% Example-11-01: A simple Inhibitor Arc example
clear all; clc;
global global_info;
global_info.STOP_AT = 13;

pns = pnstruct('simple_in_pdf');
dyn.ft = {'tS',0.2, 'tE',5};
pni = initialdynamics(pns, dyn);
sim = gpensim(pni);
plotp(sim, {'pS', 'pE'}, 0, 2);
```

Results:

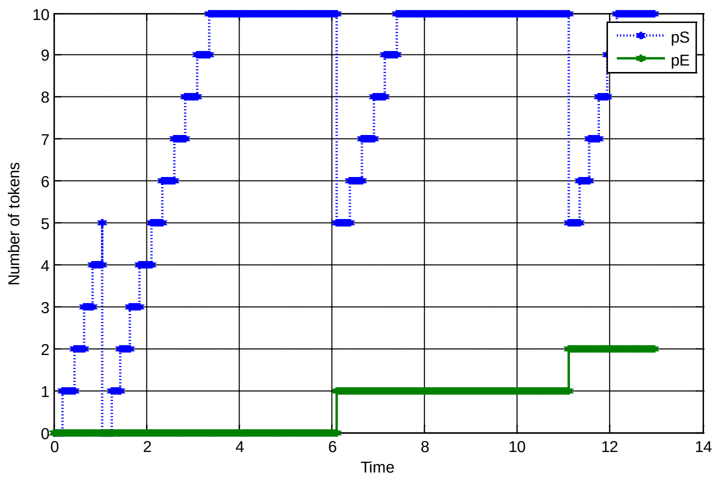


Figure 11-3: Simulation of batch processing

12. Global Timer

During simulation, GPenSIM uses an internal clock known as the global timer; the global timer is discrete and is incremented by a default value that is ¼ of the shortest firing time of any transition. If this default time increment is not satisfactory, then the sampling frequency can be changed (increased) by assigning another value to the timer increment value.

12.1 Example-12-01: DELTA_TIME demo

In the figure shown below, let **p1** has 3 initial tokens. Also let firing time of **t1** is 7 seconds. Though **t1** can fire 3 times successively, we want it to fire only at the start of every 30 seconds.

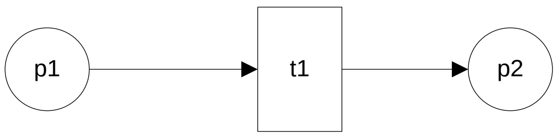


Figure 12-1: Delay example

The sampling rate (timer increment value) is 7/4 = 1.75 seconds, unless this value is overridden by global option **DELTA_TIME** (see the section on global option). See the gpensim system file '*timed_pensim.m*' for implementation details.

MSF:

```
% Example-12-01: Timer increment example

global global_info;
global_info.STOP_AT = 80;

pns = pnstruct('timer_inc_pdf');

dyn.m0 = {'p1',3};
dyn.ft = {'t1',7};
pni = initialdynamics(pns, dyn);
sim = gpensim(pni);
plotp(sim, {'p1','p2'}, 0, 2);
```

PDF:

```
% Example-12-01: Timer increment example
function [pns] = timer_inc_pdf()
pns.PN_name = 'Timer increment example';
pns.set_of_Ps = {'p1', 'p2'};
pns.set_of_Ts = {'t1'};
pns.set_of_As = {'p1','t1',1, 't1','p2',1};
```

Pre-processor t1_pre:

```
function [transition] =  t1_pre(transition)
rest = mod(current_time(), 30);
fire =  (rest < 5);   % any number less than 7 would do
```

Simulation results:

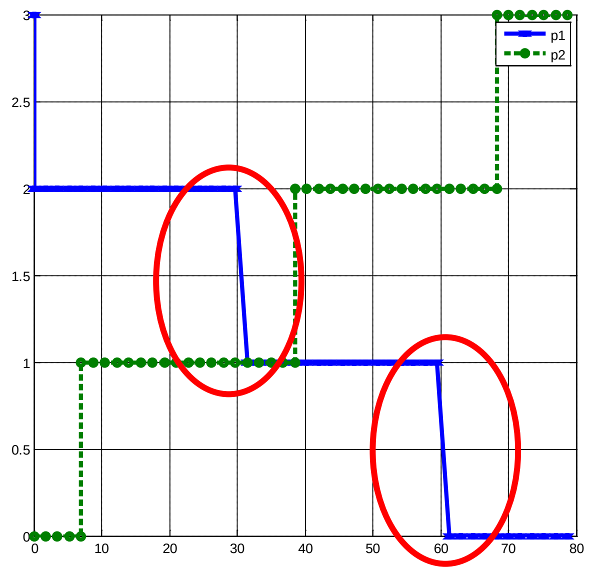


Figure 12-2: Simualtion results

If we closely look into the plot for **p1** (figure-12-2)**,** when it changes from 2 tokens to 1 token and also from 1 token to 1 token, the plot looks like a ramp function (slanted) than a pulse. However, this timer increment can be fine tuned by overriding the default sampling rate; see also the section on "**OPTIONS**". Just for now, include the following instruction at the top of MSF: **`global_info.DELTA_TIME = 0.01;`**

MSF:

```
% Example-12-01: delay example
% file: delay_demo.m:
global global_info;
global_info.DELTA_TIME = 0.01; % timer increment value is 0.01 sec
global_info.STOP_AT = 80; % stop sim after 80 seconds

pns = pnstruct('delay_demo_pdf');

dyn.m0 = {'p1',3};
dyn.ft = {'t1',7};
pni = initialdynamics(pns, dyn);

sim = gpensim(pni);
plotp(sim, {'p1','p2'}, 0, 2);
```

This means, we have reduced the time increment value from 7/4 seconds (1.75 seconds) to 0.01 seconds, by 175 times; in other words, the sampling frequency is increased 175 times. The simulation result given below in figure-12-3 proves that the sampling is done at a very high rate as the plot for **p1** now looks like a pulse than a ramp.

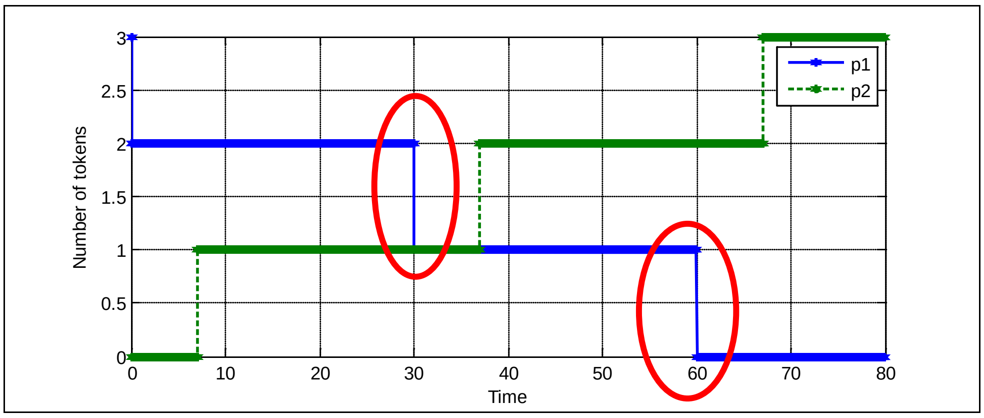


Figure 12-3: Crisp plots with overridden sampling rate

Finally, we will decrease the sampling rate and see what happens; lets slow-down sampling by assigning a larger value to the timer increment value, say 5 seconds (again, default increment value is firing time of t1 divided by 4 = 7/4 = 1.75):

MSF:

```
% Example-12-01: timer increment example
global global_info;
global_info.DELTA_TIME = 5; %time advancement is 5 sec (low samplings rate)
global_info.STOP_AT = 80;   % stop sim after 80 seconds

pns = pnstruct('timer_inc_pdf');
…
```

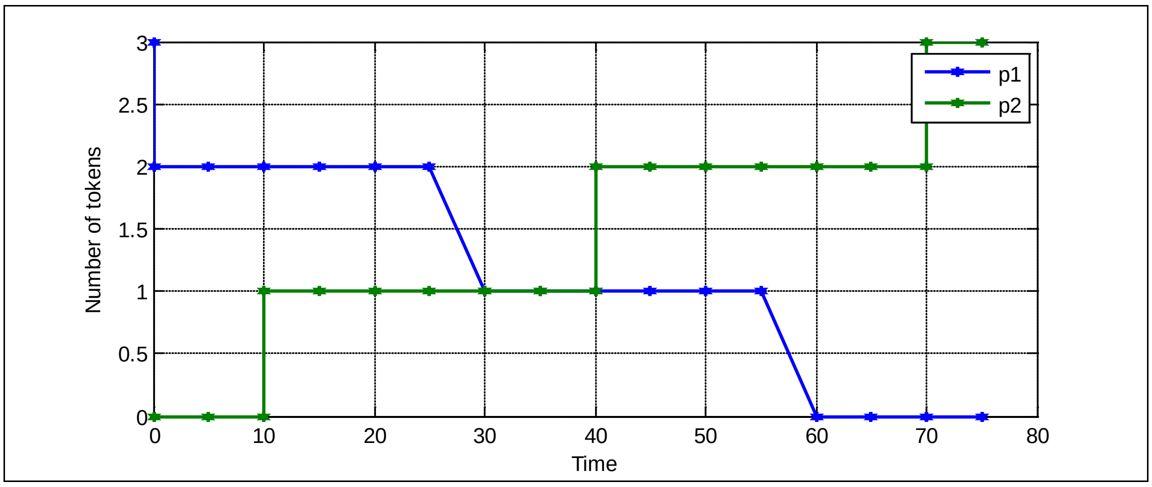


Figure 12-4: No longer impulse falls but ramps.

13. OPTIONS

'Global info' packet helps passing variables and parameters between different files (e.g. MSF, PDFs, and processors). 'Global info' packet also serves another important purpose: setting OPTIONS for simulations. As its name depicts, OPTIONS are not part of the Petri net model; OPTIONS helps change default simulation settings. OPTIONS are always stated in capital letters (e.g. 'MAX_LOOP') and added to the 'global info' packet in the main simulation file. Given below is a list of OPTIONS and their usage.

NOTE: If you use OPTIONS, remember to declare "global_info" as a global variable in the main simulation file and also in the other files where you use them.
E.g.:

```
global  global_info; % declare in the MSF
```

13.1 'STOP_AT'

We have already used this OPTION, just to stoop simulation after some time units. This option will not work for classic Petri Nets as time is undefined in them.

13.2 'MAX_LOOP'

By default, simulations are run **200** cycles or **loops**. Sometimes, simulation of a model for 200 loops seems unnecessarily lengthy; in this case 'MAX_LOOP' can be used to trim down the simulation loops to a much lower value (e.g. global_info.MAX_LOOP = 10). On the other hand, default 200 loops may seems too little, as the simulation end prematurely; in this case, we may again use 'MAX_LOOP' to assign a large value to number of the simulation loops to be run (e.g. global_info.MAX_LOOP = 20000).

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

13.2.1 Example-13-01: MAX_LOOP

This is the same as the example-06-01. This time, we will experiment with 'MAX_LOOP' OPTION.

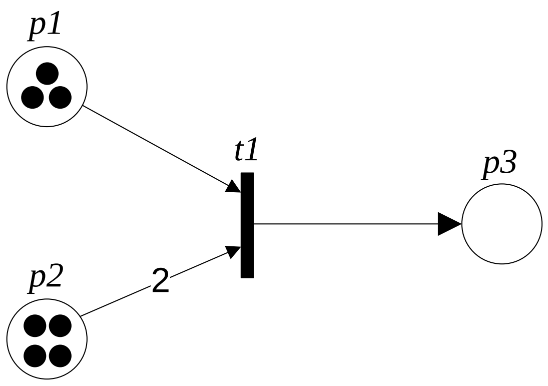


Figure 13-1: Transitions in sequence

The Petri net shown in figure-13-1, with the value of 10 TU as the firing time for t1, will only run for a short time. Thus, unless specified in the MSF, default maximum loops of 200 (by default, MAX_LOOP=200) will be run making the plot file (see figure-13-2) less detailed.

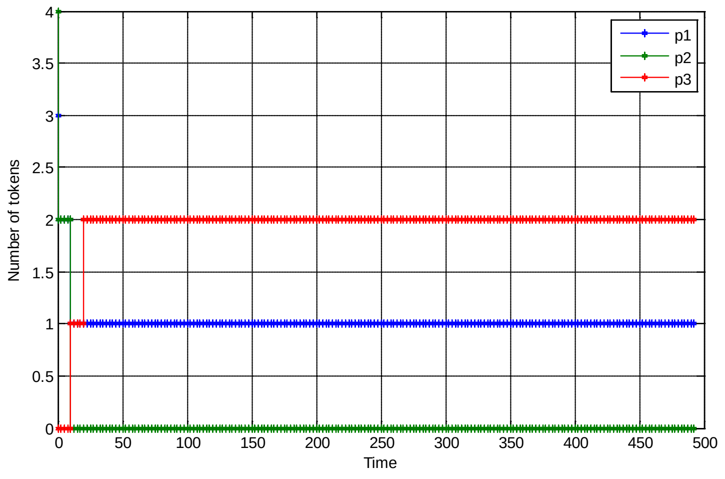


Figure 13-2: Plot with default MAX_LOOP (=200)

If we stop the simulations after a couple of simulation cycles, then we may have a more detailed plot. The statement given below limits the simulation cycles to 11, by assigning the value 11 to 'MAX_LOOP':

```
> global_info.MAX_LOOP = 11; % OPTION to limit simulation cycles to 11
```

Simulation results: When MAX_LOOP=11, a meaningful plot results.

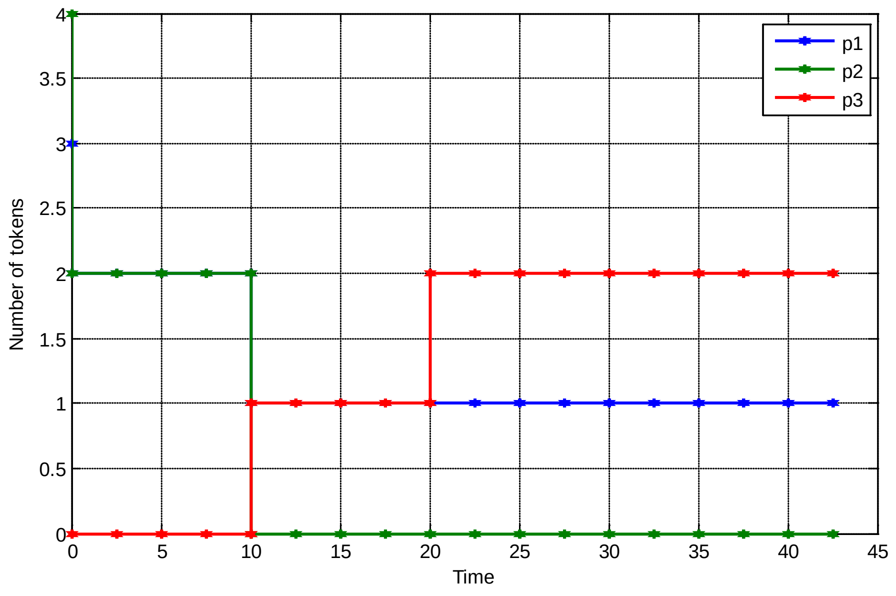


Figure 13-3: MAX_LOOP is set to 11

13.3 What are loops?

(See the section on "Design of GPenSIM" for more details)

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 processors, by calling **pn.current_time**, where **pn** is the run-time Petri net structure. Function ***current_time()***

also returns the current time of the global timer. The global timer is incremented by a discrete value after every cycle, which we call loop.

2) *Event Scheduler:* Event scheduler is the main activity in a **loop**, 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 *firing_queue* of firing transitions (implemented in file **start_firing.m**).
    b. Second: checking for completion of firing of the firing transitions in the *firing_queue*. When a firing transition is complete, it will be removed from the *firing_queue* (implemented in file **complete_firing.m**)

   In GPenSIM, file **timed_pensim.m** implements event scheduler.
3) *Firing_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 have taken place so far.
NOTE: Section on "Design of GPenSIM" gives more details.

13.4 'PRINT_LOOP_NUMBER'

When we simulate large Petri net models, during the simulations we will notice that the **MATLAB hangs**, without giving us any sign of life. It will be better, if we can see some outputs during simulations so that we'll be assured that the simulations are going on and that the system is not dead ('hanging'). By setting the **PRINT_LOOP_NUMBER=1** OPTION, we can see the loop numbers when the simulation goes on.

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

13.5 'DELTA_TIME'

The previous section on "Global Timer" describes with an example (example-12-01) about the timer increment value: after every simulation loop, the global timer is advanced by a timer increment 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 the OPTION "**DELTA_TIME**".

13.6 'STARTING_AT'

(Note: For more details, see the section on "Using Hourly Clock")
So far, we have treated clock as a unit less timer; it will always start at time=0 during simulation start, and will increase afterwards. However, there are situation where:

- We need not start at time zero. We may start at time= 600, as things only happens from that time.
- 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. In addition, for business modeling, the clock should perhaps start at 09:00 AM, rather than at '0'.

OPTION '**STARTING_AT**' is to set the start of the clock to a specific time.
E.g.

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

The time for starting could be given as a three column vector [H M S] or as a single number; if it is given as a single number, then the global timer will keep on running in terms of TUs (e.g. in seconds). Only if the starting time is given a vector of three elements ([Hour Min Sec]), then the global timer will become an hourly clock.

See the following section for more details on hourly clock.

13.7 'STOP_SIMULATION'

'**STOP_SIMULATION**' is a special kind of OPTION, in fact the only OPTION that is manipulated in the processor files (pre and post), and not in the MSF; the other OPTIONS are usually set in the MSF, at the top of the file. 'STOP_SIMULATION' is to force the simulations to stop when some specific conditions are met.

13.7.1 Example-13-02: STOP_SIMULATION Demo

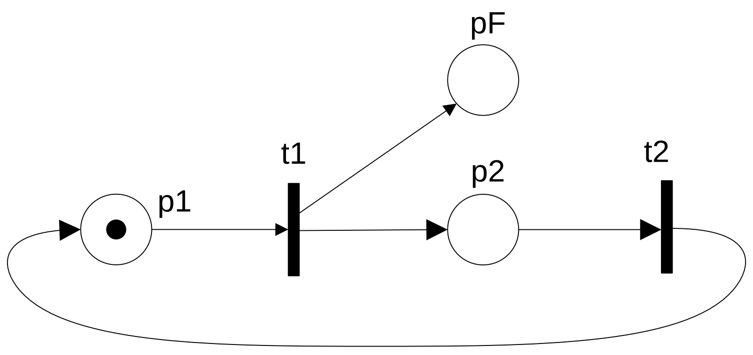


Figure 13-4: Petri Net that will run forever

Figure-13-4 shows a Petri Net that will run forever. However, we will try to stop the simulation run with 'STOP_SIMULATION' OPTION; as we now know that we could also stop the simulation by limiting either MAX_LOOP (usually for untimed systems) to a lower value or assigning STOP_AT (for timed systems only) with a specific time to stop.

Let's say that we want to stop the simulation after a specific number of **t1** firings: the simulations must be stopped after three (3) **t1** firings. This can be done simply by checking the **t1** firing count in any pre-processor (either t1_post or t2_post is more suitable, as they are run after t1 firing) and **force the simulation** to stop if the count condition is met. There are two ways to do this:

- Either, setting the **STOP_SIMULATION** option in the **t2_pre** after 3 **t1** firing
- Or, setting the **STOP_SIMULATION** option in the **t1_post** after 3 **t1** firing.

1) Using the pre-processor t2_pre

```
function [fire, transition] = t2_pre(transition)
global global_info;

t1 = get_trans('t1');
if ge(t1.times_fired, 3),
      % stop simulation, after 3 t1-firing
      global_info.STOP_SIMULATION = 1;
end;
fire = 1;
```

Or, 2) using the post-processor t1_post

```
function [] = t1_post(transition)

global global_info;

t1 = get_trans('t1');
if ge(t1.times_fired, 3),
    global_info.STOP_SIMULATION = 1;
end;
```

Results: Simulation results show that the system stops after 3 **t1**-firings.

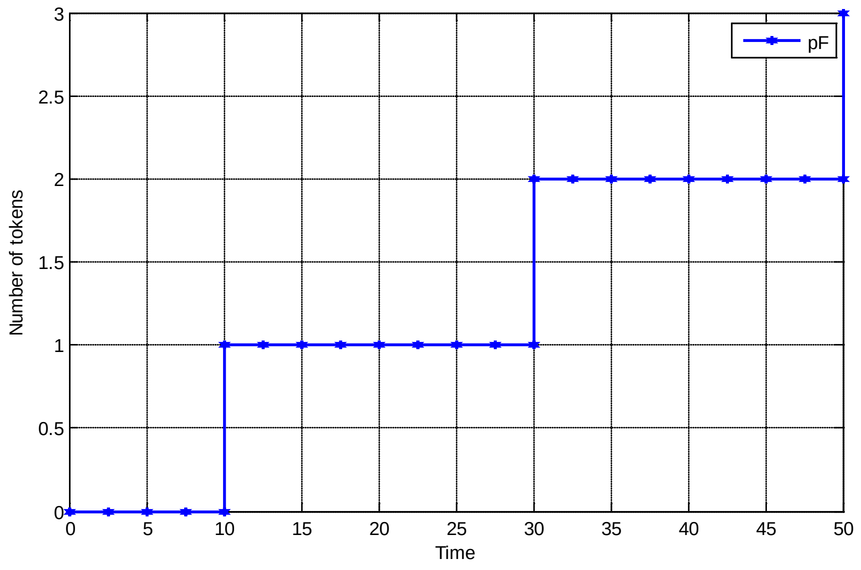


Figure 13-5: Abrupt stop of simulations

14. Using Hourly Clock

So far, we have treated clock as a unit less 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 elements (e.g. 1:00 PM as [13, 0, 0]); we can mix time in 3 column hourly format with single number; however, the single number will be taken as second.

E.g.:

[0 40 0]	**is equivalent to 40 minutes (or 2400 seconds)**
'5 + 1*rand(1)'	**is equivalent to 5 seconds on average**
180	**is equivalent to 180 seconds**
[9 0 0]	**this is 09 AM**
34200	**this is also 09 AM (9x60x60 = 32400) ONLY if it is used to add with Another vector of three element; e.g. 34200 + [0 0 0]**

14.1 Example-14-01: Hourly Clock

An office opens at 12:00 AM on every business day. Customers arrive at every 15 minutes. There are two clerks who will interact with the customers. The clerks take 45 minutes to service a customer. The office closes at 01:30 PM, and no customer will be allowed into the office. However, those customer(s) already inside the office will be serviced.

14.1.1 Functions for hourly clock

First of all, we want to start the simulation at 12:00 AM. This can be fed into the model through the global_info packet.

```
global_info.STARTING_AT = [12 0 0]; % start 12:00:00 HH:MM:SS
```

In MSF, to assign firing times to tGEN customer arrival (every 15 minutes), and to clerk-A and clerk-B (45 minutes each), we may either use the hourly clock format or times in seconds:

```
dyn.ft = {'tGEN',15*60, 'tA',45*60,'tB', [0 45 0]};
```

Note: Because of the use of hourly clock formats, the functions **prnss** and **plotp** display time information in hourly formats. The Petri net model for the system is shown in figure-14-1.

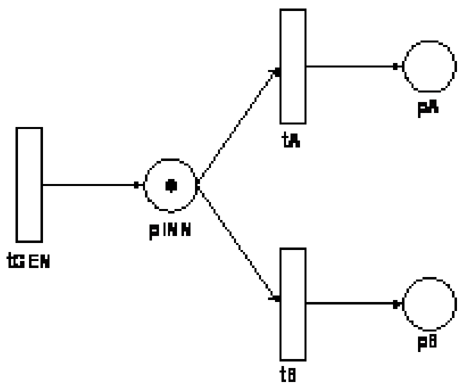


Figure 14-1: A Simple Bank

MSF:
Note: Print functions are not used in MSF, as the _pre and _post files are coded with printing statements.

```
% Example-14-01: Hourly Clock
global global_info;

global_info.START_AT = [12 0 0]; % OPTION: start simulations at 10 AM
global_info.STOP_AT  = [15 15 0]; % OPTION: stop  simulations at 15 AM
global_info.DELTA_TIME = 60;  % delta_T is 1 minutes
global_info.BANK_CLOSED = false; % initially, bank is just opened

pns = pnstruct('hourly_clock_pdf');
dyn.m0 = {'pINN',1};
dyn.ft = {'tGEN',[0 15 0], 'tA', 45*60, 'tB', [0 45 0]};
dyn.ip = {'tA',1}; % let clerk-A take the first customer

pni = initialdynamics(pns, dyn);
results = gpensim(pni);
figure(1), plotp(results, {'pINN'}, 0, 2);
figure(2), plotp(results, {'pA', 'pB'}, 0, 2);
```

PDF:

```
% Example-14-01: Hourly clock
function [pns] = oh_pdf()
pns.PN_name = 'Office Hours';
pns.set_of_Ps = {'pINN', 'pA', 'pB'};
pns.set_of_Ts = {'tGEN', 'tA', 'tB'};
pns.set_of_As = {'tGEN','pINN',1, 'pINN','tA',1, 'pINN','tB',1, ...
                 'tA','pA',1, 'tB','pB',1};
```

Pre-processor files:
There is no need for specific pre-processor files for tA and tB.
However, we need a specific Pre-processor for tGEN as it should stop generating customers after hours 13:30.

tGEN_pre: tGEN will not fire after 13:30.

```
function [fire, transition] = tGEN_pre(transition)

global global_info;

if global_info.BANK_CLOSED,
    fire = 0; return;
end;
```

```
time_stamp = num2str(string_HH_MM_SS(current_time()));
time_13_30_in_seconds = 13.5*60*60;  % Hour 13:30 in seconds

if lt(current_time(), time_13_30_in_seconds),
    disp([time_stamp, ':  ', transition.name, ' Making a customer ']);
    fire = 1;
else
    disp([time_stamp, ':  ',transition.name,' ** BANK IS CLOSED NOW. **']);
    global_info.BANK_CLOSED = true;
    fire = 0;
end;
```

We could have a COMMON pre and post files, commonly for tA and tB, just to print statements whenever the clerks take receive customers and complete business with them.

COMMON_PRE:

```
function [fire, transition] = COMMON_PRE(transition)

time_stamp = num2str(string_HH_MM_SS(current_time()));
if ismember(transition.name, {'tA', 'tB'}),
    disp([time_stamp, ':  ', transition.name, ' taking a customer ']);
end;
fire = 1;
```

COMMON_POST:

```
function [] = COMMON_POST(transition)

time_stamp = num2str(string_HH_MM_SS(current_time()));
if ismember(transition.name, {'tA', 'tB'}),
    disp([time_stamp, ':  ', transition.name, ' finished a customer']);
end;
```

Simulation results show that the last customer leaves at 15:00.

```
12:00:00:  tA taking a customer
12:00:00:  tGEN Making a customer
12:15:00:  tB taking a customer
12:15:00:  tGEN Making a customer
12:30:00:  tGEN Making a customer
12:45:00:  tA finished a customer
12:45:00:  tA taking a customer
12:45:00:  tGEN Making a customer
13:00:00:  tB finished a customer
13:00:00:  tB taking a customer
13:00:00:  tGEN Making a customer
13:15:00:  tGEN Making a customer
13:30:00:  tA finished a customer
13:30:00:  tA taking a customer
13:30:00:  tGEN ** BANK IS CLOSED NOW. **
13:45:00:  tB finished a customer
13:45:00:  tB taking a customer
14:15:00:  tA finished a customer
14:15:00:  tA taking a customer
14:30:00:  tB finished a customer
15:00:00:  tA finished a customer
```

15. 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, assuming that all the transitions always fire if enabled.

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 the same as for 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)
Step-2. NO need for pre-processor files, as we assume that transitions always fire, if enabled
Step-3. Creating main simulation file (MSF) with the the initial dynamics (only the initial markings; firing times are not relevant); running the MSF using the function ‘**cotree**’ instead of ‘gpensim’.

Function ‘**cotree**’ takes three input arguments (parameters):

1. Input parameter ***pni*** is the marked Petri Net (with initial markings)
2. Input parameter ***plot_cotree*** indicates whether we want to see the graphical plot of the coverability tree
3. Input parameter ***print_cotree*** indicates whether we want to see ASCII print of the coverability tree

E.g.:

```
cotree(pni, 1, 1); % both plot & ASCII print wanted
```

The above statement will both plot coverability tree graphicaly as well as print ASCII information on the screen.

15.1 Example-15-01: Cotree with finite states

This simple example deals with the Petri net shown in figure -15-1. The co-tree of this Petri net is shown in figure-15-2. Let us find the co-tree using GPenSIM:

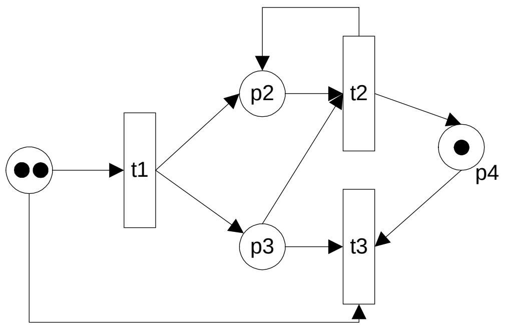


Figure 15-1: The Petri net for coverability analysis

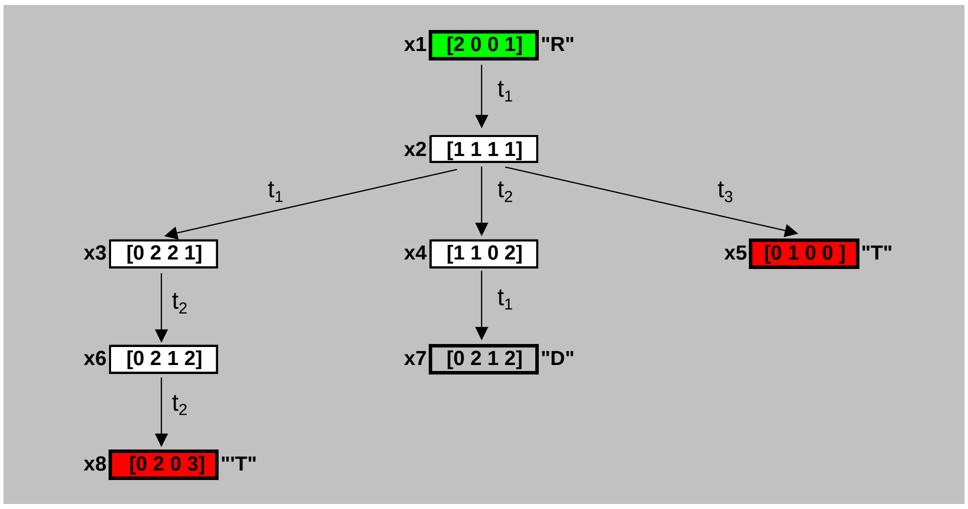


Figure 15-2: The reachable states of the Petri net shown in figure-15-1

15.1.1 Petri net definition file

The Petri net definition file is given below:

```
% PDF for Example-15-01: Cotree example-1
function [pns] = cotree_example_pdf()

pns.PN_name = 'COTREE Example: Petri Net in Figure 15-1';
pns.set_of_Ps = {'p1', 'p2', 'p3', 'p4'};
pns.set_of_Ts = {'t1', 't2', 't3'};
pns.set_of_As = {'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};
```

15.1.2 The main file

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

```
% Example-15-01: Cotree example-1
pns = pnstruct('cotree_example_pdf');
dyn.m0 = {'p1', 2, 'p4', 1}; % tokens initially
pni = initialdynamics(pns, dyn);
cotree(pni, 1, 1);
```

The function **cotree** will print the following on the screen, which is equivalent to the graphical co-tree shown in figure-15-2; **cotree** also prints the **boundedness** (maximum tokens possible in any place) and the liveness (whether the tree has any terminal nodes) of the Petri net.

The screen output given below is equivalent to the graphical plot shown in figure-15-2.

```
 ======= Coverability Tree =======
State no.: 1  ROOT node
2p1 + p4

State no.: 2    Firing event: t1
State: p1 + p2 + p3 + p4
Node type: ' '   Parent state: 1

State no.: 3    Firing event: t1
State: 2p2 + 2p3 + p4
Node type: ' '   Parent state: 2
```

```
State no.: 4    Firing event: t2
State: p1 + p2 + 2p4
Node type: ' '   Parent state: 2

State no.: 5    Firing event: t3
State: p2
Node type: 'T'   Parent state: 2

State no.: 6    Firing event: t2
State: 2p2 + p3 + 2p4
Node type: ' '   Parent state: 3

State no.: 7    Firing event: t1
State: 2p2 + p3 + 2p4
Node type: 'D'   Parent state: 4

State no.: 8    Firing event: t2
State: 2p2 + 3p4
Node type: 'T'   Parent state: 6

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

Liveness:
Terminal States: [5  8]
```

15.2 Example-15-02: Cotree with infinite states

In this example, we will use GPenSIM to generate coverability tree of the Petri net shown in figure-15-3. The co-tree of this Petri net is shown in figure-15-3. Let us find the co-tree using GPenSIM:

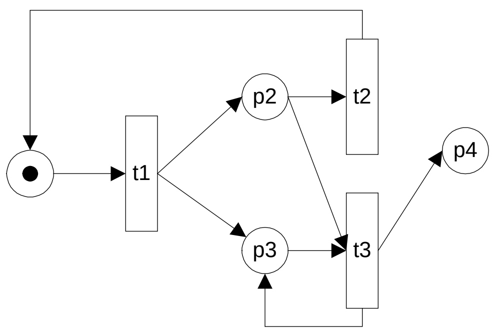


Figure 15-3: Petri net for Cotree example

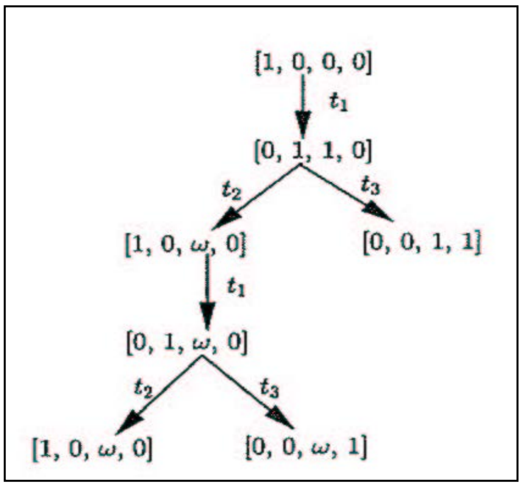


Figure 15-4: Coverability Tree

PDF:

```
% PDF Example-15-02: Cotree example-2
function [pns] = cotree_15_02_pdf()
pns.PN_name = 'Petri net: Cassandras & Lafortune, p.253, fig 4.10';
pns.set_of_Ps = {'p1', 'p2', 'p3', 'p4'};
pns.set_of_Ts = {'t1','t2', 't3'};
pns.set_of_As =  {'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};
```

MSF:

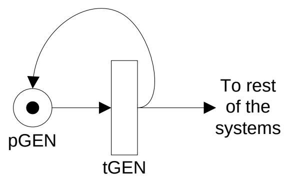
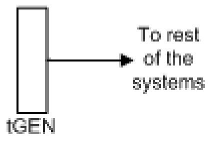
```
% Example-15-02: Cotree example-2
Clear all; clc;
pns = pnstruct('cotree_15_02_pdf');
dyn.m0 = {'p1',1};
pni = initialdynamics(pns, dyn);
cotree(pns, 1, 1);
```

The print system will print the following on the screen, which is equivalent to the graphical co-tree shown in figure-15-4.

```
 ======= Coverability Tree =======
State no.: 1  ROOT node
p1

State no.: 2    Firing event: t1
State: p2 + p3
Node type: ' '   Parent state: 1

State no.: 3    Firing event: t2
State: p1 + Infp3
Node type: ' '   Parent state: 2

State no.: 4    Firing event: t3
State: p3 + p4
Node type: 'T'   Parent state: 2

State no.: 5    Firing event: t1
State: p2 + Infp3
Node type: ' '   Parent state: 3

State no.: 6    Firing event: t2
State: p1 + Infp3
Node type: 'D'   Parent state: 5

State no.: 7    Firing event: t3
State: Infp3 + p4
Node type: 'T'   Parent state: 5

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

Liveness:
Terminal States: [4  7]
```

16. Generators

It is usual to present a generator as shown in figure-16-1 below. However, the model shown in figure-16-2 is technically equivalent to figure-16-1, and is perfectly capable of continuously generating tokens and supplying the tokens to the rest of the system. This is because the transition **tGEN** in figure-16-2 has no input places, thus does not need any input tokens to be enabled; in figure-16-2, **tGEN** is always enabled.

 **Figure 16-1: Generator with initial enabling token**	 **Figure 16-2: Generator without initial enabling token**

Token generation need not be either deterministic (firing time of **tGEN** is fixed) or stochastic (firing time of **tGEN** is stochastic, defined as a random function); stochastic firing timing is discussed in secition-19 "Stochastic Firing Times". Token generation can be pre-defined, at the times pre-defined by the user. The following example shows how this can be achieved.

16.1 Example-16-01: Generator for token generation

In this example, tokens are generated at specific times. Though these times are read from a global variable, they can be also read from a structure variable or from a file.

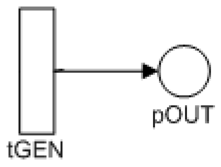


Figure 16-3: Generator (of tokens)

Figure above shows the simple Petri net. Times for firing are fed into the pre-processor as a global variable.

PDF:

```
% Example-16-01: Generator example % file: generator_pdf.m: PDF
function [pns] = generator_pdf()

pns.PN_name='Generator Example';
pns.set_of_Ps = {'pOUT'};
pns.set_of_Ts = {'tGEN'};
pns.set_of_As = {'tGEN','pOUT',1};
```

MSF: In the MSF, the times for generating tokens are stored in the global variable "`global_info.TOKEN_FIRING_TIME`". This variable will be utilized in the pre-processor.

```
% Example-16-01: GENERATOR example
global global_info;
```

```
global_info.STOP_AT = 170;
global_info.TOKEN_FIRING_TIME = [0 20 60 90 100 150];
global_info.DELTA_TIME = 1;

pns = pnstruct('generator_pdf');
% No initial tokens
% No firing times
pni = initialdynamics(pns);

sim = gpensim(pni);
prnss(sim);
plotp(sim, {'pOUT'});
```

Pre-processor:

```
function [fire, transition] = tGEN_pre(transition)

global global_info;

% if the variable "TOKEN_FIRING_TIME" is empty, then all
% the firings are done; no more firing is possible
if isempty(global_info.TOKEN_FIRING_TIME),
    fire = 0; return;
end;

time_to_generate_token = global_info.TOKEN_FIRING_TIME(1);
ctime = current_time();

% if it is time to fire, then remoev the time from variable and fire
if ge(ctime, time_to_generate_token),
   if ge(length(global_info.TOKEN_FIRING_TIME),2),
       global_info.TOKEN_FIRING_TIME = ...
             global_info.TOKEN_FIRING_TIME(2:end);
   else
       global_info.TOKEN_FIRING_TIME = [];
   end;
   fire = 1;
else  % it is not time to fire
   fire = 0;
end;
```

The pre-processor checks the times given in the global variables against the current time and fires if they are equal. After firing, the time is removed from the variable.

Simulation Results:

Plot below shows that firings were done at seconds 0, 20, 60, 90, 100, and at 150.

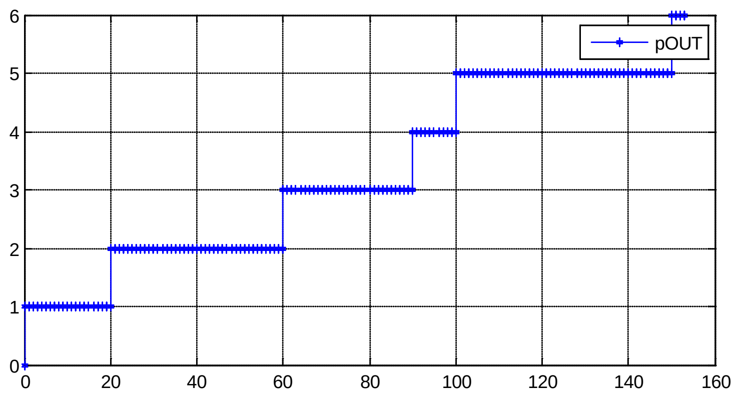


Figure 16-4: Token generation on predefined time

17. Prioritizing Transitions

In discrete systems, we need to increase or decrease priority of an event or events, in order to prioritize some event(s) and sometimes also 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, if needed; initial declaration can only be done in the main simulation file.
2) *Increase* priority of a transition (function 'priorinc')
3) *Decrease* priority of a transition (function 'priordec')
4) *Get* priority of a transition (function 'get_prior')
5) *Assign* priority to a transition (function 'priorset')
6) *Compare* priority of two transitions (function 'priorcomp')

Priority is assigned to transitons only, and can be manipulated in processors. Proirity in an integer value (positive and negative) and higher the priority value more prioritized the transition become. If not given an initial priority value, a transiton possess a default '0' value.

17.1 Priorities of transitions

Assignement of initial priorities can be done in the main simulation file; initial priorities can be coded as a part of the initial dynamics:

```
dyn.m0 = {'pS', 1}; % initial tokens
dyn.ip = {'t1',3, 't3',5}; % initial priority
pni = initialdynamics(pns, dyn);
sim = gpensim(pni);
```

In the above line, we are simply saying that t3 has top priority initially (5), followed by t1 (3); all other transitions not mention in the declaration will be assigned with initial priority 0. 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 '**priorinc**', which will increase the value just by 1.

```
priorinc('t1'); % priority of ’t1’ is now 4
```

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

```
priordec('t3'); % priority of ’t3’ is now 4
```

We can assign any priority to a transition using the function '**priorset**':

```
priorset('t1', 10); % priority of 't1' is now 10
```

We can also get current priority of a transition using the function '**get_priority**':

```
prio_val = get_priority('t1'); % a value of 10 will be returned
```

Finally, we can also compare priority of two transitions using the function '**priorcomp**':

```
HEL = priorcomp('t3', 't1'); % priority of 't1' 't3'
```

If **t3** has higher priority than **t1**, then the returned value will be 1; if both have equal priority then a value of zero will be returned; otherwise, -1 will be returned.

17.2 Example-17-01: Alternating firing with priority

Transitions t1, t2, and t3, should fire alternatively (figure-17-01).

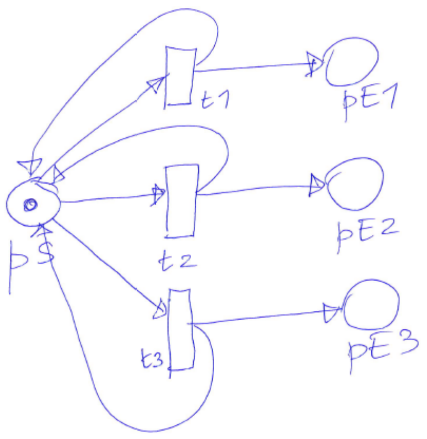


Figure 17-1: Alternating firing of t1, t2, and t3

Let's assume that **t2** is to be fired first, then **t3** then **t1**, and so on.

MSF:

```
% Example-17-01: Alternating firing using priority
global global_info;
global_info.STOP_AT = 30;

pns = pnstruct('prio_pdf');

dyn.m0 = {'pS', 1}; % initial tokens
dyn.ft = {'t1',1, 't2',1, 't3',1};
dyn.ip = {'t2', 1}; % t2 has higher priority than other two
pni = initialdynamics(pns, dyn);
sim = gpensim(pni);
plotp(sim, {'pE1', 'pE2', 'pE3'});
```

PDF:

```
% Example-17-01: Alternating firing using priority
function [pns] = prio_pdf()

pns.PN_name = 'Priority Example: Petri Net for production facility';
pns.set_of_Ps = {'pS', 'pE1', 'pE2', 'pE3'};
pns.set_of_Ts = {'t1','t2','t3'};
pns.set_of_As = {'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};
```

Post-processor for t1 ('t1_post.m'):

After firing of **t1**, **t2** should be fired. This can be assured by increasing the priority of **t2** above the priority of **t1:**

```
function [] = t1_post(transition)
```

```
% after firing t1, t2 should be fired;
% thus, increase priority of t2 above t1
pvalue = get_priority(transition.name); % here: transition.name = 't1'
priorset('t2', pvalue+1);
```

Post-processor for t2 (‘t2_post.m’):
After firing of **t2**, **t3** should be fired. This can be assured by increasing the priority of **t3** above the priority of **t2:**

```
function [] = t2_post(transition)

% after firing t2, t3 should be fired;
% thus, increase priority of t3 above t2
pvalue = get_priority(transition.name); % here: transition.name = 't2'
priorset('t3', pvalue+1);
```

Post-processor for t3 (‘t3_post.m’):
After firing of **t3**, **t1** should be fired. This can be assured by increasing the priority of **t1** above the priority of **t3:**

```
function [] = t3_post(transition)

% after firing t3, t1 should be fired;
% thus, increase priority of t1 above t3
pvalue = get_priority(transition.name); % here: transition.name = 't3'
priorset('t1', pvalue+1);
```

Simulation Results:
The results show that the three transitions fire alternatively **t1** -> **t2** -> **t3** -> **t1** …

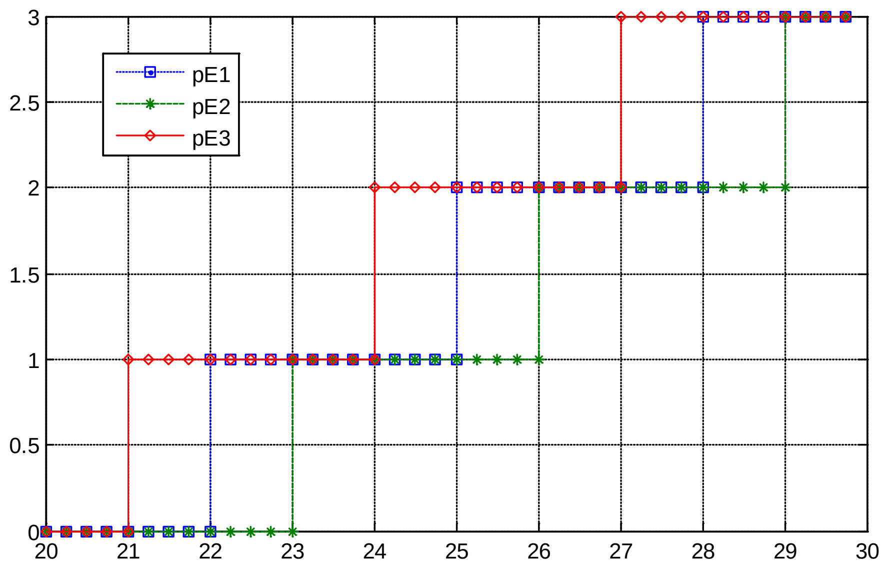


Figure 17-2: Alternating firing of t1, t2, and t3

17.3 Example-17-02: Priority Decrement Example

This example is the same as the previous example (Example-17-01) shown in figure-17-01. However, this time, we will use priority decrement rather than increment.

MSF:

```
% Example-17-02: Alternating firing using priority decrement
global global_info;
global_info.STOP_AT = 30;

pns = pnstruct('prio2_pdf');

dyn.m0 = {'pS', 1}; % initial tokens
dyn.ft = {'t1',1, 't2',1, 't3',1};
dyn.initial_priority = {'t2',2, 't3',1}; % initial priorities
sim = gpensim(pns, dyn);
plotp(sim, {'pE1', 'pE2', 'pE3'});
```

PDF: same as in the previous example:

post-processor: t1_post

```
function [] = t1_post(transition)

% after firing t1, decrease priority of t1 below the other two
pvalue2 = get_priority('t2');
pvalue3 = get_priority('t3');

pvalue = min(pvalue2, pvalue3) - 1;
priorset(transition.name, pvalue); % here: transition.name = 't1'
```

post-processor: t2_post

```
function [] = t2_post(transition)

% after firing t2, decrease priority of t2 below the other two
pvalue3 = get_priority('t3');
pvalue1 = get_priority('t1');

pvalue = min(pvalue3, pvalue1) - 1;
priorset(transition.name, pvalue); % here: transition.name = 't2'
```

post-processor: t3_post

```
function [] = t3_post(transition)

% after firing t3, decrease priority of t3 below the other two
pvalue1 = get_priority('t1');
pvalue2 = get_priority('t2');

pvalue = min(pvalue1, pvalue2) - 1;
priorset(transition.name, pvalue); % here: transition.name = 't1'
```

Simulation Results: Again, same as in the previous example!!!

18. 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. 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; thus, the three columns of the duration matrix are:

1) Column-1: The firing transition (index of the 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:

1) The first row presents total activation times (in ***TUs***) of each transition.
2) The second row presents activation in ***percentage*** of the total time.

The function occupancy also prints the activation times and percentages on screen.

18.1 Example-18-01: Measuring Activation Time

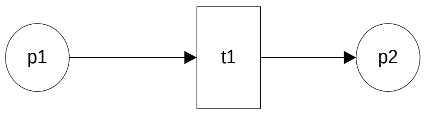


Figure 18-1: Measuring Activation Time

This example is the same as example-12-01, where the only transition of the system **t1** is forced to wait before each firing. This time, we will compute the idle time of the transition (and the activation time of the transition) with the help of the functions **extractt** and **occupancy**.

The only change (from example-12-01) this time in the MSF is that the addition of the last few lines:

MSF:

```
% Example-18-01: Delay Example for measuring activation time
global global_info;
global_info.STOP_AT = 70;
global_info.DELTA_TIME = 1;

pns = pnstruct('delay_demo_pdf');

dyn.m0 = {'p1',3};
dyn.ft = {'t1',7};
pni = initialdynamics(pns, dyn);
sim = gpensim(pni);

duration_matrix  = extractt(sim, {'t1'});
disp('** Duration_matrix: **');
disp(duration_matrix);

occupancy_matrix = occupancy(sim, {'t1'});
disp('## Occupancy Matrix: ##');
disp(occupancy_matrix);
```

Simulation results:

The duration matrix computed form the simulation result shows that the transition **t1** 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. However, the screen dump below indicates that the activation of t1 is 30.4% as the finishing time for simulation is 69 (=STOP_AT – 1).

```
** Duration Matrix: **
     1     0     7
     1    30    37
     1    60    67

Simulation Completion Time: 69
occupancy t1        :
  total time: 21
  Percentage time: 30.4348%


## Occupancy Matrix: ##
   21.0000
   30.4348

>>
```

18.2 Example-18-02: Measuring Activation time

This is another example for measuring activation time. Figure-18-2 below shows a simple system where two transitions fire sequentially, one after the other.

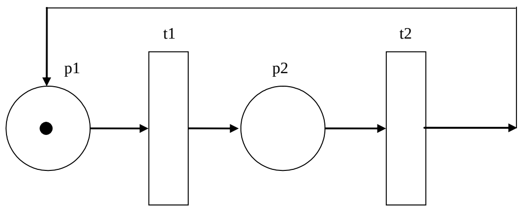


Figure 18-2: Transitions firing sequentially

The code below is for the main simulation file:

```
% Example-18-02: Measuring Activation Time
global global_info;
global_info.STOP_AT = 507; % stop afer 5 t1/t2 firings

pns = pnstruct('measure_timing_pdf');
dynamicpart.m0 = {'p1', 1};
dynamicpart.ft = {'t1', 1, 't2', 100};
sim = gpensim(pns, dynamicpart);

duartion_martix = extractt(sim, {'t1', 't2'});
disp('Duartion Martix: ');
disp(duartion_martix);
```

```
occupancy_martix = occupancy(sim, {'t1', 't2'});
disp('Occupancy Martix: ');
disp(occupancy_martix);
```

Simulation results:
Simulation result shows that transition **t1** fired during the time intervals 0-1, 101-102, 202-203, …, etc., and that transition **t2** fired during the time intervals 1-101, 102- 202, 203-303, …, etc. When the simulation was complete at time 506, t1 has fired 6 times for a total times of 6 time units, and t2 has fired 5 times a total time of 500 time units.

```
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
```

19. Stochastic Firing Times

So far, the ***firing times*** of 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.

With MATLAB's Advanced Statistical Toolbox, we can use many probability distribution functions. If our PC is not installed with Advanced Statistical Toolbox, then we have to limit stochastic firing time to basic "rand" function.

19.1 Example-19-01: Stochastic firing times with Advanced Stat. Toolbox

Again if the PC is installed with MATLAB's Advanced Statistical Toolbox, we can use any of the probability distribution functions for stochastic firing times. The following are the most used:

We refer to the CNC production system presented in example-05-01; 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-19-10: Example with stochastic timing
% the main simulation file
pns = pnstruct('stoch_example_pdf');
dynamics.m0 = {'pFrom_CNC', 20}; % initial tokens

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

pni = initialdynamics(pns, dynamics);
Results = gpensim(pni);
prnss(Results);
plotp(Results, {'pFrom_CNC', 'pBuffer_1', 'pBuffer_2', 'pBuffer_3'});
```

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

19.2 Example-19-02: Stochastic firing times WO Advanced Stat. Toolbox

If MATLAB's Advanced Statistical Toolbox is not installed on our PC, then we have to use a basic "rand" function: E.g.:

1) **Robot-1** takes random time normally distributed with mean 10 and standard deviation 2 milliseconds. ('`10 + 2*randn(1)`')
2) **Robot-2** takes random time normally distributed with mean 5 and standard deviation 2 milliseconds. ('`5 + 2*randn(1)`')
3) **Robot-3** takes random time normally distributed with mean 15 and standard deviation 2 milliseconds. ('`15 + 2*randn(1)`')

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

```
% Example-08-01: Example with stochastic timing
% the main simulation file
pns = pnstruct('stoch_example_pdf');
dynamics.m0 = {'pFrom_CNC', 8}; % initial tokens

% here comes the STOCHASTIC TIMING
dynamics.ft = {'tRobot_1', '10 + 2*randn(1)',...
    'tRobot_2', '5 + 2*randn(1)', 'tRobot_3', '15 + 2*randn(1)'};

pni = initialdynamics(pns, dynamics);
Results = gpensim(pni);
prnss(Results);
plotp(Results, {'pBuffer_1', 'pBuffer_2', 'pBuffer_3'});
```

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

NOTE: Only the standard (Basic) MATLAB is required to run GPenSIM; GPenSIM does not require MATLAB's Advanced Statistical Toolbox or any other toolboxes.

20. Modular Model Building

Figure 12 shows architecture of an adaptive supply chain based on service component architecture; see Davidrajuh (2007) for details. Figure-20-1 shows the equivalent Petri net model.

20.1 Example-20: Modular Model for Adaptive Supply Chain

The Petri net model shown in figure-20-1 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 ('monolithic model'). Instead, we can divide the Petri net graph into modules as shown in figure-20-1, and then create individual PDFs for each of the module; finally, all the PDFs are combined to form the complete model.

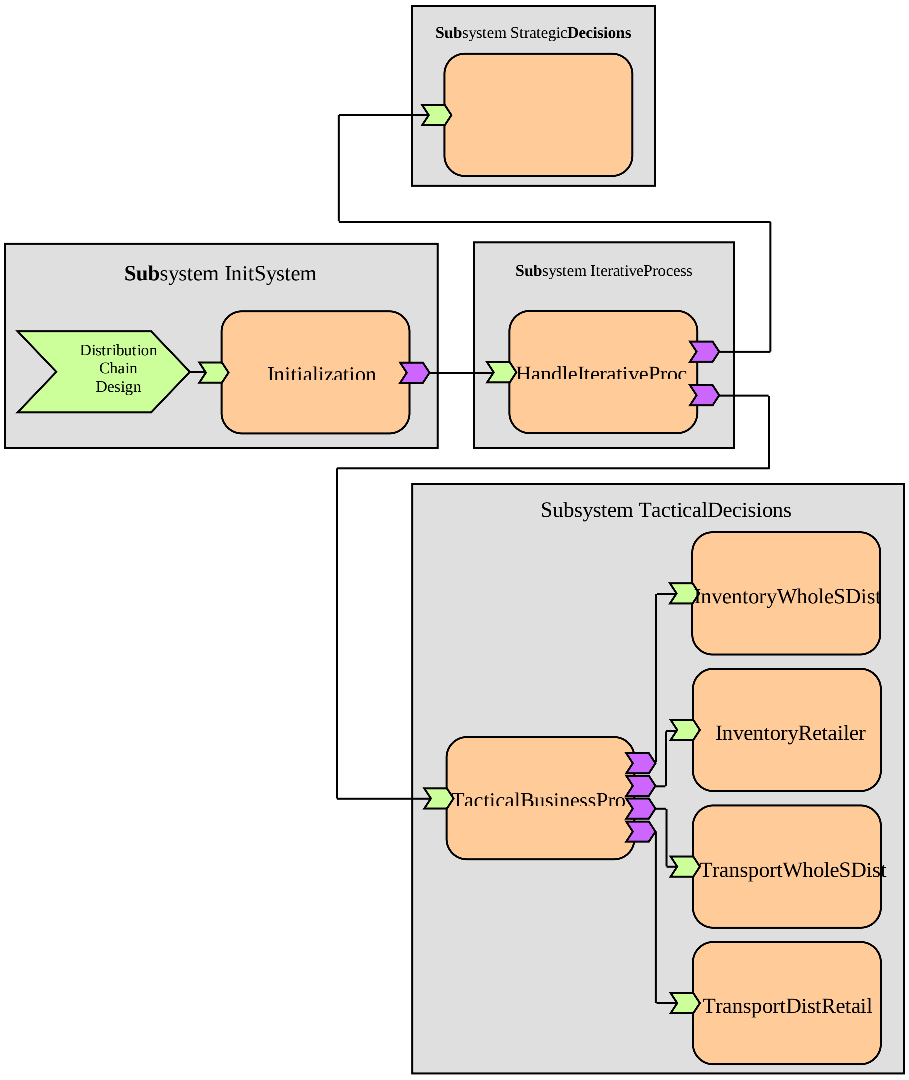


Figure 20-1: Assembly of modules

In the following subsection, we use modular (many PDFs, one PDF for each module) approach. Section 20.2 presents the pre-processor for the transition tRES; interested reader is referred to Davidrajuh (2007) for details.

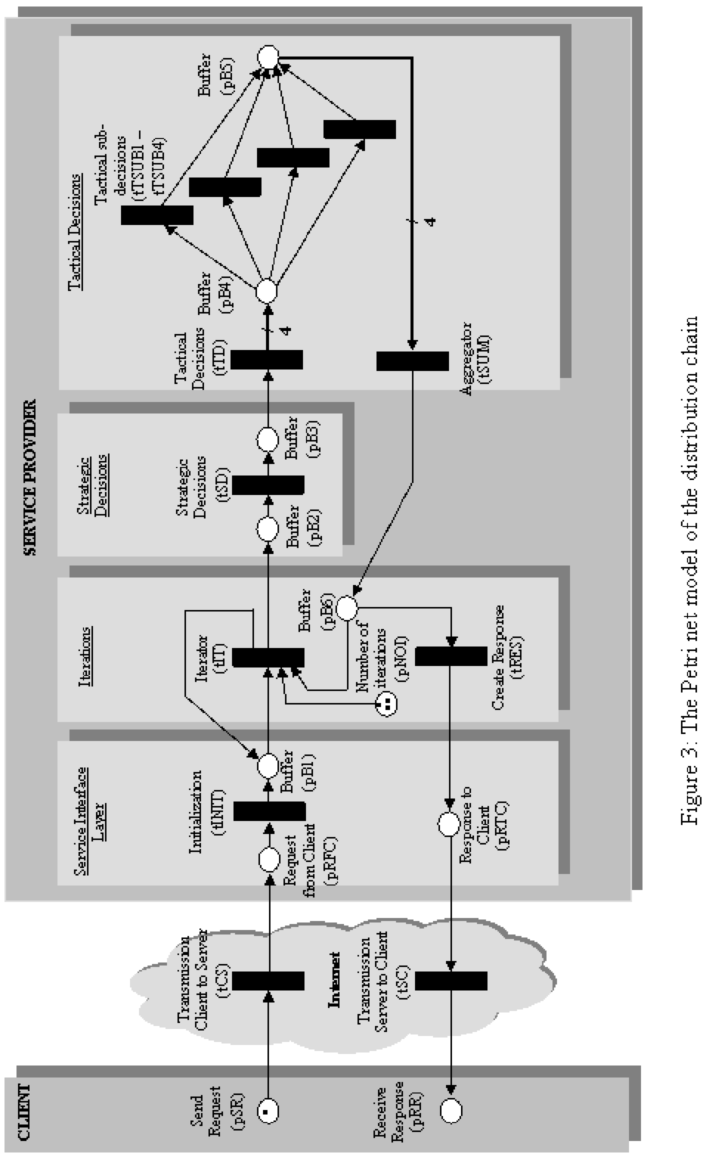


Figure 20-2: PN model of the distrubution chain

20.2 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_pdf.m'),
2) Internet transmission ('internet_pdf.m'),
3) Service Interface Layer ('sil_pdf.m'),
4) Initialization module ('init_pdf.m'),
5) Iterations module ('interate_pdf.m'),
6) Strategic module ('strategy_pdf.m'),
7) Tactical & sub tactical module (tactic_pdf.m'), and finally

8) Profile for connecting the modules together (‘conn_pdf.m’).

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

The main simulation file: ‘MIC_2006_new.m’

```
%%  Example-20: MIC – 2006 (modular model)
pns = pnstruct({'client_pdf', 'internet_pdf', 'sil_pdf',...
      'conn_pro', 'iterate_pdf', 'strategy_pdf', 'tactic_pdf'});
dyn.m0 = {'pSR',1, 'pNOI', round(3+1*randn(1)), 'pB6',1};
dyn.ft = {'tCS','5000+50*randn(1)', 'tSC','5000+50*randn(1)',...
    'tINIT','300+40*randn(1)',...
    'tRES','5+2*randn(1)', 'tSD','90+10*randn(1)',...
    'tTD','30+5*randn(1)', 'tSUB1','12+3*randn(1)',...
    'tSUB2','12+3*randn(1)', 'tSUB3','12+3*randn(1)',...
    'tSUB4','12+3*randn(1)'};
pni = initialdynamics(pns, dyn);
Results = gpensim(pni);
prnss(Results);
```

Client (‘client_pdf.m’)

```
function [pns] = client_pdf()
pns.PN_name = 'Client';
pns.set_of_Ps = {'pSR', 'pRR'};
pns.set_of_Ts = {};
pns.set_of_As = {};
```

Internet transmission (‘internet_pdf.m’)

```
function [pns] = internet_pdf()
pns.PN_name='Internet Transmission';
pns.set_of_Ps = {};
pns.set_of_Ts = {'tCS','tSC'};
pns.set_of_As = {};
```

Service Interface Layer (‘sil_pdf.m’)

```
function [pns] = sil_pdf()
pns.PN_name='Service Interface Layer';
pns.set_of_Ps = {'pRFC', 'pRTC', 'pB1'};
pns.set_of_Ts = {'tINIT'};
pns.set_of_As = {'pRFC','tINIT',1, 'tINIT','pB1',1};
```

Iterations module (‘interate_pdf.m’)

```
function [pns] = iterate_pdf()
pns.PN_name='Iterations Module';
pns.set_of_Ps = {'pNOI', 'pB6'};
pns.set_of_Ts = {'tIT','tRES'};
pns.set_of_As = {'pNOI','tIT',1, 'pB6','tIT',1, 'pB6','tRES',1};
```

Strategic module (‘strategy_pdf.m’)

```
function [pns] = strategy_pdf()
pns.PN_name = 'Strategic Module';
pns.set_of_Ps = {'pB2', 'pB3'};
pns.set_of_Ts = {'tSD'};
pns.set_of_As = {'pB2','tSD',1, 'tSD','pB3',1};
```

Tactical & sub tactical module (‘tactic_pdf.m’)

```
function [pns] = tactic_pdf()
pns.PN_name = 'Tactical & sub-tactical Module(s)';
pns.set_of_Ps = {'pB4', 'pB5'};
pns.set_of_Ts = {'tTD','tSUB1','tSUB2','tSUB3','tSUB4','tSUM'};
pns.set_of_As = {'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};
```

Module for connecting the other modules together (‘conn_pdf.m’)

```
function [pns] = conn_pdf()
pns.PN_name = 'Connections Profile';
pns.set_of_Ps = {};
pns.set_of_Ts = {};
pns.set_of_As = {'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
    };
```

20.3 Transition definition file for tRES (‘tRES_pdf.m’)

```
function [fire, trans] = tRES_pdf (trans)

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

21. Run-time PN structure

The Main Simulation File (MSF) prepares the static Petri net 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 **prnss, occupancy,** etc.

During simulations, control is passed to pre-processor if there is any. In the pre-processor, a copy of run-time PN structure is available if it is declared as a **global** variable. Then, we can inspect PN to study what's going on. Let's take a look into pre-processor for **tRES** discussed in the previous example (example-20):

```
% file: tRES_pre.m:
function [fire, transition] = tRES_pre(transition)
...
global PN; % declare PN as global variable and get access to it
PN  % dump contents of PN every time tRobot_1_pre is called

...
```

In pre-processor given above, we see that **run-time PN structure** is dumped on the screen every time **tRES** becomes enabled. 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 34 elements;** listed below are some of the elements that are part of the run-time PN structure. Some of the elements are 'STATIC' meaning that the value will not change during run time; the elements tagged as 'run-time' has values that change during simulation. Finally, elements tagged as 'OPTION' are also static – their values will not change during simulation runs; however, options are special and are explained before in section on "**OPTIONS**":

1	**STATIC**	**name: 'Stoch Example: Production facility'**
2	**STATIC**	**global_places: [1x4 struct]**
3	**STATIC**	**No_of_places: 4**
4	**STATIC**	**global_transitions: [1x3 struct]**
5	**STATIC**	**No_of_transitions: 3**
6	**run-time**	**global_Vplaces: [1x4 struct]**
7	**STATIC**	**incidence_matrix: [3x8 double]**
8	**STATIC**	**Set_of_Firing_Times: [10 5 15]**
9	**STATIC**	**PRE_exist: [1 1 1]**
10	**STATIC**	**POST_exist: [0 0 0]**
11	**STATIC**	**COMMON_PRE: 0**
12	**STATIC**	**COMMON_POST: 0**
13	**OPTION**	**MAX_LOOP: 7**
14	**OPTION**	**delta_T: 1.2500**
15	**OPTION**	**HH_MM_SS: 0**
16	**OPTION**	**PRINT_LOOP_NUMBER: 0**
17	**run-time**	**current_time: 0**
18	**run-time**	**priority_list: [0 0 0]**
19	**run-time**	**X: [0 0 0 8]**
20	**run-time**	**token_serial_numer: 8**
21	**run-time**	**resources: []**
22	**run-time**	**Firing_Transitions: [0 0 0]**
23	**run-time**	**Enabled_Transitions: [1 1 1]**

22. Petri Net Classes

There are several types of Petri Net sub-classes, based on the restrictions on the input and output places of transitions, or on the input and output transitions of places, and also on the weight of the arcs (David and Alla, 1994). We give below some the sub-classes:

- **Binary Petri net**: A Petri net is called a binary Petri net if all the weights of the arcs are 1.
- **Petri net State Machine**: A Petri net in which all the transitions have exactly one input place and one output place
- **Marked Graph (or Event Graph)**: A Petri net is which all the places have exactly one input and one output transition
- **Safe Petri nets**: A Petri net is which all the possible markings are one bounded (i.e. number of tokens in any place is at most 1)

It is easy to check the type of Petri net class with GPenSIM; the function **pnclass** does all the work. This function accepts either the static Petri net graph structure (pns) or the dynamic run-time Petri net structure (PN) as input argument.

```
% Example-22-01: Example with stochastic timing
% the main simulation file
pns = pnstruct('stoch_example_pdf');
classtype = pnclass(pns);
```

22.1 Example-22-01: Cotree with finite states

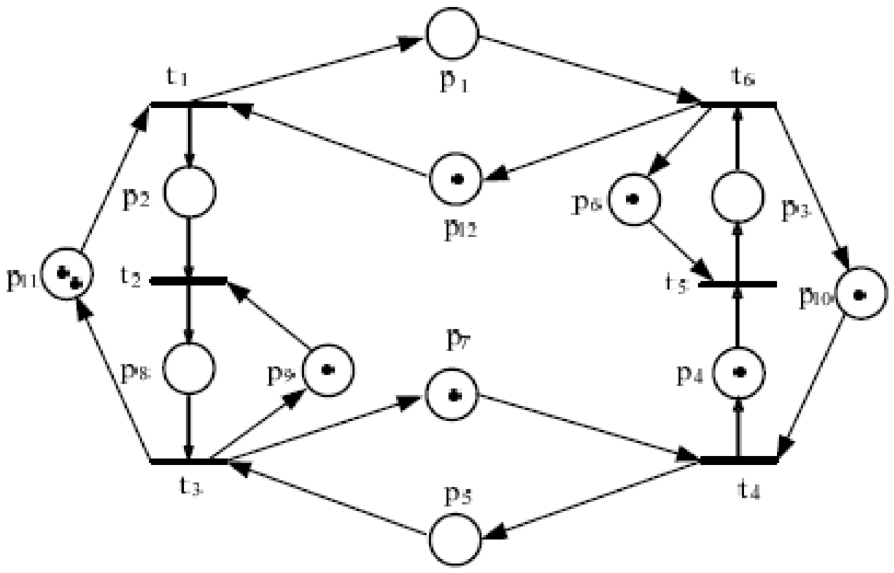


Figure 22-1: The Petri net for checking class type

We are going to check the class type of the Petri net shown above in figure-22-1. The Petri net is an event graph which is discussed further in section-24 “Marked Graph”; right now, let us check the type using GPenSIM.

PDF

```
% Example-22-01: PN Classes DEMO - 1
function [pns] = pnclass_demo1_pdf()
pns.PN_name = 'Example-22-1: Petri net Class Demo: in fig 4.12';
pns.set_of_Ps = {'p11','p2','p8','p9', 'p1','p12','p10','p3','p4','p6',...
                    'p5','p7'};
pns.set_of_Ts = {'t1','t2','t3','t4','t5','t6'};
pns.set_of_As =  {'p11','t1',1, 't1','p2',1, ...
    'p2','t2',1, 't2','p8',1, 'p8','t3',1, 't3','p11',1, ...
    't3','p9',1, 'p9','t2',1,...
    't1','p1',1, 'p1','t6',1, 't6','p12',1, 'p12','t1',1, ...
    't6','p10',1, 'p10','t4',1, 't4','p4',1, 'p4','t5',1, ...
    't5','p3',1, 'p3','t6',1, 't6','p6',1, 'p6','t5',1, ...
    't4','p5',1, 'p5','t3',1, 't3','p7',1, 'p7','t4',1};
```

The main file:

```
% Example-22-1: PN Classes DEMO - 1
pns = pnstruct('pnclass_demo1_pdf');
classtype = pnclass(pns);
```

Results:

```
This is a Binary Petri Nets
This is a Marked (Event) Graph
>>
```

23. Structural Invariants

Structural Invariants (or net invariants) are the structural properties of Petri Nets that depend only on the static (topological) structure; structural invariants are independent of the Petri net's initial markings or the dynamic run-time markings. Structural invariants are very important means of analyzing Petri nets as they allow the net's static structure to be studied independent of the dynamics activities of the net. Siphons, Traps, Place-invariants (P-invariants), and Transition-invariants (T-invariants) are some of the structural invariants (Moody and Antsaklis, 1998); see the table below for explanation.

Table 23-1: Structural invariants and their properties

Structural invariant	***Properties***
Place-invariants (P-invariants)	Place invariants are the set of places whose weighted token sum remains constant for all possible markings.
Transition-invariants (T-invariants)	Transition invariants are a set of firings that will cause a cycle in the statespace, meaning we will come back to the original state (markings)
Traps	Traps are a set of places which if become marked will always remain marked for all reachable markings of the net
Siphons	Siphons are a set of places which if become empty of tokens, will always remain empty for all reachable markings of the net.

Note: The GPenSIM functions described in this section use code from the "Petri Net Control Toolbox" by Univ. of Cagliari, Italy.

23.1 Example-23-01: Finding siphons and minimal siphons

This example uses the same Petri net as in example-15-1. The Petri net model shown in figure-15-1 is repeated below as figure-23-1. We will see soon that the Petri Net has a single siphon ('p1') and a single trap ('p2').

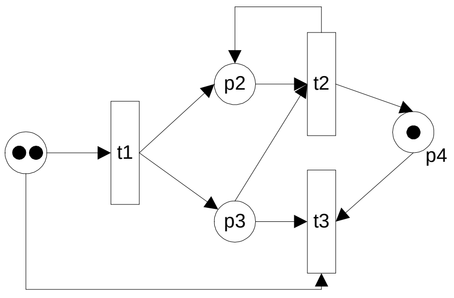


Figure 23-1: Finding siphons

Functions '**siphons_minimal'** and '**siphons'** returns a matrix of place indices, in addition to printing the siphons on the screen.

```
% Example-23-01: Siphon example
pns = pnstruct('siphon_ex01_pdf');
SM = siphons_minimal(pns);
disp(['Minimal siphons (in matrix form): [', int2str(SM), ']']);

S = siphons(pns);
disp('Siphons (in matrix form): ');
disp(S);
```

The minimal siphons and siphons are:

```
Minimal siphons in this net:
{p1}
Minimal siphons (in matrix form): [1  0  0  0]

Siphons in this net:
{p1}
{p1,p2}
{p1,p3}
{p1,p2,p4}
{p1,p3,p4}
Siphons (in matrix form):
     1     0     0     0
     1     1     0     0
     1     0     1     0
     1     1     0     1
     1     0     1     1
```

Analysis of the results: The results show that '**p1**' is a siphon, which is indeed correct as if **t1** and/or **t3** fires a couple of times to remove (siphon) all the tokens in **p1**, and once **p1** becomes empty, it will remain as empty as there isn't any transition that will deposit tokens into **t1** (**t1** is a source).

23.2 Example-23-02: Finding traps and minimal traps

Function for finding traps and minimal traps are very similar to finding siphons. Using the same net from the previous example (example-23-01), we will find traps and minimal traps here. Functions **traps_minimal** and **traps** return a matrix of place indices and also print the traps on the screen.

```
% Example-23-02: Traps example
pns = pnstruct('traps_ex02_pdf');
TM = traps_minimal(pns);
disp(['Minimal traps (in matrix form): [', int2str(TM), ']']);

T = traps(pns);
disp('Traps (in matrix form): ');
disp(T);
```

The traps and minimal traps are:

```
Minimal traps in this net:
{p2}
Minimal traps (in matrix form): [0  1  0  0]

Traps in this net:
{p2}
Traps (in matrix form):
     0     1     0     0
```

Analysis of the results: The results also show that '**p2**' is a trap, which is also correct; this is because, if **p2** gets any token by firing of **t1**, **p2** will never loose the tokens as the one that removes the tokens (**t2**) will always put it back into **p2**.

23.3 Example-23-03: Finding P-invariants and T-invariants

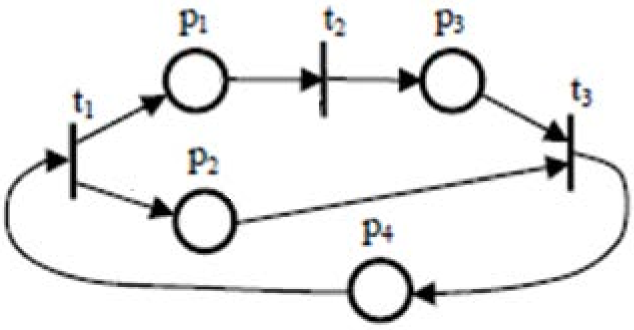


Figure 23-1: The PN for testing P- and T-invariants

PDF:

```
% Example-23-3: P-invariants & T-invariants
function [png] = ptinvar_pdf()

png.PN_name = 'P-invariants & T-invariants';
png.set_of_Ps = {'p1', 'p2', 'p3', 'p4'};
png.set_of_Ts = {'t1', 't2', 't3'};
png.set_of_As =  {...
    'p4','t1',1, 't1','p1',1, 't1','p2',1, ...  %t1
    'p1','t2',1, 't2','p3',1, ...               %t2
    'p2','t3',1, 'p3','t3',1, 't3','p4',1, ...  %t3
    };
```

MSF:

```
% Example-23-03: P/T Invariants
pns = pnstruct('cotree_example_Q6_pdf');
PI = pinvariant(pns);
TI = tinvariant(pns);
```

Simulation results:

```

P-invariants:
{p2,p4}
{p1,p3,p4}


T-invariants:
{t1,t2,t3}
```

Analysis of the results: The results also show:
P-invariants: 1) weighted sum of token in **p2** and **p4** is a constant, and 2) weighted sum of tokens in **p1**, **p3** and **p4** is also a constant.

T-invariants: no matter what the state is, if the transitions **t1**, **t2**, and **t3** are fired, then we return to the original state. This propertry will be verified in the section-25 “Firing Sequence”.

24. Minimum Cycle Time in Marked Graphs

Marked graphs (aka event graphs) are a class of Petri nets in which *all* the places have exactly one input and one output transition; mathematically, ∀p ∈ P: (•p = 1) and (p• = 1). For marked graphs, there is a simple way of finding the performance bottlenecks – the technique is called "minimum cycle time". See for example Hruz and Zhou (2007) for details. The ***minimum cycle time*** of a marked graph is give by the equation:

$$\mu = \max_i \frac{D_i}{N_i}$$

Where, D_i is the total time delay of the *i-th* directed simple cycle and N_i is the total number of tokens in this cycle, D_i/N_i is the cycle time.
The bottleneck cycle is the *j-th* one where $D_j/N_j = \mu$ holds. To remove bottleneck, we try to:

- Decrease the *D*-value: this means, improving the speed of a processes involved in that cycle (reducing the firing times of the respective transitions), and/or,
- Increase the *N*-value: add additional resources (increase the token count) in that cycle.

The technique is explained below with the help of an example.

24.1 Example-24-1: Finding Minimal-Cycle-Time

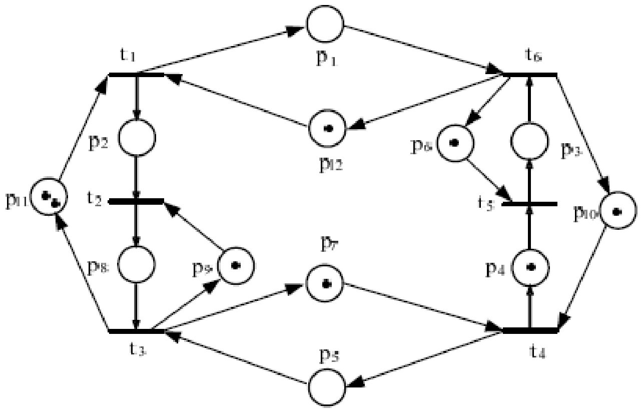


Figure 24-1: A Marked Graph for finding Minimal-Cycle-Time (taken from Hruz and Zhou, 2007)

Figure-24-1 shows a binary Petri net (because all the arc has weight equal to unity), and it is also a marked graph as all the places have exactly one input and one output transitions. This model has all together 8 cycles. The table below shows the cycles; also shown in the table is the total time delay (TD - summation of the firing times of all the transitions in that cycle), the total number of tokens on each cycle (token sum), and the cycle times.

Table 24-1: Finding Minimum-Cycle-Time

	Cycle	***Total TD***	***Token Sum***	***Cycle Time***
1.	**p10, t4, p4, t5, p3, t6**	**15**	**2**	**7.5**
2.	**p11, t1, p1, t6, p10, t4, p5, t3**	**14**	**3**	**4.7**
3.	**p1, t6, p12,t1**	**7**	**1**	**7**
4.	**p8, t3, p11, t1, p2, t2**	**6**	**2**	**3**
5.	**t2, p8, t3, p9**	**5**	**1**	**5**
6.	**t4, p4, t5, p3, t6, p12, t1, p2, t2, p8, t3, p7**	**21**	**3**	**7**
7.	**p3, t6, p6, t5**	**11**	**1**	**11**
8.	**p5, t3, p7, t4**	**7**	**1**	**7**

From the table above, we see that the bottleneck is the cycle-7 as it takes the most cycle time, which is equal to 11. Thus, if we want to increase performance of the entire system, we need to concentrate on the elements (transitions) in the cycle-7. We can, either put an extra machine of t5 and t6 (increase the tokens count N_i) and/or make the machines faster (reduce firing times of t5 and t6) in in order to reduce the delay D_i.

Let’s make use of GPenSIM to find out all these 8 cycles and calculate the other details like total time delay, token sum, and the cycle times.

PDF:
```
% Example-24-01: Finding Minimum-Cycle-Time in Marked Graph
%     This example is taken from Hruz & Zhou, fig 10.1

function [png] = marked_graph01_pdf()

png.PN_name = ‘Marked Graph’;
png.set_of_Ps = {‘p11’,’p2’,’p8’,’p9’, ‘p1’,’p12’,’p10’,’p3’,’p4’,’p6’,…
                    ‘p5’,’p7’};
png.set_of_Ts = {‘t1’,’t2’,’t3’,’t4’,’t5’,’t6’};
png.set_of_As =  {‘p11’,’t1’,1, ‘t1’,’p2’,1, …
    ‘p2’,’t2’,1, ‘t2’,’p8’,1, ‘p8’,’t3’,1, ‘t3’,’p11’,1, …
    ‘t3’,’p9’,1, ‘p9’,’t2’,1,…
    ‘t1’,’p1’,1, ‘p1’,’t6’,1, ‘t6’,’p12’,1, ‘p12’,’t1’,1, …
    ‘t6’,’p10’,1, ‘p10’,’t4’,1, ‘t4’,’p4’,1, ‘p4’,’t5’,1, …
    ‘t5’,’p3’,1, ‘p3’,’t6’,1, ‘t6’,’p6’,1, ‘p6’,’t5’,1, …
    ‘t4’,’p5’,1, ‘p5’,’t3’,1, ‘t3’,’p7’,1, ‘p7’,’t4’,1};
```

MSF:
```
% Example-24-01: Finding Minimum-Cycle-Time in Marked Graph
pns = pnstruct(‘marked_graph01_pdf’);

dyn.ft = {‘t1’,1, ‘t2’,2, ‘t3’,3, ‘t4’,4, ‘t5’,5, ‘t6’,6};
dyn.m0 = {‘p12’,1, ‘p6’,1, ‘p4’,1, ‘p10’,1, ‘p7’,1,’p9’,1, ‘p11’,2};

pni = initialdynamics(pns, dyn);

V3 = mincyctime(pni);
```

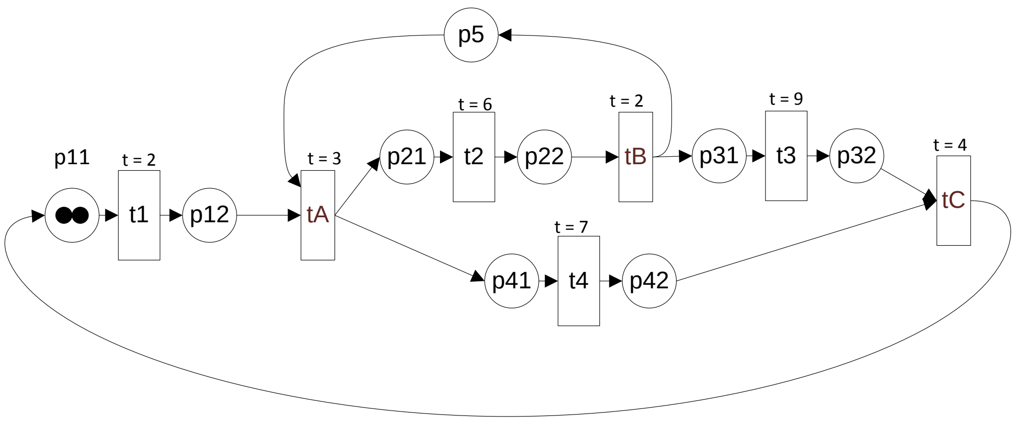
Simulation results:
```
This is a Binary Petri Nets
This is a Marked (Event) Graph

Cycle-1 :    -> p10 -> t4 -> p4 -> t5 -> p3 -> t6
TotalTD = 15    TokenSum = 2     Cycle Time = 7.5

Cycle-2 :    -> p11 -> t1 -> p1 -> t6 -> p10 -> t4 -> p5 -> t3
TotalTD = 14    TokenSum = 3     Cycle Time = 4.6667

Cycle-3 :    -> p1 -> t6 -> p12 -> t1
TotalTD = 7    TokenSum = 1     Cycle Time = 7

Cycle-4 :    -> p8 -> t3 -> p11 -> t1 -> p2 -> t2
TotalTD = 6    TokenSum = 2     Cycle Time = 3

Cycle-5 :    -> t2 -> p8 -> t3 -> p9
TotalTD = 5    TokenSum = 1     Cycle Time = 5

Cycle-6 :    -> t4 -> p4 -> t5 -> p3 -> t6 -> p12 -> t1 -> p2 -> t2 -> p8 -
> t3 -> p7
```

```
TotalTD = 21    TokenSum = 3     Cycle Time = 7

Cycle-7 :    -> p3 -> t6 -> p6 -> t5
TotalTD = 11    TokenSum = 1     Cycle Time = 11

Cycle-8 :    -> p5 -> t3 -> p7 -> t4
TotalTD = 7    TokenSum = 1     Cycle Time = 7


Minimum-cycle-time is: 11  in cycle number-7
```

24.2 Example-24-2: Finding Bottleneck in an Enterprise Information System

Figure-24-1 shows a Petri net model of an enterprise information system that involes the following:

- An entry pint (front end) client requests is represented by transition **t1** the input and out buffers are by places **p11** and **p12**,
- The database server is represented by transition **t2** the input and out buffers are by places **p21** and **p22**,
- The graphic presentation server is represented by transition **t3** the input and out buffers are by places **p31** and **p32**, and
- The CRM server is represented by transition **t4** the input and out buffers are by places **p41** and **p42**.
- Places p5 represents the availability of the database server.
- Maximum 2 client requests can be processed at a time.
- Firing times shown in the figure are given in milliseconds.



Figur 24-1: Finding Bottleneck in an Enterprise Information System

PDF:

```
% Example-24-02: Finding Minimum-Cycle-Time in Marked Graph
%     This example is taken from Hruz & Zhou, fig 10.1

function [png] = mct_ex02_pdf()

png.PN_name = ‘Bottle neck in EIS’;
png.set_of_Ps = {‘p11’,’p12’, ‘p5’, ‘p21’,’p22’,’p31’,’p32’,’p41’,’p42’};
png.set_of_Ts = {‘t1’, ‘tA’,’t2’,’t4’, ‘tB’, ‘t3’, ‘tC’};

png.set_of_As = {‘p11’,’t1’,1, ‘t1’,’p12’,1, …  %t1
    ‘p12’,’tA’,1, ‘p5’,’tA’,1, ‘tA’,’p21’,1, ‘tA’,’p41’,1, …  %tA
    ‘p21’,’t2’,1, ‘t2’,’p22’,1, … %t2
    ‘p22’,’tB’,1, ‘tB’,’p31’,1, ‘tB’,’p5’,1,… %tB
```

```
     ‘p31’,’t3’,1, ‘t3’,’p32’,1, … % t3
     ‘p41’,’t4’,1, ‘t4’,’p42’,1, … % t4
     ‘p32’,’tC’,1, ‘p42’,’tC’,1, ‘tC’,’p11’,1, …  % tC
     };
```

MSF:

```
% Example-24-02: Finding minimum-cycle-time in Event Graphs
clear all; clc;

pns = pnstruct(‘mct_ex02_pdf’);

dyn.ft = {‘t1’,2,’tA’,3,’t2’,6,’tB’,2, ‘t3’,9,’t4’,7, ‘tC’,4};
dyn.m0 = {‘p11’,2, ‘p5’,1};

pni = initialdynamics(pns, dyn);

V3 = mincyctime(pni);
```

Simulation results:

```
This is a Binary Petri Nets
This is a Marked (Event) Graph

Cycle-1 :     -> p21 -> t2 -> p22 -> tB -> p31 -> t3 -> p32 -> tC -> p11 -> t1 ->
p12 -> tA
TotalTD = 26    TokenSum = 2     Cycle Time = 13

Cycle-2 :    -> tA -> p21 -> t2 -> p22 -> tB -> p5
TotalTD = 11    TokenSum = 1     Cycle Time = 11

Cycle-3 :    -> tC -> p11 -> t1 -> p12 -> tA -> p41 -> t4 -> p42
TotalTD = 16    TokenSum = 2     Cycle Time = 8

Minimum-cycle-time is: 13  in cycle number-1
```

25. Firing Sequence

When we run simulations by calling the function gpensim, the maximal set of enabled tranitions will be fired at point of time. Some times, we may want controlled firing: we want to fire a predefined series of transitions to see how the system behaves. For example, in the second example of the section-23 “Structural Invariants”, it is said that is the transitons t1, t2, and t3 are fired one after the other, then the system will go back to the orginal state. If we want to verify this statement, then we may want to fire the series {‘t1’,’t2’,’t3’} in that strict order; the function “**firingseq**” will exactly do that.

The function **firingseq** is similar to the function **gpensim**; the only difference being that gpensim allow the maximal set of enabling transitions to fire at any point of time, whereas firingseq follows the given firing sequence strictly.

Usage:

```
sim = firingseq(pni, firing_seq, repeat_seq, allow_parallel);
```

Function firingseq takes at least two input arguments: pni is the marked Petri Net, and firing_seq is the given sequence of transitions. In addition, two other optional parameters may be given: repeat_seq is a Boolean value indicating the firing_seq has to be repeated, and allow_parallel is another boolen value indicating whether to permit parallel (overlapping) firing.

25.1 Example-25-01: Verfying T-invariants

In the example-23-03, we’ve seen the transitions {‘t1’, ‘t2’, ‘t3’} as a T-invariant meaning if these transitions are fired then we will go back to the original state. Let’s verify this statement.

First of all, let’s allow the initital tokens for the places be random; then we will enforce the transitions {‘t1’, ‘t2’, ‘t3’} to be fired strictly in that sequence.

MSF:

```
% Example-25-01: P-invariants & T-invariants
clear all; clc;

global global_info;
global_info.STOP_AT = 13;

pns = pnstruct(‘firingseq_ex01_pdf’);

m0p1 =  ceil(unifrnd(0, 10));
m0p2 =  ceil(unifrnd(0, 10));
m0p3 =  ceil(unifrnd(0, 10));
m0p4 =  ceil(unifrnd(0, 10));

dyn.m0 = {‘p1’,m0p1, ‘p2’,m0p2, ‘p3’,m0p3, ‘p4’,m0p4};
dyn.ft = {‘t1’,1, ‘t2’,2, ‘t3’,3};
pni = initialdynamics(pns, dyn);

firing_seq = {‘t1’, ‘t2’, ‘t3’};

sim = firingseq(pni, firing_seq, 1);

prnss(sim);
plotp(sim, {‘p1’, ‘p2’, ‘p3’, ‘p4’});
```

Simuation results:

```
Simulation of “Example-25-01: firing Sequence”:
 ======= State Diagram =======
**    Time: 0    **
State:0 (Initial State): 7p1 + p2 + 9p3 + 10p4
At start ….
At time: 0,  Enabled transitions are:    t1   t2   t3
At time: 0,  Firing transitions are:     t1

**    Time: 1    **
State: 1
Fired Transition: t1
Current State: 8p1 + 2p2 + 9p3 + 9p4
Virtual tokens: (no tokens)

Right after new state-1 ….
At time: 1,  Enabled transitions are:    t1   t2   t3
At time: 1,  Firing transitions are:     t2


**    Time: 3    **
State: 2
Fired Transition: t2
Current State: 7p1 + 2p2 + 10p3 + 9p4
Virtual tokens: (no tokens)

Right after new state-2 ….
At time: 3,  Enabled transitions are:    t1   t2   t3
At time: 3,  Firing transitions are:     t3


**    Time: 6    **
State: 3
Fired Transition: t3
Current State: 7p1 + p2 + 9p3 + 10p4
Virtual tokens: (no tokens)

Right after new state-3 ….
At time: 6,  Enabled transitions are:    t1   t2   t3
At time: 6,  Firing transitions are:     t1


**    Time: 7    **
State: 4
Fired Transition: t1
Current State: 8p1 + 2p2 + 9p3 + 9p4
Virtual tokens: (no tokens)

Right after new state-4 ….
At time: 7,  Enabled transitions are:    t1   t2   t3
At time: 7,  Firing transitions are:     t2


**    Time: 9    **
State: 5
Fired Transition: t2
Current State: 7p1 + 2p2 + 10p3 + 9p4
Virtual tokens: (no tokens)

Right after new state-5 ….
```

```
At time: 9,  Enabled transitions are:    t1   t2   t3
At time: 9,  Firing transitions are:     t3


**    Time: 12   **
State: 6
Fired Transition: t3
Current State: 7p1 + p2 + 9p3 + 10p4
Virtual tokens: (no tokens)

Right after new state-6 ….
At time: 12,  Enabled transitions are:    t1   t2   t3
At time: 12,  Firing transitions are:     t1
```

The simualtions show that the initial state is: `7p1 + p2 + 9p3 + 10p4`.
After the firing sequence of {'t1', 't2', and 't3'}, at time= 06, we go back to the original state as state-3. Again, after another firing sequence of {'t1', 't2', and 't3'}, at time= 12, we again go back to the original state as state-6.

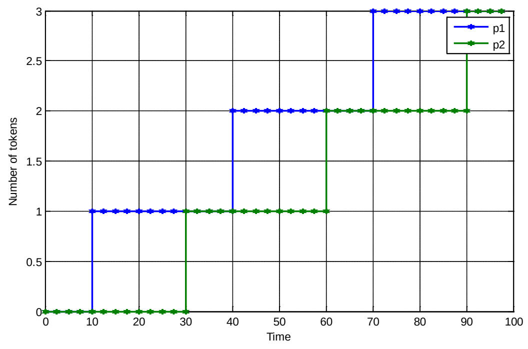
25.2 Example-25-02: Load Balance with Firing Sequence

In example-09-01, we made two transitions (tX1 and tX2) to fire alternatively using binary semafor. Later, in the example-17-01, we achieved the same result (alternating firing of tX1 and tX2) using priorities. In this section, we will achieve the same results much effortlessly: we will use the function **firingseq**, with {'tX1', 'tX2'} as the firing sequence that is to be repeated.

MSF:

```
% Example-25-02: Load Balance with Firing Sequence
clear all; clc;

global global_info;
global_info.STOP_AT = 100;      % GLOBAL DATA: binary semafor

pns = pnstruct('firingseq_ex02_pdf');

dyn.m0 = {'pSTART', 10};
dyn.ft = {'tX2',20, 'tX1',10, };
pni = initialdynamics(pns, dyn);

sim = firingseq(pni, {'tX1', 'tX2'}, 1); % 1 means repeat
plotp(sim, {'p1', 'p2'}, 0, 2);
```

Simualtion results:



Figur 25-1: Cool!

Part-II: Coloring Tokens

26. Coloring Token

So far, we have treated tokens inside a place as indistinguishable: all the tokens inside a place are homogenous (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.
However, from now on, we can make each token as a unique one, identifiable with a unique token ID, and also we can add some tags ('colors') to each token.

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

1. **Only transitions can manipulate colors:** in Pre-processor, one can add, delete, or alter colors of the output tokens.
2. **By default, colors are inherited**: that is when a transition fires, it collects all the colors from the consumed (input) tokens and then it passes these colors to the deposited (output) tokens. However, color inheritance can be prevented by **overriding**
3. An enabled transition can select **specific input tokens based on preferred colors**
4. An enabled transition can select **specific input tokens based on time; e**.g. the time the tokens are created
5. Structure of tokens: tokens have a unique tokID number, in addition, creation time, and a set of colors.

26.1 Structure of Tokens

A token has a structure consisting of 3 elements:

1. **tokID** (integer value): a unique token ID
2. **creation_time** (real 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: 30.25
      t_color: {'Tamil', 'Norsk', 'English'}
```

26.2 Functions for selection of tokens based on their colors

Table below shows the GPenSIM functions that are used for color manipulation:

Table 26-1: GPenSIM Functions for manipulation of token color

Function	*Description*
`tokenAllColor`	Select tokens with **all** of the specific colors from a specific place
`tokenAny`	Select any tokens (without any preference on color) from a specific place
`tokenAnyColor`	Select tokens with **any** of the specific colors (t_color) from a specific place (placeI); selected tokens must have at least one of the specified color
`tokenArrivedBetween`	Select tokens that arrived in a specific place between the time intervals (discussed in section-30 "Token Selection based on Time")
`tokenArrivedEarly`	Select tokens that arrived earliest to a specific place (discussed in section-30 "Token Selection based on Time")
`tokenArrivedLate`	Select tokens that arrived latest to a specific place (discussed in

	section-30 “Token Selection based on Time”)
tokenColorless	Select only the colorless tokens (tokens with NO color) from a specific place
tokenEXColor	Select tokens with **exact** colors (no more or no less) from a specific place
tokenWOAllColor	Exclude a token ONLY if its color contains all of the specified colors
tokenWOAnyColor	Exclude a token ONLY if its color contains ANY of the specified colors
tokenWOEXColor	Exclude a token ONLY if it has **exact** colors as specified
tokIDs	Returns a set of tokIDs of tokens in a place; if the second argument ‘nr_tokIDs_wanted’ is not specified, then tokIDs of all the tokens in the place is returned.
prnfinalcolors	This function returns colors of the final tokens; final tokens are the tokens that left in places when simultion was stopped or completed. In addition to the first input argument simulation results, the optional second input argument limits the places we are interested in.
prncolormap	This function returns colors of all the toeksn that were in different places during the simulations. the final tokens; the optional second input argument limits the places we are interested in.

The functions that are available for selecting tokens based on their colors usually take three input arguments and returns two output arguments.

- The input arguments: first we have to identify the place (place) from which we are going to select the tokens; the second argument is the number of tokens wanted (nr_tokens_wanted) with the specified color, and finally, the specific colors are defined in the third input argument (t_color)
- The output arguments: first one output argument (set_of_tokID) is a set of tokIDs; the length of the set is equal to the input argument the nr_tokens_wanted; the second output argument (nr_token_av) is the number of valid tokIDs available in the set set_of_tokID: the set may have trailing zeros just to make its length equal to nr_tokens_wanted.

The using the functions:

- ‘**tokenAnyColor**’: this function returns a set of tokens (tokIDs) that has any of the specified colors. Usage:
  [set_of_tokID, nr_token_av] = tokenAnyColor(place, nr_tokens_wanted, t_color)
  Ex: if we want 3 tokens (nr_tokens_wanted =3) from the place p1 (place =’p1’), where each token should contain **at least one** of the colors ‘A’, ‘B’, or ‘C’ (t_color ={‘A’, ‘B’, ‘C’}).

- ‘**tokenAllColor**’: this function returns a set of tokens where each token color consists of all the specified colors. Usage:
  [set_of_tokID, nr_token_av] = tokenAllColor(placeI, nr_tokens_wanted, t_color)
  Ex: we want 1 token (nr_tokens_wanted =1) from the place p1 (place =’p1’), where each token should contain *at least* **all** of the colors ‘A’, ‘B’, or ‘C’ (t_color ={‘A’, ‘B’, ‘C’}).

- ‘**tokenEXColor**’: (**EX** standsfor ‘exact’) this function returns a set of tokens where each token has exactly the same color set as specified. Usage:
  [set_of_tokID, nr_token_av] = tokenEXColor(placeI, nr_tokens_wanted, t_color)
  Ex: we want tokens with colors exactly {‘A’, ‘B’}; the color of the tokens must not contain any more or less colors.

- ‘**tokenAny**’: this function just returns a set of tokens regardless of their colors. Usage:

[set_of_tokID, nr_token_av] = tokenAny(place, nr_tokens_wanted)

- **'tokenColorless':** this function returns a number of tokens that are colorless (note: only 2 input arguments). Usage:
  [set_of_tokID, nr_token_av] = tokenColorless(place, nr_tokens_wanted)

- '**tokenWOAnyColor**': (**WO** standsfor 'without') this function returns a set of tokens (tokIDs) **excluding** the ones that has **any** of the specified colors. Usage:
  [set_of_tokID, nr_token_av] = tokenWOAnyColor(place, nr_tokens_wanted, t_color)
  Ex: we don't want tokens containing **any** of the colors specified in t_color; all others tokens are acceptable

- '**tokenWOAllColor**': this function returns a set of tokens (tokIDs) **excluding** the ones that has **all** of the specified colors. Usage:
  [set_of_tokID, nr_token_av] = tokenWOAllColor(place, nr_tokens_wanted, t_color)
  Ex: we don't want token that posses **all** of the colors specified in t_color; all others tokens are acceptable

- '**tokenWOEXColor**': (EX standsfor 'exact') this function returns a set of tokens (tokIDs) **excluding** the ones that has **exactly the same colors** as specified. Usage:
  [set_of_tokID, nr_token_av] = tokenWOEXColor(place, nr_tokens_wanted, t_color)
  Ex: we don't want token that posses **exactly** the same colors specified in t_color; all others tokens are acceptable

- '**tokIDs**': this function returns a set of tokens (tokIDs) from a place, regardless of color (note: this function may take either one or two input arguments). If the second input argument 'nr_tokens_wanted' is not specified, then the tokIDs of all the tokens in the place will be returned. Usage:
  [set_of_tokID, nr_token_av] = tokIDs(place, nr_tokens_wanted)

26.3 Color Inheritance

In GPenSIM, colored tokens can only utilized by transitions; since transitions are active, transition definition files (specific and COMMON pre files) can be coded with manipulating token colors:

1. When a transition fires, **it can choose input tokens with specific colors**
2. When a transition fires, **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**.
3. When new tokens are deposited into the output place, **new colors can be added by the transition**. This new color will be added 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)

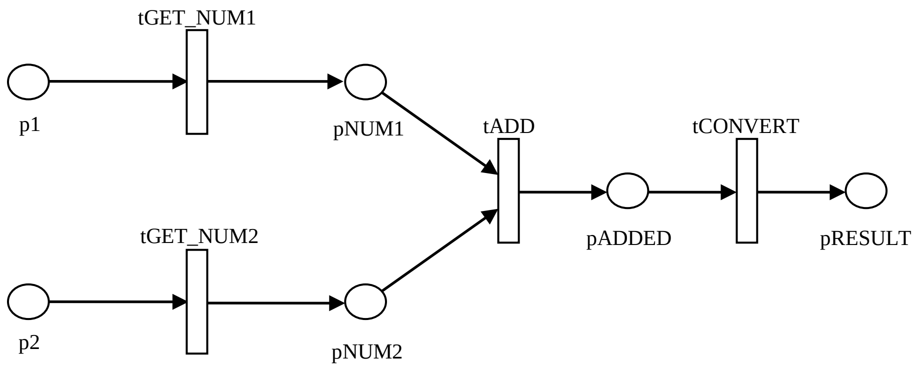
Let us experiment coloring with the help of a simple example candidly called 'simple_adder'.

26.4 Example-26-1: Simple Adder

This example presents an "adder" that adds two (different) numbers input by the user. **Note: this adder will not function properly if the two input numbers are same.**

Petri net model of a simple adder shown in figure-26-1 has 6 places and 4 transitions. Places **p1** and **p2** are just to keep the initial tokens so that the transitions **tGET_NUM1** and **tGET_NUM2** can fire only once. 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’}).



Figur 26-1: Simple Adder

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**

MSF: ’simple_adder.m’

```
% Example-26-1: SIMPLE ADDER with colored tokens
pns = pnstruct('simple_adder_pdf');
dyn.m0 = {'p1',1, 'p2',1};
pni = initialdynamics(pns, dyn);

disp('Enter two DIFFERENT numbers:');
sim = gpensim(pni);

prnfinalcolors(sim);
```

PDF: ’simple_adder_pdf.m’

```
% Example-26-1: SIMPLE ADDER with colored tokens
function [pns] = simple_adder_pdf()
pns.PN_name = 'Color example: Simple Adder';
pns.set_of_Ps = {'p1', 'p2', 'pNUM1', 'pNUM2', 'pADDED','pRESULT'};
pns.set_of_Ts = {'tGET_NUM1','tGET_NUM2','tADD','tCONVERT'};
pns.set_of_As = {'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};
```

Pre-processor: ’tGET_NUM1_pre.m’
The pre-processor will ask the user to input a number:

```
function [fire, transition] = tGET_NUM1_pre (transition)
num1 = input('input number-1: ');
transition.new_color = num2str(num1);
fire = 1;  % let it fire
```

Pre-processor: ’tGET_NUM2_pre.m’
This will ask the user to input another number; otherwise same as **tGET_NUM1_pre**

Pre-processor: ’tADD_pre.m’
There is no need for tADD_pre. By default, it will inherit colors from input tokens and put the colors to the output token.

Pre-processor: ’tCONVERT_pre.m’

```
function [fire, transition] = tCONVERT_pre (transition)

tokID = tokenAny('pADDED', 1); % select a token
colors = get_color(tokID); % get the colors of the selected token

num1 = str2num(colors{1}); % convert the color-1 into number1
num2 = str2num(colors{2}); % convert the color-2 into number2

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

fire = 1;  % let it fire
```

Simulation Results
The statement, `prncolormap(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

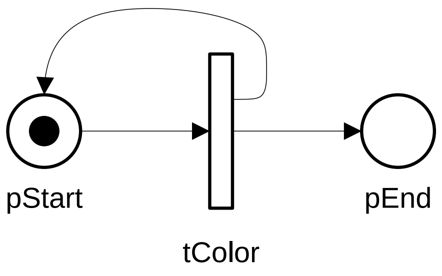
```
input number-1: 21
input number-2: 45

Colors of final Tokens:

No. of final tokens: 1
    'Time:'    '15'    ' Place:'    'pRESULT'    ' Colors:'    '66'
```

27. Color Pollution

Inhering colors - from input tokens and then passing to the output tokens - can cause expected results. The following example explains:

27.1 Example-27-1: Color Pollution

In the figure-27-1 shown below, **tColor** is to deposit 3 tokens into **pEnd** each with different colors such as "RED", "GREEN", and "BLUE".



Figur 27-1: Color Pollution

However, as the simulation results show that two of the three tokens in **pEnd** has the more than one colors.

```
Colors of final Tokens:

No. of final tokens: 4
    'Time:' '10'    'Place:' 'pEnd'     ' Colors:' 'RED'
    'Time:' '20'    'Place:' 'pEnd'     ' Colors:' 'GREEN' 'RED'
    'Time:' '30'    'Place:' 'pEnd'     ' Colors:' 'BLUE'  'GREEN'  'RED'
    'Time:' '30'    'Place:' 'pStart'   ' Colors:' 'BLUE'  'GREEN'  'RED'
```

The transition **tColor** is supposed to one color each to otherwise colorless input token from **pStart:** When **tStart** fires for the very first time, it takes a token (the only token) from **pStart**; this token is the initial token from the program start, thus colorless. **tColor** add color "RED" to this token and by firing, this token is deposited into **pEnd**; firing of **tColor** also deposits a token with color "RED" back into **pStart**. When **tStart** fires again, it takes the token (which has the color "RED") from **pStart** and then adds the color "GREEN"; now the token has two colors "RED" and "GREEN", which is going to be deposited into **pEnd** as well as into **pStart**. This is not what we wanted. We wanted a sequence of tokens into **pEnd** with a single colors either "RED", "GREEN", or "BLUE".

This color pollution is a result of simple negligence; by setting "override =1" will prevent inheritance thus this color pollution will not happen. Shown below is the simulation result, after activating override in the Pre-processor for **tColor**; the results show that the tokens in pEnd has only one color each as intented.

```
Colors of final Tokens:

No. of final tokens: 4
    'Time:'  '10'    'Place:' 'pEnd'     ' Colors:' 'RED'
    'Time:'  '20'    'Place:' 'pEnd'     ' Colors:' 'GREEN'
    'Time:'  '30'    'Place:' 'pEnd'     ' Colors:' 'BLUE'
    'Time:'  '30'    'Place:' 'pStart'   ' Colors:' 'BLUE'
```

MSF and Pre-processor for tColor are given below.

MSF:
```
% Example-27-1: Color Pollution Example

clear all; clc;

global global_info;
global_info.THREE_COLORS = {'RED', 'GREEN', 'BLUE'};

png = pnstruct('color_poll_pdf');

dyn.m0 = {'pStart',1};
dyn.ft = {'tColor',10};
pni = initialdynamics(png, dyn);
sim = gpensim(pni);

%%%% PRINT RESULTS %%%%%
plotp(sim, {'pEnd'});
prnfinalcolors(sim);
```

Pre-processor for tColor (‘tColor_pre.m’):
```
% Example-27-1: Color Pollution Example

function [fire, transition] = tColor_pre (transition)

global global_info;

tColor = get_trans('tColor');
tColorFired = tColor.times_fired; % how many times tColor has already fired

% tColor should not fire more than 3 times
if eq(tColorFired, 3),
    global_info.STOP_SIMULATION = 1;
    fire = 0;
    return
end;

new_color = global_info.THREE_COLORS{tColorFired+1};
transition.new_color = new_color;

%%%%%% override
transition.override = 1;
fire = 1;
```

28. Token Selection based on Color

A transition may select input tokens based on color. This is done by executing any of the functions mentioned in the previous section (**tokenAnyColor**, **tokenAllColor**, **tokenEXColor**, **tokenAny**, **tokenColorless**, **tokenWOAnyColor**, **tokenWOAllColor**, **tokenWOEXColor**).

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

```
[set_of_tokIDs, nr_tokens_av] = tokenAnyColor('pBUFF',4,{'Color-A'});
```

The returned value (`set_of_tokIDs`) is a set of **tokID** consisting of **tokID** of 0-4 tokens (e.g. `set_of_tokIDs = [54 98 0 0]);` the `set_of_tokIDs` has trailing zeros, only the first and second tokIDs are valid (not zero); the second output parameter (`nr_tokens_av`) has a value of 2, which also indicates that only the first and second tokIDs in the `set_of_tokIDs` are valid. If there are no tokens with color ‘Color-A’ in the place ‘pBUFF’, then the returned **set_of_tokIDs** is will be an arry of 4 zeros (because the input parameter indicates that we wanted 4 tokens).

28.1 Example-28-1: Selecting Input Tokens with Specific Color

Figure given below depicts a production process. Transition **tGEN** represents a machine that rotationally produces 4 types of products, ‘A’, ‘B’, ‘C’, and ‘X’. Though buffer **pBUFF** contains all four types of products, **tA** is supposed to select ‘A’ products only. Similarly, **tB** selects ‘B’ products and **tC** selects ‘C’ products only.

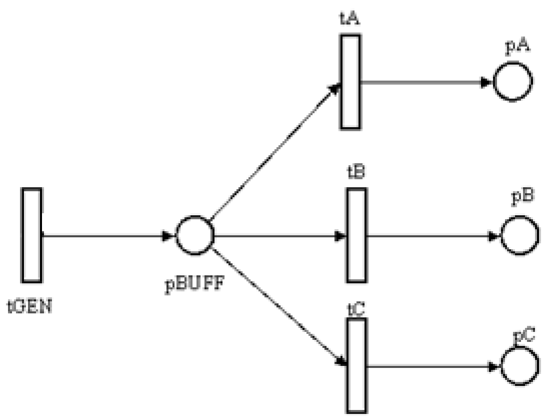


Figure 28-1: Selecting input tokens based on color

MSF:

```
% Example-28-1: Color Selection
global global_info;
global_info.STOP_AT = 8.8; % stop at 8.8 time units

% for color generation
global_info.cr = {'A', 'B', 'C', 'X'}; % color rotation 'A'-'X'
global_info.cr_index = 0; % initially color rotation index = 0

pns = pnstruct('select_color_pdf');

%%%% DYNAMIC DETAILS %%%%
dyn.ft = {'tGEN',1, 'allothers',0.1};
pni = initialdynamics(pns, dyn);
sim = gpensim(pni);
```

```
plotp(sim, {'pA', 'pB', 'pC'});
prnfinalcolors(sim);
```

PDF:

```
% Example-28-1: Color Selection
function [pns] = select_color_pdf()
pns.PN_name = 'EX-28-1: SELECT COLOR Example';
pns.set_of_Ps = {'pBUFF', 'pA', 'pB', 'pC'};
pns.set_of_Ts = {'tGEN', 'tA', 'tB', 'tC'};
pns.set_of_As = {'tGEN','pBUFF',1, 'pBUFF','tA',1,'tA','pA',1,...
    'pBUFF','tB',1,'tB','pB',1, 'pBUFF','tC',1,'tC','pC',1};
```

COMMON_PRE:

COMMON_PRE is a summation of **tGEN_pre** and the pre fioles for **tA**, **tB**, and **tC**. tGEN is to produce tokens with a color: 'A', 'B', 'C', or 'X'; whereas, the other transiitons tA, tB, and tC should select tokens with color 'A', 'B', or 'C', respectively.

```
function [fire, transition] = COMMON_PRE (transition)

global global_info;

if strcmp(transition.name, 'tGEN'),
    index = mod(global_info.cr_index, 4)+1;
    global_info.cr_index = global_info.cr_index + 1;
    transition.new_color = global_info.cr(index);
    fire = 1;
    return;
end;

if strcmp(transition.name, 'tA'),
    tokID1 = tokenAnyColor('pBUFF',1,{'A'});
elseif strcmp(transition.name, 'tB'),
    tokID1 = tokenEXColor('pBUFF',1,{'B'});
elseif strcmp(transition.name, 'tC'),
    tokID1 = tokenAllColor('pBUFF',1,{'C'});
else
    % not possible to come here
end;

transition.selected_tokens = tokID1;
fire = (tokID1);
```

Simulation results

Printout below shows that **pA, pB,** and **pC**, possess tokens with color 'A', 'B', or 'C', respectively. Hence, tokens with color 'X', stays in the place **pBUFF**.

```
Colors of final Tokens:
No. of final tokens: 8

    'Time:'    '1.125'    ' Place:'    'pA'    ' Colors:'    'A'
    'Time:'    '5.175'    ' Place:'    'pA'    ' Colors:'    'A'
    'Time:'    '2.15'    ' Place:'    'pB'    ' Colors:'    'B'
    'Time:'    '6.175'    ' Place:'    'pB'    ' Colors:'    'B'
    'Time:'    '4.075'    ' Place:'    'pBUFF'    ' Colors:'    'X'
    'Time:'    '8.075'    ' Place:'    'pBUFF'    ' Colors:'    'X'
    'Time:'    '3.175'    ' Place:'    'pC'    ' Colors:'    'C'
    'Time:'    '7.175'    ' Place:'    'pC'    ' Colors:'    'C'
```

28.2 Example-28-2: Required or Preferred Color?

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

In the previous example, we forced the transition **tA** to select a token with color 'A'. Function `tokenAnyColor` will return a tokID if a token is with 'A' color is available or else returned tokID value will be zeros ('[0]'). 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*** 'A' tokens. This means, if 'A' tokens are available, they will be consumed; if not, one of the other existing tokens of 'B', 'C', or 'X' will be consumed. In the newer pre-processor given below **tA** prefers (rather than forcing) 'A' tokens, whereas **tB** and **tC** are very selective as before:

```
% Example-28-2: tA prefers color 'A', but others colors are acceptable
function [fire, transition] = COMMON_PRE (transition)

global global_info;

if strcmp(transition.name, 'tGEN'),
    index = mod(global_info.cr_index, 4)+1;
    global_info.cr_index = global_info.cr_index + 1;
    transition.new_color = global_info.cr(index);
    fire = 1;
    return;
end;

if strcmp(transition.name, 'tA'),
    tokID1 = tokenAnyColor('pBUFF',1,{'A'});
    transition.selected_tokens = tokID1;
    % tA prefers color 'A', but others colors are also acceptable
    fire = 1;
    return
elseif strcmp(transition.name, 'tB'),
    tokID1 = tokenEXColor('pBUFF',1,{'B'});
elseif strcmp(transition.name, 'tC'),
    tokID1 = tokenAllColor('pBUFF',1,{'C'});
else
    % not possible to come here
end;

transition.selected_tokens = tokID1;
% tB and tC demand color 'B' or 'C', respectively
fire = (tokID1);
```

The transition **tA** always fires if enabled (because fire=1), regardless of 'A' tokens are available or not. It will also consume '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.

Simulations show that now **pA** has tokens with many colors, whereas **pB** and **pC** have only specific colors.

```
Colors of final Tokens:

No. of final tokens: 8
    'Time:'    '1.125'    ' Place:'    'pA'    ' Colors:'    'A'
    'Time:'    '4.175'    ' Place:'    'pA'    ' Colors:'    'X'
    'Time:'    '5.175'    ' Place:'    'pA'    ' Colors:'    'A'
    'Time:'    '8.175'    ' Place:'    'pA'    ' Colors:'    'X'
    'Time:'    '2.15'    ' Place:'    'pB'    ' Colors:'    'B'
    'Time:'    '6.175'    ' Place:'    'pB'    ' Colors:'    'B'
    'Time:'    '3.175'    ' Place:'    'pC'    ' Colors:'    'C'
    'Time:'    '7.175'    ' Place:'    'pC'    ' Colors:'    'C'
```

28.3 Example-28-3: Using the color selction functions "tokenWO…"

In this example, we study about the usage of the functions **tokenWOAnyColor**, **tokenWOAllColor**, and **tokenWOEXColor**. Figure-28-3 shows the petri net model which stepwise filter tokens based on their colors. **tColor** produces tokens with color that consists of different combination of 'A', 'B', 'C', 'X', 'Y', and 'Z'. The Petri net model should filter these tokens in order to capture tokens that possess either the three colors 'A', 'B', 'C', or the three colors 'X', 'Y', and 'Z'.

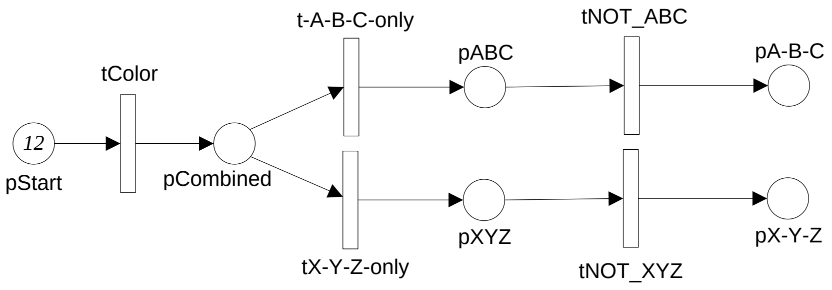


Figure 28-2: Using "tokenWO…" functions as filters

In the Petri net model:

- **tColor** produces tokens with any combination of A-B-C-X-Y-Z, and dumps into **pCombined**.
- **tA-B-C-only** is a filter that transfers only the tokens with color combination A-B-C into **pABC**. Similarly, **tX-Y-Z-only** is also a filter that transfers only the tokens with color combination X-Y-Z into **pXYZ**. This means, tokens with mixed colors will be left in **pCombined**.
- **pABC** contains tokens with any combination of A-B-C. **tNOT_ABC** filter away tokens that do not consists of all three 'A', 'B', and 'C' colors, into **pA-B-C**, so that the tokens with all three 'A', 'B', 'C' remains in **pABC**. Similarly, **pXYZ** contains tokens with any combination of 'X', 'Y', and 'Z'. **tNOT_XYZ** filters away tokens that do not consists of all three 'X', 'Y', and 'Z' colors, into **pX-Y-Z**, so that the tokens with all three 'X', 'Y', 'Z' remains in **pXYZ**.

The simulation results show that pABC is left with token that possess the colors 'A', 'B', and 'C'; whereas pA-B-C has tokens with other combinations (but not all three) of 'A', 'B', and 'C'.

```
   'Time:'   '2'    ' Place:' 'pABC'    ' Colors:' 'A'    'B'    'C'
```

```
    'Time:'  '13'    ' Place:' 'pA-B-C'  ' Colors:' 'A'    'B'
    'Time:'  '14'    ' Place:' 'pA-B-C'  ' Colors:' 'C'
```

Similarly, pXYZ has a token that possesses all three colors of 'X', 'Y', and 'Z'; whereas pX-Y-Z has tokens with other combinations (but not all three) of 'X', 'Y', and 'Z'.

```
'Time:'  '3'    ' Place:'  'pXYZ'    ' Colors:' 'X'    'Y'    'Z'
```

```
    'Time:'  '9'    ' Place:'  'pX-Y-Z'  ' Colors:' 'X'
    'Time:'  '10'   ' Place:'  'pX-Y-Z'  ' Colors:' 'X'    'Y'
```

Finally, pCombined is left with tokens that has cross-colors, both from A-B-C and X-Y-Z.

```
'Time:' '3'   ' Place:' 'pCombined'   ' Colors:' 'A'  'B'  'C'  'X'  'Y'
'Time:' '4'   ' Place:' 'pCombined'   ' Colors:' 'A'  'B'  'C'  'X'
'Time:' '5'   ' Place:' 'pCombined'   ' Colors:' 'A'  'X'  'Y'  'Z'
'Time:' '6'   ' Place:' 'pCombined'   ' Colors:' 'A'  'C'  'X'  'Y'  'Z'
'Time:' '9'   ' Place:' 'pCombined'   ' Colors:' 'A'  'X'  'Y'
'Time:''10'   ' Place:' 'pCombined'   ' Colors:' 'A'  'B'  'Y'
```

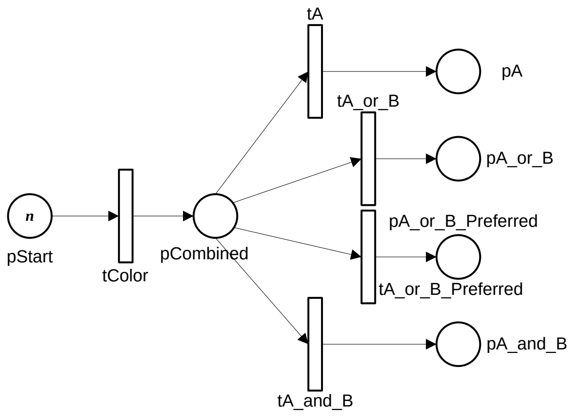
Note: This example is somewhat unecessary, as we could have made compact and elegant model by using **tokenAnyColor** and **tokenEXColor**. However, this example is for **token<u>WO…</u>**, so only the **token<u>WO…</u>** functions are used here. The COMMON_PRE is given below:

```
function [fire, transition] = COMMON_PRE (transition)
global global_info;

if strcmpi(transition.name, 'tColor'),
    global_info.counter = global_info.counter + 1;
    switch global_info.counter
        case {1}
            transition.new_color = {'A','B','C'};
        case {2}
            transition.new_color = {'X','Y','Z'};
        case {3}
            transition.new_color = {'A','B','C','X','Y'};
        case {4}
            transition.new_color = {'A','B','C','X'};
        case {5}
            transition.new_color = {'X','Y','Z','A'};
        case {6}
            transition.new_color = {'X','Y','Z','C','A'};
        case {7}
            transition.new_color = {'X'};
        case {8}
            transition.new_color = {'X','Y'};
        case {9}
            transition.new_color = {'A','X','Y'};
        case {10}
            transition.new_color = {'A','B','Y'};
        case {11}
            transition.new_color = {'A','B'};
        case {12}
            transition.new_color = {'C'};
        otherwise
            fire = 0; return;
    end;
    fire = 1; return;
end;

if strcmpi(transition.name, 'tA-B-C-only'),
    tokID1 = tokenWOAnyColor('pCombined',1, {'X','Y','Z'});
elseif strcmpi(transition.name, 'tX-Y-Z-only'),
    tokID1 = tokenWOAnyColor('pCombined',1, {'A','B','C'});
elseif strcmpi(transition.name, 'tNOT_ABC'),
    tokID1 = tokenWOEXColor('pABC',1, {'A','B','C'});
elseif strcmpi(transition.name, 'tNOT_XYZ'),
    tokID1 = tokenWOEXColor('pXYZ',1, {'X','Y','Z'});
end;
transition.selected_tokens = tokID1;
fire = tokID1;
```

29. Wrapping Up: Token Selection based on Color

29.1 Example-29-1: 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’}}.



Figur 29-1: Simple PN for color demo

- Transition **tA** selects token with color ‘A’ from **pCombined** (meaning tokens with color {‘A’}or {‘A’, ‘B’} or {‘A’, ‘X’} are relevant):
  Program code in pre-processor:
```
selected_tokens = tokenAnyColor('pAB', 1, {'A'})
fire = (selected_tokens);  % must
```

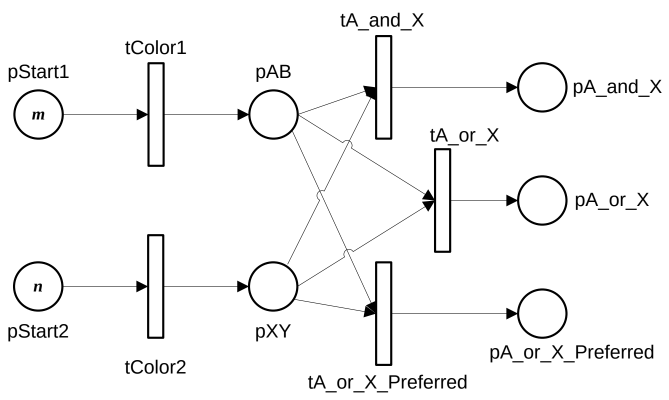
- Transition **tA_or_B** selects ‘A’ **or** ‘B’ from **pCombined**:
  Program code in pre-processor:
```
tokID1 = tokenAnyColor('pAB', 1, {'A','B'});
selected_tokens = tokID1;    % tokens to be removed
fire = tokID1; % must
```

- Transition **tA_or_B_Preferred** **prefers** ‘A’ or ‘B’ from **pCombined**:
  Program code in pre-processor:
```
tokID1 = tokenAnyColor('pAB', 1, {'A','B'});
selected_tokens = tokID1; % preferred tokens to be consumed
fire = 1; % fire anyway
```

- Transition **tA_and_B** selects a token with ‘A’ **and** ‘B’ from **pCombined**:
  Program code in pre-processor:
```
tokID1 = tokenAllColor('pAB', 1, {'A','B’});
selected_tokens = tokID1;    % tokens to be removed
fire = tokID1; %must
```

29.2 Example-29-2: 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’}}.



Figur 29-2: PN with two input places for color demo

- Transition **tA_and_X** selects ‘A’ from **pAB** **and** ‘Y’ from **pXY**:
  Program code in pre-processor:
```
tokID1 = tokenAnyColor('pAB', 1, {'A'});
tokID2 = tokenAnyColor('pXY', 1, {'X'});
selected_tokens = [tokID1 tokID2];    % tokens to be removed
fire = all(selected_tokens); % must
```

- Transition **tA_or_X** select ‘A’ from **pAB** **or** ‘X’ from **pXY** (at least one token be ‘A’ or ‘X’):
  Program code in pre-processor:
```
tokID1 = tokenAnyColor ('pAB', 1, {'A'});
tokID2 = tokenAnyColor ('pXY', 1, {'X'});
selected_tokens = [tokID1 tokID2];    % tokens to be removed
fire = any(selected_tokens); % must
```

- Transition **tA_or_X_Preferred** **prefers** ‘A’ from **pAB** or ‘X’ from **pXY**:
  Program code in pre-processor:
```
tokID1 = tokenAnyColor('pAB', 1, {'A'});
tokID2 = tokenAnyColor('pXY', 1, {'X'});
selected_tokens = [tokID1 tokID2];    % tokens to be removed
fire = 1;
```

30. Token Selection based on Time

Table below shows the select functions that deals with arrival time of a token in a specific place:

Table 30-1: GPenSIM Functions for selection of tokens based on time

Function	*Description*
`tokenArrivedBetween`	Select tokens that arrived in a specific place between the time intervals
`tokenArrivedEarly`	Select tokens that arrived earliest to a specific place
`tokenArrivedLate`	Select tokens that arrived latest to a specific place

A transition may select input tokens based on time; tokens from the input places can be selected based on the time these tokens are deposited into the places. Selection can be done by three ways:

- Tokens that are deposited into the place earliest '**FCFS**' (**First-Come-First-Served**), [set_of_tokID, nr_token_av] = tokenArrivedEarly(place, nr_tokens_wanted)

- Tokens that are deposited into the place latest '**LCFS**' (**Last-Come-First-Served**), [set_of_tokID, nr_token_av] = tokenArrivedLate(place, nr_tokens_wanted)

- Tokens that are deposited into the place between a time interval (early_time – final_time). [set_of_tokID, nr_token_av] = tokenArrivedBetween(place, nr_tokens_wanted, et, ft)

The output parameters of the functions are a set of IDs of the selected tokens (set of tokID 'set_of_tokID') and the actual number of valid tokID in the set ('nr_token_av').

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

```
function [fire, transition] = tLR_A_pre (transition)

selected_tokens = tokenArrivedEarly('pBUFF',4);
fire = 1;
```

If **pBUFF** has more than or equal to 4 tokens, then the returned value **`selected_tokens`** will have tokIDs of the 4 oldest tokens. Otherwise (if **pBUFF** has less than 4 tokens), then **`selected_tokens`** will have tokIDs of all the tokens, padded with 0s.

E.g. if a transition wants **4** **youngest** tokens from the input place **pBUFF**, where pBUFF has only two tokens at theat time:

```
function [fire, transition] = tLR_A_pre (transition)

selected_tokens = tokenArrivedLate('pBUFF',4);
fire = 1;
```

Composition of **`selected_tokens`** will be [tokIDi tokIDj 0 0], where tokIDi and tokIDj are valid tokIDs.

30.1 Example-30-1: Token selection based on time

Figure-30-1 shows the example for token selection based on time. **pStart** has 100 initial tokens (initial tokens are, of course, colorless). **tColor** add colors to the tokens it deposits into **pQueue**. The branch "**pDelay – tDelay – pReady**" is a delay, just to keep **tSelect** wait until all the 100 tokens from **pStart** through **tColor** are deposited into **pQueue**.

tColor adds color to tokens followingly:

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

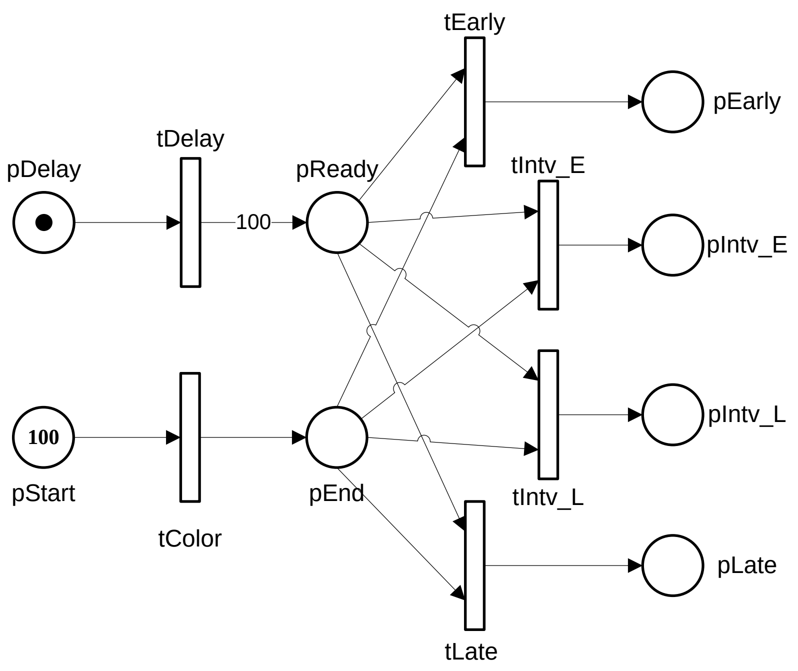


Figure 30-1: Token selection based on time

The transitions **tEarly**, **tIntv_E**, **tIntv_L**, and **tLate** are allowed to fire three (3) times only. These four transitions select input tokens from **pQueue** accordingly:

- **tEarly** selects only the earliest tokens in **pQueue**
- **tIntv_E** selects only the tokens that arrived at **pQueue** within the time interval [10 20]
- **tIntv_L** selects only the tokens that arrived at **pQueue** within the time interval [80 90]
- **tLate** selects only the tokens that arrived latest at **pQueue**

PDF:

```
% Example-30-1: Token selection based on time
function [png] = select_time_pdf()
png.PN_name = 'Token selection based on time';
png.set_of_Ps = {'pStart', 'pQueue', 'pDelay', 'pReady',...
                 'pEarly', 'pIntv_E', 'pIntv_L', 'pLate'};
png.set_of_Ts = {'tColor', 'tDelay', ...
                 'tEarly', 'tIntv_E', 'tIntv_L', 'tLate'};
png.set_of_As = {'pStart','tColor',1, 'tColor','pQueue',1,...  % tColor
    'pDelay','tDelay',1, 'tDelay','pReady',100,...      % tDelay 100
    'pQueue','tEarly',1, 'pReady', 'tEarly',1, ...      % tEarly
    'tEarly','pEarly',1, ...                            % tEarly
    'pQueue','tIntv_E',1, 'pReady', 'tIntv_E',1, ...    % tIntv_E
    'tIntv_E','pIntv_E',1, ...                          % tIntv_E
    'pQueue','tIntv_L',1, 'pReady', 'tIntv_L',1, ...    % tIntv_L
    'tIntv_L','pIntv_L',1, ...                          % tIntv_L
    'pQueue','tLate',1, 'pReady', 'tLate',1, ...        % tLate
    'tLate','pLate',1, ...                              % tLate
                };
```

MSF:

```
% Example-30-1: Token selection based on time
clear all; clc;

global global_info;
global_info.STOP_AT = 150;

png = pnstruct('select_time_pdf');

dyn.m0 = {'pStart',100, 'pDelay',1};
dyn.ft = {'tDelay',110, 'allothers',1};
pni = initialdynamics(png,dyn);
sim = gpensim(pni);

prnfinalcolors(sim, {'pEarly', 'pIntv_E', 'pIntv_L', 'pLate'});
```

COMMON_PRE:

```
function [fire, transition] = COMMON_PRE (transition)

if strcmpi(transition.name, 'tColor'),
    transition.new_color = num2str(current_time());
    fire = 1;
    return
end;

% the following transitions are allowed to fire only three times
if ismember(transition.name, {'tEarly', 'tIntv_E', 'tIntv_L', 'tLate'}),
    tx = get_trans(transition.name);
    if eq(tx.times_fired,3), % if the trans is already fired
        fire = 0;
        return;
    end;

    if strcmpi(transition.name, 'tEarly'),
        tokID1 = tokenArrivedEarly('pQueue', 1);
    elseif strcmpi(transition.name, 'tIntv_E'),
        tokID1 = tokenArrivedBetween('pQueue', 1, 10, 20);
    elseif strcmpi(transition.name, 'tIntv_L'),
        tokID1 = tokenArrivedBetween('pQueue', 1, 80, 90);
    else %strcmpi(transition.name, 'tIntv_E'),
        tokID1 = tokenArrivedLate('pQueue', 1);
    end;
    transition.selected_tokens = tokID1;
end;

fire = 1;
```

Simulation results: We break the results into three parts:
The first part, given below, clearly indicates that **tEarly** selected the earliest arrived tokens from **pQueue**, as the colors attached to the tokens are ‘0’-‘2’, meaning these tokens were made by **tColor** at the times 0-2.

```
    'Time:'     '111'     ' Place:'     'pEarly'     ' Colors:'     '0'
    'Time:'     '112'     ' Place:'     'pEarly'     ' Colors:'     '1'
    'Time:'     '113'     ' Place:'     'pEarly'     ' Colors:'     '2'
```

The second part, given below, also shows that **tLate** selected the latest arrived tokens from **pQueue**, as the colors attached to the tokens are ‘97’-‘99’, meaning these tokens were made by **tColor** at the times 97-99.

```
    'Time:'    '111'    ' Place:'    'pLate'    ' Colors:'    '99'
    'Time:'    '112'    ' Place:'    'pLate'    ' Colors:'    '98'
    'Time:'    '113'    ' Place:'    'pLate'    ' Colors:'    '97'
```

The third part is for **tIntv_E** which selects tokens deposited (arrived) at pQueue within the time interval [10 20]. However, from the printout below, one of the colors indicate that the tokens was made at time=9, which is quite OK. This is because,

- (current time = 9): the pre-processor of **tColor** (**tColor_pre**) makes the color label at *t = 9*
- **tColor** start firing which takes 1 TU to finish
- (current time = 10) **tColor** has completed firing and the token is deposited into **pQueue**. Eventhough the token was deposited at t=10 into **pQueue**, the color label on the token says “t=9” as the label was made at t=9.

```
    'Time:'    '111'    ' Place:'    'pIntv_E'    ' Colors:'    '9'
    'Time:'    '112'    ' Place:'    'pIntv_E'    ' Colors:'    '10'
    'Time:'    '113'    ' Place:'    'pIntv_E'    ' Colors:'    '11'
```

Part-III: Resources

31. Resources

In engineering systems, there are always resources; e.g. resources like humans or robots are needed to operate some machines; printers are common resources in a computer LAN network. Resources are usally represented by places, with number of initial token equaling to the available instances of the resource (transitions can also be used to represent reosurces, depending on the situation).

In GPenSIM, resources can be handled differently, so that the Petri net model becomes much simplified. In GPenSIM, resources can be treated as variables. GPenSIM also provides is a print function called '**prnschedule**' to print the usage of the resources.

There are different types of resource access, as resources can be:

- Generic: this means, a resource has many indistinguiable instantces. E.g. in a bank, there are 3 cashiers, all of them process all kind of transactions
- Specific: this means, there are a number of resources available, and each of these resource is unique. E.g. in a car worksted, 'Al' is a specialist in fixing carburetor, 'Bob' is a specialist in reparing transmission, and 'Chuck' is a painter.
- Write access: a resource may contain many instances, and a transition may try to acquire one or many of these intances. Write acces means, the resource will be locked and all the instances will be made available to a requesting (single) transition.

Table 31-1: GPenSIM Functions for resource

Function	*Description*
`request`	request resource or resource instances
`requestWR`	request resource with write access (exclusive access): this means, if the resources has many instances, either all the instances are granted (if available), or none is granted.
`release`	Release all the resources and resource instances held by a transition.

31.1 Declaring Resources

The resources are to be declared first in the MSF. For example, if a LAN network has 5 identical printer resources (called 'instances') then they are added to the dynamic part; also assume that each printer instance can be used as long as (as much as) wanted. In MSF:

```
dynamicpart.re = {'printer', 5, inf};
```

Declaring three mechanics (one instance each) named 'Al', 'Bob', and 'Chuck'; assume that each mechanics can also work up to 10, 20, and 30 hours, respectively in a day (also known as **cycle time** in scheduling algorithms):

```
dynamicpart.re = {'Al',1,10, 'Bob',1,20, 'Chuck',1,30};
```

31.2 Requesting resources

A transition can only request resource(s) in either in its specific pre-processor file or in the COMMON_PRE file. Requesting a resource can be done through the function '**request**'. For example:

```
[granted] = request('T1'); %T1 seeks any one resource
```

Here, transition '**T1**' requests one instance of a resource (any available resource). If allocation was successful, the flag 'granted' will be true.

Requesting two printers can be done through the function ‘**request**’ in the following way:

```
[granted1] = request('T1'); % T1 seek one printer
[granted2] = request('T1'); % T1 seek one more printer

% T1 can fire only if it has secure two printers
%     meaning, both granted1 and granted2 reutrned true values
fire = and(granted1, granted2);
```

Alternatively, requesting two printers can be done by the following way too:

```
[granted1] = request('T1', {'Printer', 2}); % T1 seek 2 printer instances
% T1 can fire if both printers are granted
fire = granted1;
```

Requesting specific resources, let’s say 2 intances of the resource type ‘Al’ and 1 instance of the resource type ‘Bob’:

```
% T1 seeks specific resources: 2 instances of ’Al’ and 1 ’Bob’
[granted] = request('T1', {'Al',2, 'Bob',1});
```

requestWR: requesting all the instances of a resource
Let’s say that there are altogether 5 intances of a machine park. If we want to perform maintenance of the machine park, we have to wait until all the instances of the machine park is free and available. Then, we will request all the instances so that we can perform maintenance on all of them.

```
% T1 seeks all the instances of a specific resource
[granted] = requestWR('T1', 'MC/Park');
```

Note: A transition may request resources in its speific pre-processor or COMMON_PRE file. The request may be granted by the underlying system. However, only if the transition starts firing, these (granted) resources will be allocated to the transition. On the other hand, if the transition is not allowed to fire (fire=0 in the pre-processor files), then the granted resources will be taken away from the transition.

31.3 Releasing the Resources (after usage)

After firing, a transition has to release ***all the resources*** it is holding; after firing, **releasing some resources (not all) is not possible**. Releasing the resources is usually done in the post-processor files (specific post or COMMON_POST).

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

31.3.1 Function ‘prnschedule’

Function ‘**prnschedule**’ is exclusively used when resources are involved. After simulations, this function prints how much (how long) each of the resources and their instances are used, and by whom (which transitions).

31.4 Example-31-1: Using Resources to realize critical section

This example is the again alternating firing of two transitions. Previously, in example-09-01, we achieved alternating firing with the help of binary semaphore. In example-17-01, we achieved alternating firing by manipulating priorities of the two transitions. This time, we make use of ‘resources’.

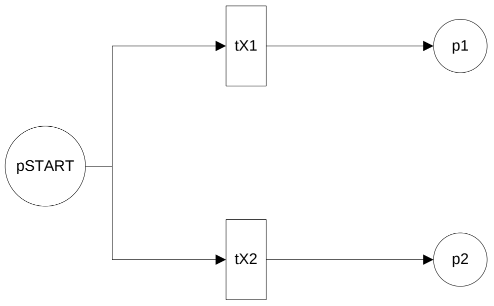


Figure 31-1: Alternative firing, achieved by using resouces

In the figure-31-1 shown below, transitions **tX1** and **tX2** are supposed to fire alternatively. To allow the transitions to fire alternatively, these two transitions seek a common *resource*; the common resource behaves like a criticial region that allows only one transition to use it at a time. If a transition does not get the resource, its priority is increased so that next time it will get it.

PDF:

```
% Example-30-1: Resource as Critical Region
function [png] = critical_region_pdf()

png.PN_name = 'Example-30-1: Resource as Critical Region';
png.set_of_Ps = {'pSTART', 'p1', 'p2'};
png.set_of_Ts = {'tX1','tX2'};
png.set_of_As = {'pSTART','tX1',1, 'tX1','p1',1,...
             'pSTART','tX2',1, 'tX2','p2',1};
```

MSF:

```
% Example-30-1: Resource as Critical Region
global global_info;
global_info.DELTA_TIME = 0.5; %
global_info.STOP_AT  = 50; %

pns = pnstruct('critical_region_pdf');

dyn.m0 = {'pSTART',10};
dyn.ft = {'tX1',1, 'tX2',5};
dyn.re = {'Resource-X',1,inf}; % resource as semafor
dyn.ip = {'tX1', 1}; % let tX1 fire first
pni = initialdynamics(pns, dyn);

sim = gpensim(pni);

plotp(sim, {'p1', 'p2'});

prnschedule(sim);
occupancy(sim, {'tX1', 'tX2'});
```

COMMON_PRE:

In the COMMON_PRE file, **tX1** will try to acquire the common resource (critical regoin); if the resource is being used by the other transition **tX2**, then priority of **tX1** will be increased and at the same time priority of **tX2** will be decreased, so that next time **tX1** will get it.

```
function [fire, transition] = COMMON_PRE(transition)

granted = request(transition.name, {'Resource-X',1});

if not(granted), % if not suceeded, increase priority so next time it will
    if strcmp(transition.name, 'tX1'),
        priorset('tX1', 1); %
        priorset('tX2', 0); %
    else
        priorset('tX1', 0); %
        priorset('tX2', 1); %
    end;
end;

fire = granted; % fire only if resource acquire is sucessful
```

COMMON_POST:
As usual, the reources used by the fired transition will be released.

```
function [] = COMMON_POST(transition)

% release all resources used by transition
release(transition.name); %
```

Results:
The results show that the two transitions fire alternatively.

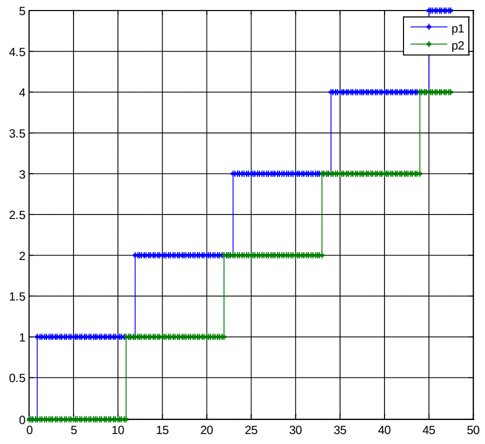


Figure 31-2: Simulation results showing alterive firing

Simulation results:
Screen dump of the functions prnschedule and occupancy are given below:

```
RESOURCE USAGE:
RESOURCE INSTANCES OF ***** Resource-X *****
tX1 [0 : 1]
tX2 [1 : 6]
tX1 [6 : 7]
tX2 [7 : 12]
tX1 [12 : 13]
tX2 [13 : 18]
tX1 [18 : 19]
```

```
tX2 [19 : 24]
tX1 [24 : 25]
tX2 [25 : 30]
Resource Instance: Resource-X:: Used 10 times. Utilization time: 30

RESOURCE USAGE SUMMARY:
Resource-X:  Total occasions: 10   Total Time spent: 30

*****  LINE EFFICIENCY AND COST CALCULATIONS: *****
  Number of servers:  k = 1
  Total number of server instances:  K = 1
  Completion = 50
  LT = 50
  Total time at Stations: 30
  LE = 60 %
  **
  Sum resource usage costs: 0   (NaN% of total)
  Sum firing costs: 0   (NaN% of total)
  Total costs:  0
  **
```

```
Occupancy analysis ....
Simulation Completion Time: 50
occupancy tX1:
  total time: 5
  Percentage time: 10%
occupancy tX2:
  total time: 25
  Percentage time: 50%
```

32. Example-32-1: Using Specific Resources

This example is important and interesting in the sense it shows why GPenSIM handles resources differently; this example shows that bu using resources, the size of the Petri net model can be minimized. This example is adapted from Hruz and Zhou (2007), page 185.

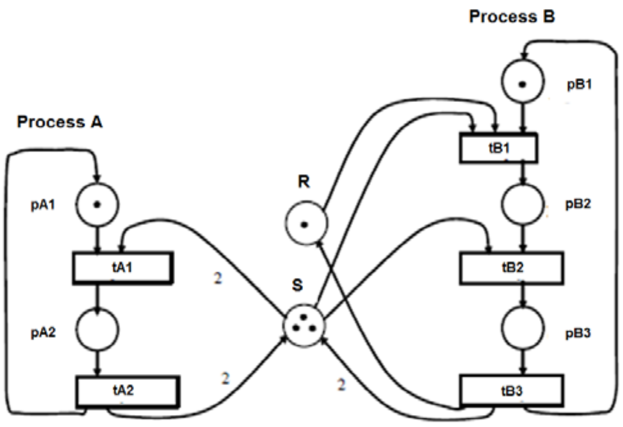


Figure 32-1: Two processes (A and B) with two resources (R and S) [adapted from Hruz and Zhou, 2007]

Figure given above shows a system with two processes (**A** and **B**) that share two different kinds of resources (**R** and **S**). There are one instance of resource R and three instances of resource S available. The process A (manufacturing of a product of type A) needs two instances of the resource S to be assigned in a certain time. The process B also needs two instances of S but during different stages, in addition the robot R for its finishing. Both processes are running cyclically.

The standard P/T Petri net model shown in figure-32-1 reveals that the resources are represented by tokens inside places **R** and **S**; this means, there must be arcs connecting transitions (like **tA1**, **tB2**, etc.) to the places **R** and **S**. Shown below in figure-32-2 is the GPenSIM version of the same system.

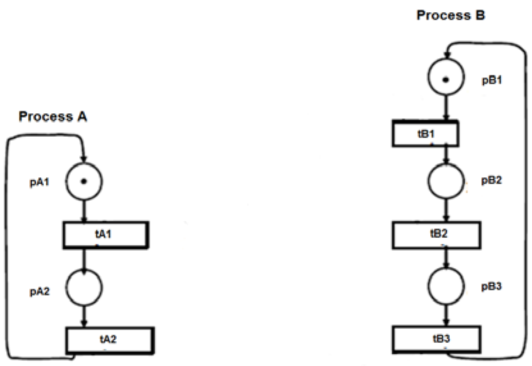


Figure 32-2: Simplified model, GPenSIM style

In the GPenSIM version, shown in figure 32-2, resources are treated like variables and will be not shown in the Petri net model; thus, there will be no arcs from transitions requesting resources to places that are supposed to keep the resources. Thus, GPenSIM version is much simpler.

PDF:
```
% Example-32-1: AOPN Model: System with 2 processes:
%                realized with GPenSIM resources
function [png] = sysAB_AOPN_pdf()
png.PN_name = 'SYS realized with GPenSIM resources ';
png.set_of_Ps = {'pA1','pA2', 'pB1','pB2','pB3'};
png.set_of_Ts = {'tA1','tA2', 'tB1','tB2','tB3'};
png.set_of_As = {'pA1','tA1',1, 'tA1','pA2',1, ...
                 'pA2','tA2',1, 'tA2','pA1',1, ...
                 'pB1','tB1',1, 'tB1','pB2',1, ...
                 'pB2','tB2',1, 'tB2','pB3',1, ...
                 'pB3','tB3',1, 'tB3','pB1',1};
```

COMMON_PRE:

```
function [fire, transition] = COMMON_PRE(transition)

if strcmp(transition.name, 'tA1'),
    granted = request(transition.name, {'S', 2});

elseif strcmp(transition.name, 'tB1'),
    granted = request(transition.name, {'R',1, 'S',1});

elseif strcmp(transition.name, 'tB2'),
    granted = request(transition.name, {'S', 1});
else
    granted = 1;
end;

fire = granted; % fire only if resource acquisition was sucessful
```

COMMON_POST:

IMPORTANT: After firing, **tA2** must release resources acquired by **tA1**; **tA2** should not try to release resource that acquired by itself, as there aren't any resources acquired by it. Similarly, after firing, **tB3** must release resources acquired by **tB1** and **tB2**; **tB3** should not try to release resource that acquired by itself, as there aren't any resources acquired by it.

```
function [] = COMMON_POST(transition)

if strcmp(transition.name, 'tA2'),
    release('tA1');

elseif strcmp(transition.name, 'tB3'),
    release('tB1');
    release('tB2');
end;
```

MSF:

```
% Example-32-1: AOPN Model of System with 2 processes:
%                realized with GPenSIM resources
global global_info;
global_info.STOP_AT = 20;

pns = pnstruct('sysAB_AOPN_pdf');

dynamicpart.m0 = {'pA1',1, 'pB1',1};
dynamicpart.ft = {'allothers',1};
dynamicpart.re = {'R',1,inf, 'S',3,inf};
pni = initialdynamics(pns, dynamicpart);

sim = gpensim(pni);
prnschedule(sim);
occupancy(sim, {'tA1', 'tB1'});
```

Simulation results:

```
RESOURCE USAGE:
RESOURCE INSTANCES OF ***** R *****
tB1 [0 : 4]
tB1 [4 : 10]
tB1 [10 : 18]
Resource Instance: R:: Used 3 times. Utilization time: 18

RESOURCE INSTANCES OF ***** S *****
(S-1):   tA1 [0 : 2]
(S-2):   tA1 [0 : 2]
(S-3):   tB1 [0 : 4]
(S-1):   tB2 [2 : 4]
(S-2):   tA1 [4 : 6]
(S-3):   tA1 [4 : 6]
(S-2):   tA1 [6 : 8]
(S-3):   tA1 [6 : 8]
(S-1):   tB1 [4 : 10]
(S-2):   tB2 [8 : 10]
(S-1):   tA1 [10 : 12]
(S-2):   tA1 [10 : 12]
(S-1):   tA1 [12 : 14]
(S-2):   tA1 [12 : 14]
(S-1):   tA1 [14 : 16]
(S-2):   tA1 [14 : 16]
(S-3):   tB1 [10 : 18]
(S-1):   tB2 [16 : 18]
Resource Instance: (S-1):: Used 7 times.  Utilization time: 18
Resource Instance: (S-2):: Used 7 times.  Utilization time: 14
Resource Instance: (S-3):: Used 4 times.  Utilization time: 16

RESOURCE USAGE SUMMARY:
R:  Total occasions: 3   Total Time spent: 18
S:  Total occasions: 18   Total Time spent: 48
```

```
*****  LINE EFFICIENCY AND COST CALCULATIONS: *****
  Number of servers:  k = 2
  Total number of server instances:  K = 4
  Completion = 20
  LT = 80
  Total time at Stations: 66
  LE = 82.5 %
  **
  Sum resource usage costs: 0   (NaN% of total)
  Sum firing costs: 0   (NaN% of total)
  Total costs:  0
  **
```

33. Scheduling using Resources

Scheduling is basically resource management thus scheduling can be efficiently modeled and simulated by Petri Nets. Because of the support for resources provided by GPenSIM, modeling, simulation, and performance analysis of scheduling can be done more elegantly.

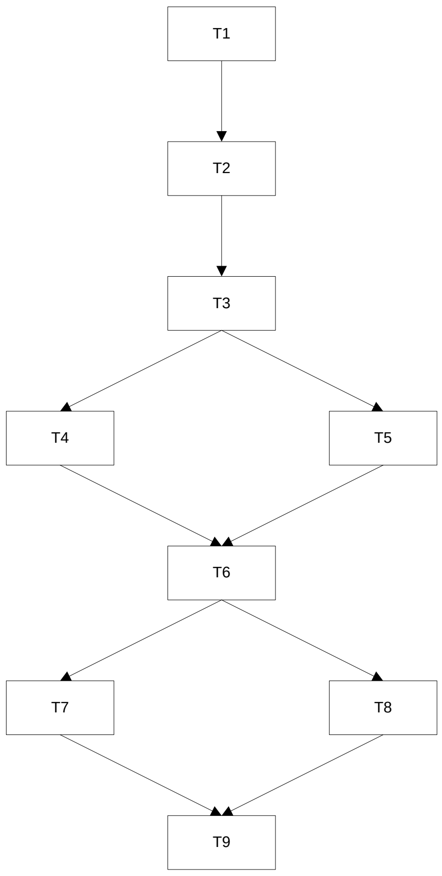


Figure 33-1: Scheduling of software developement project

33.1 Example-33-01: Scheduling of Software Development Project

A team of software developers are about to start a software development project. As the delivery time is fixed and critical, they want to evaluate the possibilities of completing the project within the time frame and whether it will cost them additional staff.

The ***digraph*** shown as figure-33-1 reveals how the different stages of the project are related to each other; stage-9 (**T9**) is a 'dummy' stage, just to indicate completion of the whole project. The table given below shows the stages in the software development, probable time (in month) each stage will take and the resources demanded by each stage (task).

Table 33-1: The stages of the project

	Stage (or task)	Time (months)	Resources (developers)
T1.	Requirements analysis	3	**2**
T2.	Planning, Research, Technology Selection	3	**2**
T3.	Modeling, software design and prototyping	3	**3**
T4.	Coding and integration	6	**5**
T5.	Code and Product Documentation	6	**1**
T6.	Testing (Validation) and Optimization	2	**3**
T7.	Launch, Deployment (or Installation)	1	**2**
T8.	Maintenance	1	**2**

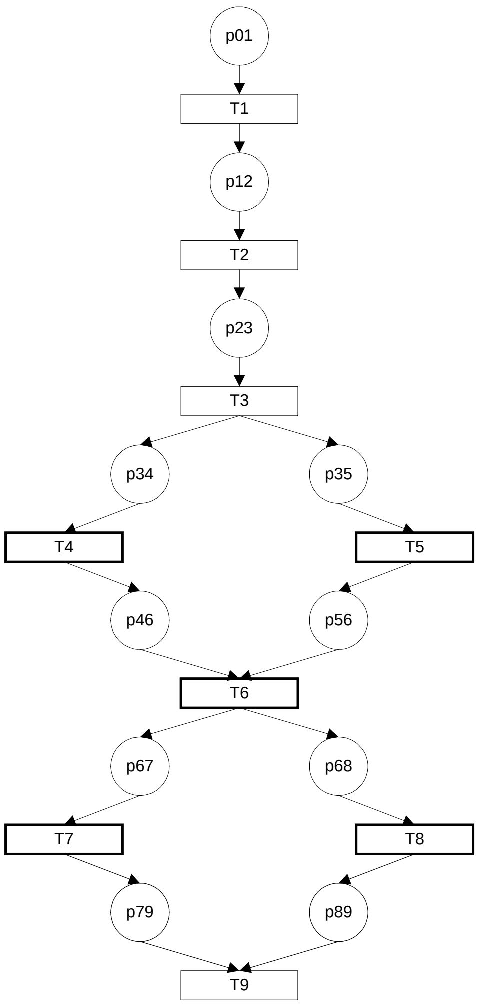


Figure 33-2: Partial Petri Net model of the SW developement project

Petri net model: Petri net model can easily built from the digraph shown in figure-33-1. First of all, we replace each task (square box) in the digraph by a transition. And then, between any two transitions, we inject a place as a buffer. In addition, just to start the system, we put a place with an initial token at the top above **T1**. Figure-33-2 shows the ***partial*** Petri net model.

To make a complete Petri net model, we have to add the resource restriction to each transition. For example, for Stage-1 (task **T1**) to start, there must be at least 2 developers (resources) available. Figure-33-3 below shows the complete Petri net model.

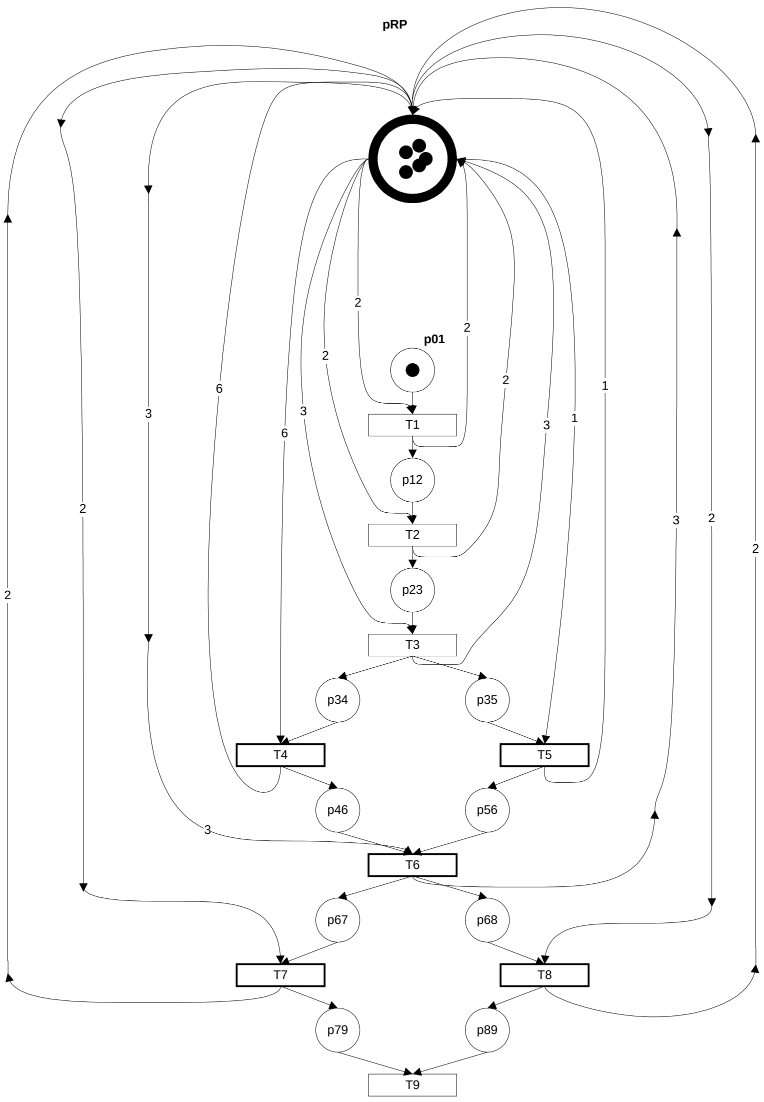


Figure 33-3: (Complete) Petri net model for software development project

The Petri net model shown in figure-33-3 is unnecessarily complex and it is rigid; the model is complex because of the resource restrictions imposed by the arcs from place **pRP** to all the tasks. The model is also rigid as it does not differentiate between the resources: for example, there is no distinction between a project manager, a senior developer, and a trainee developer, as they all are assumed as just homogenous tokens in **pRP**. One way to differentiate the tokens is to color each token, but in this section, we will use resources.

Let us eliminate resources from the remove Petri Net model: let us remove the place **pRP** (which represent he worker resources) from the model, and push it into the background as resource variable. Eliminating **pRP** from the model also removes all the arcs from this place to all other transitions **T1-T9**. Thus, the newer slimmer Petri Net model has become one that is already shown in the figure-33-2 as the partial Petri net model. However, this model shown in figure-33-2 is no longer partial, as it the full model that will use resources as variables.

PDF: PDF is much simpler due to the elimination pRP and its arcs to all the transitions (figure-33-2).

```
% Example-33-1: Scheduling with Resources
function [png] = schedule_w_resources_pdf()
png.PN_name   = 'ex-33-3: schedule_w_resources_pdf';
png.set_of_Ps = {'p01','p12','p23','p34','p35','p46','p56',...
                 'p67','p68','p79','p89','p90'};
```

```
png.set_of_Ts = {'T1','T2','T3','T4','T5','T6','T7','T8','T9'};
png.set_of_As = {...
    'p01','T1',1, 'T1','p12',1,...                   % T1
    'p12','T2',1, 'T2','p23',1, ...                  % T2
    'p23','T3',1, 'T3','p34',1, 'T3','p35',1, ...    % T3
    'p34','T4',1, 'T4','p46',1, ...                  % T4
    'p35','T5',1, 'T5','p56',1, ...                  % T5
    'p46','T6',1, 'p56','T6',1, 'T6','p67',1, 'T6','p68',1, ... % T6
    'p67','T7',1, 'T7','p79',1, ...                  % T7
    'p68','T8',1, 'T8','p89',1, ...                  % T8
    'p79','T9',1, 'p89','T9',1, 'T9','p90',1, ...    % T9
                };
```

MSF: In the MSF, the resources are defined as a part of the dynamic details.

```
% Example-33-1: Scheduling with rescources

clear all; clc;
global_info.MAX_LOOP = 30;

pns = pnstruct('schedule_w_resources_pdf');

dp.m0 = {'p01', 1};
dp.ft = {'T4',6,'T5',6, 'T6',2, 'T7',1,'T8',1, 'T9',0.1, 'allothers',3};
dp.re = {'developers',9,inf};
pni = initialdynamics(pns, dp);

sim = gpensim(pni);
prnschedule(sim);
```

COMMON_PRE: all the transitions request resources through COMMON_PRE.

```
function [fire, trans] = COMMON_PRE(trans)

global global_info;

if ismember(trans.name, {'T1', 'T2', 'T7', 'T8'}),
    % request 2 instances of resource
    granted = request(trans.name, {'developers', 2});
elseif ismember(trans.name, {'T3', 'T6'}),
    % request 3 instances of resource
    granted = request(trans.name, {'developers', 3});
elseif strcmpi(trans.name, 'T4'),
    % request 6 instances of resource
    granted = request(trans.name, {'developers', 6});
elseif strcmpi(trans.name, 'T5'),
    % request 1 instance of resource
    granted = request(trans.name);
else  % 'T9'
    global_info.STOP_SIMULATION = 1; % stop simulations
    granted = 1;
end;
fire = granted;
```

COMMON_POST: After firing, transitions release the resources through COMMON_POST.

```
function [] = COMMON_POST(transition)
release(transition.name);
```

Simulation results:

```
RESOURCE USAGE:

RESOURCE INSTANCES OF ***** developers *****
(developers-1):   T1 [0 : 3]
(developers-2):   T1 [0 : 3]
(developers-1):   T2 [3 : 6]
(developers-2):   T2 [3 : 6]
(developers-1):   T3 [6 : 9]
(developers-2):   T3 [6 : 9]
(developers-3):   T3 [6 : 9]
(developers-7):   T5 [9 : 15]
(developers-1):   T4 [9 : 15]
(developers-2):   T4 [9 : 15]
(developers-3):   T4 [9 : 15]
(developers-4):   T4 [9 : 15]
(developers-5):   T4 [9 : 15]
(developers-6):   T4 [9 : 15]
(developers-1):   T6 [15 : 17]
(developers-2):   T6 [15 : 17]
(developers-3):   T6 [15 : 17]
(developers-3):   T8 [17 : 18]
(developers-4):   T8 [17 : 18]
(developers-1):   T7 [17 : 18]
(developers-2):   T7 [17 : 18]
Resource Instance: (developers-1):: Used 6 times.  Utilization time: 18
Resource Instance: (developers-2):: Used 6 times.  Utilization time: 18
Resource Instance: (developers-3):: Used 4 times.  Utilization time: 12
Resource Instance: (developers-4):: Used 2 times.  Utilization time: 7
Resource Instance: (developers-5):: Used 1 times.  Utilization time: 6
Resource Instance: (developers-6):: Used 1 times.  Utilization time: 6
Resource Instance: (developers-7):: Used 1 times.  Utilization time: 6
Resource Instance: (developers-8):: Used 0 times.  Utilization time: 0
Resource Instance: (developers-9):: Used 0 times.  Utilization time: 0

RESOURCE USAGE SUMMARY:
developers:  Total occasions: 21   Total Time spent: 73

*****  LINE EFFICIENCY AND COST CALCULATIONS: *****
  Number of servers:  k = 1
  Total number of server instances:  K = 9
  Completion = 18
  LT = 162
  Total time at Stations: 73
  LE = 45.0617 %
  **
  Sum resource usage costs: 0   (NaN% of total)
  Sum firing costs: 0   (NaN% of total)
  Total costs:  0
  **
```

The results show that the resources (work force) is not utilized fully; there are too many developers, as developer-8 and developer-9 are not utilized at all. In additon, developers-5, 6, and 7 are used only in one stage, and developer-4 only in two stages. Thus, resource (work force) utilization is only 45%.

33.2 Example-33-02: Scheduling with Specific Resources

Table-33-1 is rudimentary as it does not reveal the specific types of resources (like project manager, chief developer, etc.) needed for each stage. We will make use of table-33-2 instead, as table-33-2 is more advanced as it shows the specific resource requirements for each stage.

Let us assume that the the project team consists of 9 members: a project manager, a chief developer, 2 senior developers, 3 developers and 2 trainees. The table-33-2 given below shows specific resources needed by each stage, otherwise same as table-33-1.

Table 33-2: The specific resources needed in each stage

	<u>**Stage (or task)**</u>	<u>**Time (months)**</u>	<u>**Resources Needed**</u>
T1.	Requirements analysis	3	**project manager** and **chief developer**
T2.	Planning, Research, Technology Selection	3	**project manager** and **chief developer**
T3.	Modeling, software design and prototyping	3	**project manager**, **chief developer**, and 1 **senior developers**
T4.	Coding and integration	6	Any 6 member
T5.	Code and Product Documentation	6	Any 1 member
T6.	Testing (Validation) and Optimization	2	Any 3 member
T7.	Launch, Deployment (or Installation)	1	**chief developer +** any 1 member
T8.	Maintenance	1	**chief developer +** any 1 member

Note that the stage-7 and 8 both demand the same resource 'chief developer'; this means, these two stages cannot be executed in parallel, eventhough the digraph permits parallel execution of these two stages.

<u>**PDF**</u>: (same as before, as resource is not part of the static Petri Net structure)

<u>**MSF**</u>: almost same as before, except the resource declaration:

```
dp.re = {'project manager',1,inf, 'chief developer',1,inf, ...
         'senior developers',2,inf, 'developers',3,inf, 'trainees',2,inf};
```

COMMON_PRE: there are changes as each transition requests specific resources:

```
function [fire, trans] = COMMON_PRE(trans)
global global_info;

if ismember(trans.name, {'T1', 'T2'}),
    granted = request(trans.name,...
        {'project manager',1, 'chief developer',1});

elseif strcmpi(trans.name, {'T3'}),
    granted = request(trans.name, ...
        {'project manager',1, 'chief developer',1, 'senior developers',1});

elseif strcmpi(trans.name, {'T4'}),
    % T4 seeks any 6 resources instances
    granted_res = 0;
    for i = 1:6,
        if request(trans.name), granted_res = granted_res + 1; end;
    end;
    granted = eq(granted_res, 6);

elseif strcmpi(trans.name, {'T5'}),
    % T5 seeks any 1 resources instance
    granted = request(trans.name);

elseif strcmpi(trans.name, {'T6'}),
    % T6 seeks any 3 resources instances
    granted_res = 0;
    for i = 1:3,
        if request(trans.name), granted_res = granted_res + 1; end;
    end;
    granted = eq(granted_res, 3);

elseif ismember(trans.name, {'T7', 'T8'}),
    % T7 and T8 seek 'chief developer' and any 1 resources instances
    granted1 = request(trans.name, {'chief developer',1});
    granted2 = request(trans.name);
    granted = and(granted1, granted2);

else
    % 'T9'
    global_info.STOP_SIMULATION = 1; % stop simulations
    granted = 1;
end;

fire = granted;
```

COMMON_POST: same as before

<u>Simulation results</u>:

```
RESOURCE USAGE:

RESOURCE INSTANCES OF ***** project manager *****
T1 [0 : 3.025]
T2 [3.025 : 6.025]
T3 [6.025 : 9.025]
T5 [9.025 : 15.025]
T6 [15.025 : 17.05]
T7 [17.05 : 18.075]
T8 [18.075 : 19.1]
Resource Instance: project manager:: Used 7 times. Utilization time: 19

RESOURCE INSTANCES OF ***** chief developer *****
T1 [0 : 3.025]
T2 [3.025 : 6.025]
T3 [6.025 : 9.025]
T4 [9.025 : 15.025]
T6 [15.025 : 17.05]
T7 [17.05 : 18.075]
T8 [18.075 : 19.1]
Resource Instance: chief developer:: Used 7 times. Utilization time: 19

RESOURCE INSTANCES OF ***** senior developers *****
(senior developers-1):   T3 [6.025 : 9.025]
(senior developers-1):   T4 [9.025 : 15.025]
(senior developers-2):   T4 [9.025 : 15.025]
```

```
(senior developers-1):   T6 [15.025 : 17.05]
Resource Instance: (senior developers-1):: Used 3 times.  Utilization time:
11
Resource Instance: (senior developers-2):: Used 1 times.  Utilization time:
6

RESOURCE INSTANCES OF ***** developers *****
(developers-1):   T4 [9.025 : 15.025]
(developers-2):   T4 [9.025 : 15.025]
(developers-3):   T4 [9.025 : 15.025]
Resource Instance: (developers-1):: Used 1 times.  Utilization time: 6
Resource Instance: (developers-2):: Used 1 times.  Utilization time: 6
Resource Instance: (developers-3):: Used 1 times.  Utilization time: 6

RESOURCE INSTANCES OF ***** trainees *****
Resource Instance: (trainees-1):: Used 0 times.  Utilization time: 0
Resource Instance: (trainees-2):: Used 0 times.  Utilization time: 0

RESOURCE USAGE SUMMARY:
project manager:  Total occasions: 7   Total Time spent: 19.1
chief developer:  Total occasions: 7   Total Time spent: 19.1
senior developers:  Total occasions: 4   Total Time spent: 17.025
developers:  Total occasions: 3   Total Time spent: 18
trainees:  Total occasions: 0   Total Time spent: 0

*****  LINE EFFICIENCY AND COST CALCULATIONS: *****
  Number of servers:  k = 5
  Total number of server instances:  K = 9
  Completion = 19.125
  LT = 172.125
  Total time at Stations: 73.225
  LE = 42.5418 %
  **
  Sum resource usage costs: 0   (NaN% of total)
  Sum firing costs: 0   (NaN% of total)
  Total costs:  0
  **
```

Note: Due to the conflict in stages-7 and 8 (both require ‘chief developer’), these two stages run one after the other. Because of this, completion time is delayed by one month (from 18 to 19).

34. Estimating Project Completion Time

This is just an application of Petri Nets with GPenSIM resources, for estimating project completion time.

34.1 Example-34-1: Project Completion Time

The following example is taken from **James D: Stein, "How Math Explains the World", page-3.** Figure-34-1 is a digraph showing the order of the tasks (and the time taken by the tasks) to be done to complete a work.

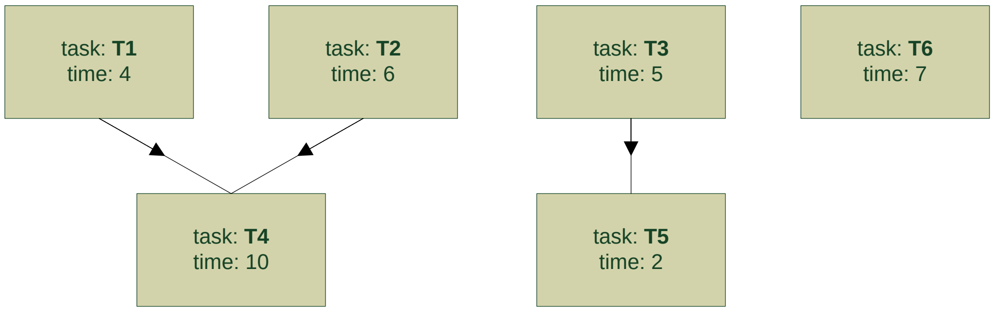


Figure 34-1: Digraph showing order of tasks to be completed

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 order of priority (high to low) is assumed to be T1, T2, … , and T6. Each task demands a human resource. There are two human resources (generic – capable of doing any task) are available.

Petri net model:
Petri Net model (figure-34-2) is easily obtained from the digraph, by substituting transitions for the tasks, and injecting places between the transitions.

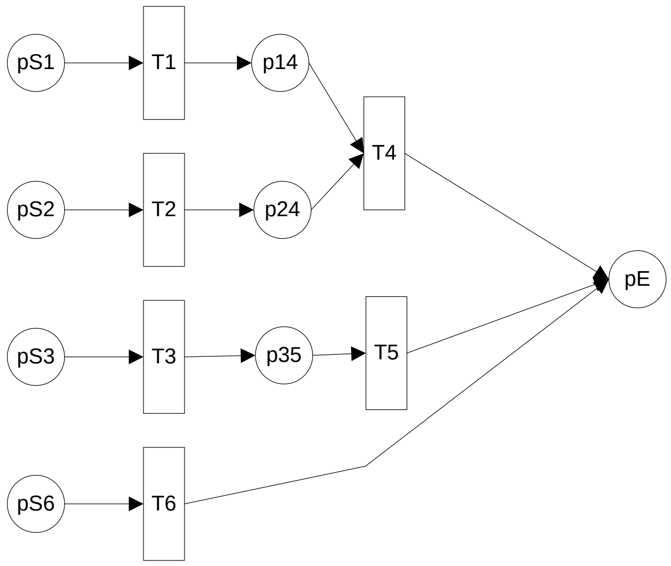


Figure 34-2: Petri Net model

PDF:

```
% Example-34-1: Project Completion Time
function [png] = project_completion_pdf()

png.PN_name  = 'project_completion_pdf';
png.set_of_Ps = {'pS1','pS2','pS3','pS6', 'p14','p24', 'p35', 'pE'};
png.set_of_Ts = {'T1', 'T2', 'T3', 'T4', 'T5', 'T6'};
png.set_of_As = {...
    'pS1','T1',1, 'T1','p14',1, ...  % T1
    'pS2','T2',1, 'T2','p24',1, ...  % T2
    'pS3','T3',1, 'T3','p35',1, ...  % T3
    'p14','T4',1, 'p24','T4',1, 'T4','pE',1, ... % T4
    'p35','T5',1, 'T5','pE',1, ...   % T5
    'pS6','T6',1, 'T6','pE',1};      % T6
```

MSF:

```
% Example-34-1: Project Completion Time
% Problem taken from James D: Stein, "How Math Explains the World", page.3
clear all; clc;

global global_info;
global_info.STOP_AT = 30;

pns = pnstruct('project_completion_pdf');

dp.m0 = {'pS1',1, 'pS2',1, 'pS3',1, 'pS6',1};
dp.ft = {'T1',4, 'T2',6, 'T3',5, 'T4',10, 'T5',2, 'T6',7};
dp.ip = {'T1',6, 'T2',5, 'T3',4, 'T4',3, 'T5',2, 'T6',1};
dp.re = {'Al',1,inf, 'Bob',1,inf};
pni = initialdynamics(pns, dp);

sim = gpensim(pni);

prnschedule(sim);
```

COMMON_PRE: each task needs one resource

```
function [fire, transition] = COMMON_PRE(transition)
fire = request(transition.name); % request any one instance of resource
```

COMMON_POST: in addition, to releasing resources after firing, there is one more important thing to do: **STOP the simulations, if the transitions T4, T5, and T6 have fired**.

```
function [] = COMMON_POST(transition)

global global_info;

release(transition.name);

% check if the transitions T4, T5, and T6 has fired.
% if so, stop the simulations, immediately
if ismember(transition.name, {'T4', 'T5', 'T6'}),
    t4 = get_trans('T4');
    t5 = get_trans('T5');
    t6 = get_trans('T6');
    t4_has_fired = t4.times_fired;
    t5_has_fired = t5.times_fired;
    t6_has_fired = t6.times_fired;
    if and(t4_has_fired, and(t5_has_fired, t6_has_fired)),
        global_info.STOP_SIMULATION  = 1;
    end;
end;
```

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

```
RESOURCE USAGE:

RESOURCE INSTANCES OF ***** Al *****
T1 [0 : 4]
T3 [4 : 9]
T5 [9 : 11]
T6 [11 : 18]
Resource Instance: Al:: Used 4 times. Utilization time: 18

RESOURCE INSTANCES OF ***** Bob *****
T2 [0 : 6]
T4 [6 : 16]
Resource Instance: Bob:: Used 2 times. Utilization time: 16

RESOURCE USAGE SUMMARY:
Al:  Total occasions: 4   Total Time spent: 18
Bob:  Total occasions: 2   Total Time spent: 16

*****  LINE EFFICIENCY AND COST CALCULATIONS: *****
  Number of servers:  k = 2
  Total number of server instances:  K = 2
  Completion = 18
  LT = 36
  Total time at Stations: 34
  LE = 94.4444 %
  **
  Sum resource usage costs: 0   (NaN% of total)
  Sum firing costs: 0   (NaN% of total)
  Total costs:  0
  **
```

'Al' was working all the time (for all 18 time units), whereas 'Bob' has nothing to do during the time interval 16-18 time units. Thus, efficiency is (18+16)/(2*18) = 94.44%

Part-IV: Cost Analysis

35. Getting Started with Cost Analysis

No simulator for discrete event system is complete unless it provides some in-built support cost analysis. This is because, in engineering and in industry, we do study about performance of discrete event systems in order to save time and resources which ultimately results in saving money (costs).

In GPenSIM, as explained before, only transitions are active and only the actvities can cost money. Places are passive and their sole purpose is to keep tokens. In GPenSIM, places and their storage of tokens can not cost money.

When a transition fires:

- At the start of firing, the transition consume input tokens: the cost of the firing (***Cost_of_Firing***) at this stage is the summation of individual costs of all the input tokens; this means, every token carries the costs with it. Let us denote the summation of individual costs of all the input tokens as ***Total_Cost_of_Input_Tokenens***

  Again, at the start of firng: ***Cost_of_Firing = Total_Cost_of_Input_Tokenens***
- At the end of firirng, the cost of the firing becomes more as the firing cost of the transiton has to be added to the totl cost; however, the **firing cost** of the transition is two folded: 1) fixed cost **(*fc_fixed*)**, and variable cost (***fc_variable***). Hence, the total cost at the end of firing becomes:

  Cost_of_Firing = Total_Cost_of_Input_Tokenens + fc_fixed + (firing_time * fc_variable)

- When the firing completes, the input tokens are transformed into a number of output tokens that will be deposited into the output places. The cost of firing will be equally distributed to the output tokens. Thus, cost of each output token becomes:

  Cost_of_each_output_token = Cost_of_Firing / number_of_output_tokens (t^o)

In additon to transitions, reosurce usage will also also cost money. In this section, we shall study about how to compute costs induced by transitions without using resources.

35.1 Firing Costs of Transition

As explained above, a transition can have up to two firing costs:

1. **Fixed cost (*fc_fixed*)**: this is one time cost, as the transition fires, this cost will be added to the total firing cost.
2. **Variable cost (*fc_variable*)**: this is firing cost per firing for a unit of time. Thus, this cost will be multiplied with firing time before added to the total firing cost.

The firing costs are declared in the MSF, as a part of initial dynamics. For example,

```
% first costs analysis
…
…
pns = pnstruct('abc3_pdf');

dyn.m0 = {'p1',4, 'p2',3, 'p3',5};
dyn.ft = {'t1',2, 't2',5,'allothers',1};
dyn.fc_fixed = {'t1',50, 't2',100};
dyn.fc_variable = {'t1',20, 't1',25};
pni = initialdynamics(pns, dyn);

sim = gpensim(pni);
```

In the snippet shown above, the fixed firing cost of **t1** is NOK 50. The variable firing cost of **t1** is NOK 20 per unit of firing time. Whereas, the fixed firing cost of **t2** is NOK 100, and the variable firing cost of **t2** is NOK 20 (per unit of firing time).

35.2 Example-35-1: Firing Costs of Transition

Figure-35-1 shows a Petri Net with 4 transitions.

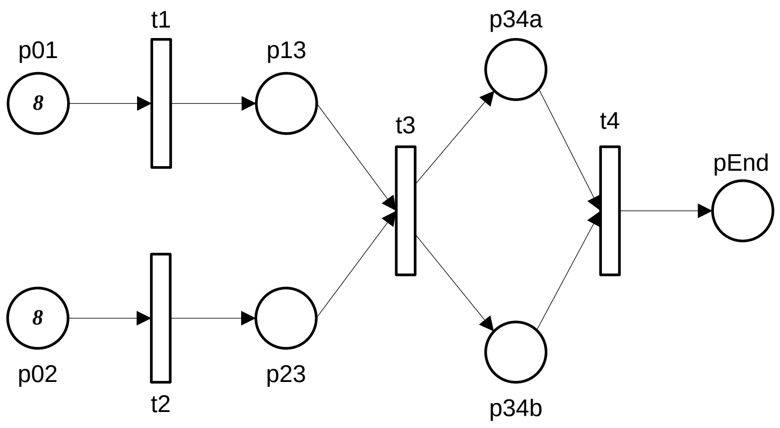


Figure 35-1: Petri Net fopr simple cost calculation

The firing times and the firing costs of the transitions are given below:

Table-35-1: Properties of the transitions

Transition	**Firing time (ft)**	**Firing Cost**	
		Fixed (*fc_fixed*)	**Varbiable (*fc_variable*)**
t1	10	3	5
t2	10	250	50
t3	10	5000	500
t4	10	10	10

In addition, transition **t1** and **t2** will be allowed to fire only 6 times (so that they will leave two tokens each in p01 and p02), **t3** will be allowed to fire only 4 times (so that it will leave two tokens each in p13 and p23), and **t4** will be allowed to fire only 2 times (so that it will leave two tokens each in p34a and p34b).

At the end of the simulation, there will be 2 tokens each as shown in figure-35-2:

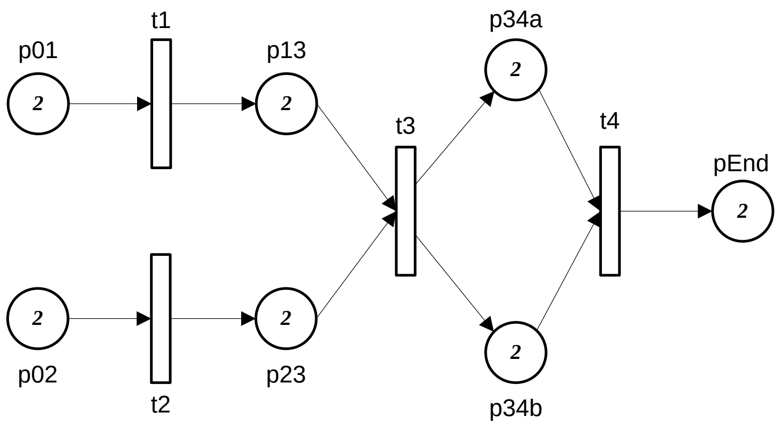


Figure 35-2: After simulation run, two tokens left in each place

Let us inspect the cost of each token:

```
Colors of Final Tokens:
  'Time:'  '0'   ' Place:'  'p01'  ' *** NO COLOR ***'  ' Cost: '  '0'
  'Time:'  '0'   ' Place:'  'p01'  ' *** NO COLOR ***'  ' Cost: '  '0'

  'Time:'  '0'   ' Place:'  'p02'  ' *** NO COLOR ***'  ' Cost: '  '0'
  'Time:'  '0'   ' Place:'  'p02'  ' *** NO COLOR ***'  ' Cost: '  '0'

  'Time:'  '50'  ' Place:'  'p13'  ' *** NO COLOR ***'  ' Cost: '  '53'
  'Time:'  '60'  ' Place:'  'p13'  ' *** NO COLOR ***'  ' Cost: '  '53'

  'Time:'  '50'  ' Place:'  'p23'  ' *** NO COLOR ***'  ' Cost: '  '750'
  'Time:'  '60'  ' Place:'  'p23'  ' *** NO COLOR ***'  ' Cost: '  '750'

  'Time:'  '40'  ' Place:'  'p34a'  ' *** NO COLOR ***'  ' Cost: '  '5401.5'
  'Time:'  '50'  ' Place:'  'p34a'  ' *** NO COLOR ***'  ' Cost: '  '5401.5'

  'Time:'  '40'  ' Place:'  'p34b'  ' *** NO COLOR ***'  ' Cost: '  '5401.5'
  'Time:'  '50'  ' Place:'  'p34b'  ' *** NO COLOR ***'  ' Cost: '  '5401.5'

  'Time:'  '30'  ' Place:'  'pEnd'  ' *** NO COLOR ***'  ' Cost: '  '10913'
  'Time:'  '40'  ' Place:'  'pEnd'  ' *** NO COLOR ***'  ' Cost: '  '10913'
```

After simulation run, there are two tokens left in each place.

- The tokens in places **p01** and **p02** are initial tokens thus have 0 costs.
- The tokens in **p13** have a cost of 53. This is correct as, for **t1**, ***Total_Cost_of_Input_Tokenens + fc_fixed + (fc_variable * ft)*** = 0 + 3 + 5*10 = 53
- The tokens in **p23** have a cost of 750. This is also correct as, for **t2**, ***Total_Cost_of_Input_Tokenens + fc_fixed + (fc_variable * ft)*** = 0 + 250 + 50*10 = 750
- The tokens in **p34a** and **p34b** have a cost of 5401.5. This is because, for **t3**, ***Cost_of_Firing = Total_Cost_of_Input_Tokenens + fc_fixed + (fc_variable * ft)*** = (53+750) + 5000 + 500*10 = 10803. This cost must be equally divided between the two output tokens. Thus, cost of a token in **p34a** or **p34b** is = 10803/2 = 5401.5
- The tokens in **pEnd** have a cost of 10913. This is because, for **t4**, ***Cost_of_Firing = Total_Cost_of_Input_Tokenens + fc_fixed + (fc_variable * ft)*** = (2*5401.5) + 10 + 10*10 = 10913.

Now, the MSF:

```
% Example-35-1: Firing Costs of Transition
global global_info;
global_info.STOP_AT = 80;

pns = pnstruct('abc1_pdf');

dyn.m0 = {'p01',8, 'p02',8}; % tokens initially
dyn.ft = {'allothers',10};
dyn.fc_fixed = {'t1',3, 't2',250, 't3',5000, 't4',10};
dyn.fc_variable = {'t1',5, 't2',50, 't3',500, 't4',10};

pni = initialdynamics(pns, dyn);
sim = gpensim(pni);

prnfinalcolors(sim);
```

PDF:

```
% Example-35-1: Firing Costs of Transition
function [png] = abc1_pdf()
png.PN_name = 'Simple cost calculation';
png.set_of_Ps = {'p01', 'p02','p13', 'p23', 'p34a', 'p34b', 'pEnd'};
png.set_of_Ts = {'t1', 't2', 't3', 't4'};
png.set_of_As = {'p01','t1',1, 't1','p13',1, ... % t1
    'p02','t2',1, 't2','p23',1, ... % t2
    'p13','t3',1, 'p23','t3',1, 't3','p34a',1, 't3','p34b',1, ... % T3
    'p34a','t4',1, 'p34b','t4',1, 't4','pEnd',1};
```

COMMON_PRE: for limiting the firings of the transitions to the predefined times.

```
function [fire, transition] = COMMON_PRE(transition)

tx = get_trans(transition.name); % get the transition information
times_fired = tx.times_fired; % get the number of times the trans has fired
fire = 1; % initial assumption

if ismember(transition.name, {'t1', 't2'}),
    if eq(times_fired, 6),  % t1 and t2 are allowed to fire only 6 times
        fire = 0;
    end;
elseif strcmpi(transition.name, 't3'),
    if eq(times_fired, 4),  % t3 is allowed to fire only 4 times
        fire = 0;
    end;
elseif strcmpi(transition.name, 't4'),
    if eq(times_fired, 2),  % t4 is allowed to fire only 2 times
        fire = 0;
    end;
end;
```

35.3 Example-35-2: Itertaive Firing Costs of Transition

This is yet another simple example, just to experiment with accumulated costs when transitions fire iteratively. The model shiwn in figure-35-3 depicts that the only transition t1 will fire 3 times exactly; we will study how the costs of the tokens evolve with time.

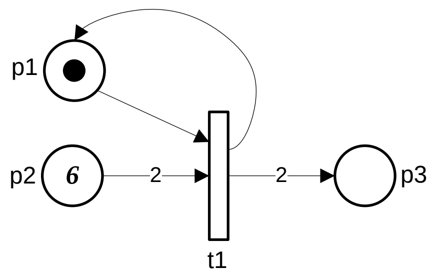


Figure 35-3: Iterative firing

Result of the simulation run is shown below:

```
Colors of Final Tokens:
  'Time:'   '3.1'   ' Place:'   'p1'   ' *** NO COLOR ***'   '  Cost: '   '13'

  'Time:'   '1.1'   ' Place:'   'p3'   ' *** NO COLOR ***'   '  Cost: '   '9'
  'Time:'   '1.1'   ' Place:'   'p3'   ' *** NO COLOR ***'   '  Cost: '   '9'
  'Time:'   '2.1'   ' Place:'   'p3'   ' *** NO COLOR ***'   '  Cost: '   '12'
  'Time:'   '2.1'   ' Place:'   'p3'   ' *** NO COLOR ***'   '  Cost: '   '12'
  'Time:'   '3.1'   ' Place:'   'p3'   ' *** NO COLOR ***'   '  Cost: '   '13'
  'Time:'   '3.1'   ' Place:'   'p3'   ' *** NO COLOR ***'   '  Cost: '   '13'
```

The simulation result show that the tokens in **p3** has cost from 9 to 13. Let us verify:

- Initially, **p1** has one token (cost is 0) and **p2** has 6 tokens (costs of all these initial tokens are 0).
- When **t1** fires for the first time, it takes one token from **p1** and one from **p2**. ***Total_Cost_of_Input_Tokenens = 0.*** Firing of **t1** costs the following:
  Cost_of_Firing = Total_Cost_of_Input_Tokenens + fc_fixed + (fc_variable * ft)
  = 0 + 9 + 1*18 = 27
  This cost has to divided equally between 3 output tokens, thus cost of an output token (1 into **p1** and 2 into **p3**) becomes **9**.
- When **t1** fires for the second time, it again takes one token from **p1** (cost is 9) and one from **p2** (cost is 0). Thus, ***Total_Cost_of_Input_Tokenens*** is 9. Firing of **t1** costs the following:
  Cost_of_Firing = Total_Cost_of_Input_Tokenens + fc_fixed + (fc_variable * ft)
  = 9 + 9 + 1*18 = 36
  This cost has to divided equally between 3 output tokens, thus cost of an output token (1 into **p1** and 2 into **p3**) becomes **12**.
- When **t1** fires for the third time, it again takes one token from **p1** (cost is 12) and one from **p2** (cost is 0). Thus, ***Total_Cost_of_Input_Tokenens*** is 12. Firing of **t1** costs the following:
  Cost_of_Firing = Total_Cost_of_Input_Tokenens + fc_fixed + (fc_variable * ft)
  = 12 + 9 + 1*18 = 39
  This cost has to divided equally between 3 output tokens, thus cost of an output token (1 into **p1** and 2 into **p3**) becomes **13**.

Thus, we find 3 tokens in p3, one with cost 9, the other with 12, and the third with 13. MSF for this example is given below:

```
% Example-35-2: Itertaive Firing Costs of Transition
```

```
global global_info;
global_info.STOP_AT = 10;
global_info.DELTA_TIME = 0.1;
pns = pnstruct('abc2_pdf');

dyn.m0 = {'p1',1, 'p2',6}; % tokens initially
dyn.ft = {'allothers',1};
dyn.fc_fixed = {'t1',9};
dyn.fc_variable = {'t1',18};

pni = initialdynamics(pns, dyn);
sim = gpensim(pni);

plotp(sim, {'p2'});

prnfinalcolors(sim);
```

36. Token Selection based on Cost

As we have seen in the previous section, costs are generated by transitions and it is the tokens that carry the costs around the net as they follow around. Table below shows the select functions that deal with cost of a token in a specific place:

Table 36-1: GPenSIM Functions for selection of tokens based on time

Function	*Description*
`tokenCheap`	Select tokens that are the cheapest in a specific place
`tokenCostBetween`	Select tokens that cost between two limits
`tokenExpensive`	Select tokens that are the most expensive in a specific place

A transition may select input tokens based on cost. Selection can be done by three ways:

- Tokens that are the cheapest in the place,
  [set_of_tokID, nr_token_av] = tokenCheap(place, nr_tokens_wanted)

- Tokens that are the most expensive in the place,
  [set_of_tokID, nr_token_av] = tokenExpensive(place, nr_tokens_wanted)

- Tokens that cost between two limits lower cost (lc) and upper cost (uc).
  [set_of_tokID, nr_token_av] = tokenCostBetween(place, nr_tokens_wanted, lc, uc)

The output parameters of the functions are a set of IDs of the selected tokens (set of tokID ‘set_of_tokID’) and the actual number of valid tokID in the set (‘nr_token_av’).

E.g: if a transition wants **4** **cheapest** tokens from the input place **pBUFF**, then the transition will execute the following statement:

```
function [fire, transition] = tLR_A_pre (transition)

selected_tokens = tokenCheap('pBUFF',4);
fire = all(selected_tokens);
```

If **pBUFF** has more than or equal to 4 tokens, then the returned value **`selected_tokens`** will have tokIDs of the 4 oldest tokens. Otherwise (if **pBUFF** has less than 4 tokens), then **`selected_tokens`** will have tokIDs of all the tokens, padded with 0s.

E.g. if a transition wants **4** **most expensive** tokens from the input place **pBUFF**, where **pBUFF** has only two tokens at that time:

```
function [fire, transition] = tLR_A_pre (transition)

selected_tokens = tokenExpensive('pBUFF',4);
fire = 1;
```

Composition of **`selected_tokens`** will be [tokIDi tokIDj 0 0], where tokIDi and tokIDj are valid tokIDs.

36.1 Example-36-1: Token selection based on cost

Figure-36-1 depicts a simple Petri Net where **t1** is allowed to fire 9 times, depositing 18 tokens into **p2**. Once **t1** has completed 9 firings, the other three transitions (**tChp**, **tMid**, and **tExp**) will start sorting the tokens in **p2**; this is done by **tChp** taking 3 cheapest tokens, **tExp** taking the 3 most expensive tokens, and **tMid** taking tokens that cost in between.

Since sorting costs nothing (the other three transitions **tChp**, **tMid**, and **tExp** do not incur any cost as their firing costs are assumed default 0), the tokens that end up in places **p2**, **pChp**, **pMid**, and **pExp**, has costs that are induced by **t1** alone.

When **t1** fires for the first time, the input token cost nothing, thus the cost of output tokens deposited into **p1** and **p2** become (9+18)/2 = 27/2. When **t1** fires for the second time, it takes the token from **p1** (cost 27/2), and add firing cost (9+18 = 27), thus the total cost becomes (27/2 + 27), which has to be divided by 2 as there are two output tokens.
Subsequently, the cost of ouput token become = 27/2 + 27/4 + 27/8 + …. and so on.

The simulation results show that pChp has 3 cheapest tokens, pMid has tokens that cost between 25.5 and 26.5, and pExp has the most expensive tokens.

```
'Time:'   '11.2'   ' Place:'   'pChp'   ' *** NO COLOR ***'   '  Cost: '   '13.5'
'Time:'   '12.3'   ' Place:'   'pChp'   ' *** NO COLOR ***'   '  Cost: '   '20.25'
'Time:'   '13.4'   ' Place:'   'pChp'   ' *** NO COLOR ***'   '  Cost: '   '23.625'
```

```
'Time:'   '11.2'   ' Place:'   'pExp'   ' *** NO COLOR ***'   '  Cost: '   '26.9473'
'Time:'   '12.3'   ' Place:'   'pExp'   ' *** NO COLOR ***'   '  Cost: '   '26.8945'
'Time:'   '13.4'   ' Place:'   'pExp'   ' *** NO COLOR ***'   '  Cost: '   '26.7891'
```

```
'Time:'   '11.2'   ' Place:'   'pMid'   ' *** NO COLOR ***'   '  Cost: '   '26.1563'
```

MSF:

```
% Example-36-1: token selection based on Costs
clear all; clc;
global global_info;
global_info.STOP_AT = 50;
global_info.DELTA_TIME = 0.1;

pns = pnstruct('abc3_pdf');

dyn.m0 = {'p1',1};
dyn.ft = {'allothers',1};
dyn.fc_fixed = {'t1',9};
dyn.fc_variable = {'t1',18};

pni = initialdynamics(pns, dyn);
sim = gpensim(pni);

prnfinalcolors(sim);
```

PDF:

```
% Example-36-1: token selection based on Costs
function [png] = abc3_pdf()
png.PN_name = 'Ex-35-3: token selection based on Costs';
png.set_of_Ps = {'p1', 'p2', 'pChp', 'pMid','pExp'};
png.set_of_Ts = {'t1', 'tChp', 'tMid','tExp'};
png.set_of_As = {...
    'p1','t1',1, 't1','p1',1, 't1','p2',1, ... % t1
    'p2','tChp',1, 'tChp','pChp',1, ... % tChp
    'p2','tMid',1, 'tMid','pMid',1, ... % tMid
    'p2','tExp',1, 'tExp','pExp',1, ... % tExp
    };
```

COMMON_PRE: this Pre-processor performs the operations:

- stops t1 after 9 firing,
- stops the following transitions after 3 firings: tChp, tMid, and tExp,
- allows tChp to select the cheapest,
- allows tExp to select the most expensive, and
- allows tMid to select token cost within 25.5 and 26.5

```
% Example-36-1: token selection based on Costs
function [fire, transition] = COMMON_PRE(transition)

tx = get_trans(transition.name);
times_fired = tx.times_fired;

% allow t1 to fire only 9 times
if strcmpi(transition.name, 't1'),
    if eq(times_fired, 9);
        fire = 0;return;
    else
        fire = 1; return;
    end;
end;

% freeze the other transitions until t1 has fired 9 times,
% producing all the tokens in pBUFF.
ct = current_time();
if le(ct, 10),
    fire = 0;return;
end;

% allow 'tChp', 'tMid', and 'tExp' to fire ONLY 3 times
if eq(times_fired, 3);
    fire = 0;
    return;
end;

if strcmpi(transition.name, 'tChp'),  % tChp selects cheap tokens
    tokID = tokenCheap('p2',1);
elseif strcmpi(transition.name, 'tExp'), % tExp selects expensive tokens
    tokID = tokenExpensive('p2', 1);
else % 'tMiddle'    % tMid selects tokens that cost between (25.5, 26.5)
    tokID = tokenCostBetween('p2', 1, 25.5, 26.5);
end;

transition.selected_tokens = tokID;
fire = tokID;
```

37. Resource Usage Cost

In the two previous chapters, we studied about cost additions due to transition firing. However, if the model uses (consume) resources, then resource cost will also come into the calculations, in addition to the transition firing costs.

Just like transition firing costs, each resource instance can also have two costs:

1. **Fixed cost (*rc_fixed*)**: this is one time cost, as each time a transition start using a resource, this cost will be added to the total cost.
2. **Variable cost (*rc_variable*)**: this cost per unit of time resource usage. Thus, this cost will be multiplied with firing time before added to the total resource cost.

The resource costs are declared in the MSF, as a part of initial dynamics. For example,

```
% first costs analysis
…
…

dyn.ft = {'t1',2, 't2',5,'allothers',1};
dyn.fc_fixed = {'t1',50, 't2',100};
dyn.fc_variable = {'t1',20, 't1',25};

dyn.re = {'R1',1,inf, 'R2',1,inf};
dyn.rc_fixed = {'R1',750, 'R2',1000};
dyn.rc_variable = {'R1',300, 'R2',250};

pni = initialdynamics(pns, dyn);

sim = gpensim(pni);
```

In the code snippet shown above, there are two resource types (‘**R1**’ amd ‘**R2**’) and each resource type has only one instance. Fixed cost of **R1** is NOK 750 and the variable cost of **R1** is NOK 1000 per unit of firing time. Whereas, the fixed cost of **R2** is NOK 300, and the variable cost of **R2** is NOK 250 (per unit of firing time).

38. Example-38-1: Machine and Resource Usage Costs

In this section, we shall work on a simple problem of finding total costs of production when machines and resources add costs to the production.

38.1 The problem of finding the costs of using alternative printers

A commercial printing facility is planning to buy a printering machine. There are two alternatives to consider as two different printers can be used for the same job:

- Printer-A is faster but costs more to operate.
- Printer-B is slower but cheaper to operate.
- These two printers also use different amount of resources (inks).

The factsheet from the manufactures of the printers are given below.

Table-38-1: Properties of the printers for a typical job

	Production time	**Production Cost**	
		Fixed	**Varbiable**
Printer-A	10	1000	100
Printer-B	20	500	50

During the printing operations, the two printers use different types of inks:

- Printer-A uses one tube of ink "Premia" and tube of ink "Rainbow".
- Printer-B uses two tubes of "Rainbow".

The table-38-2 shows the properties of the inks.

Table-38-2: Properties of the inks

Resource type	**Number of tubes**	**Resource Cost**	
		Fixed	**Varbiable**
Ink "Premia"	1	1000	100
Ink "Rainbow"	3	500	50

38.2 Which printer to choose?

A quick look into table-38-1 reveals that Printer-A is two times faster than Printer-B, but Printer-A is also twice expensive than Printer-B. Hence, it seems that the choice is between fast or cheap. However, as the company wants numerical answers, we are going to make a Petri Net model of the problem and then run simulations to get numerical answers.

38.3 The Model

Figure-38-1 shows a production facility where both printing machines are employed to run 6 sample jobs (test jobs). In the model, the resource usages of the two machines are also indicated.

PDF:

```
% Example-38-1: machine and resource Usage Cost
function [png] = mruc_pdf()
png.PN_name = 'ex-38-1: machine and resource Usage Cost';
png.set_of_Ps = {'In', 'Out-A','Out-B'};
png.set_of_Ts = {'Printer-A', ' Printer-B'};
png.set_of_As = {'In','Printer-A',1, 'Printer-A','Out-A',1, ... % Printer-A
                 'In','Printer-B',1, 'Printer-B','Out-B',1}; % Printer-B
```

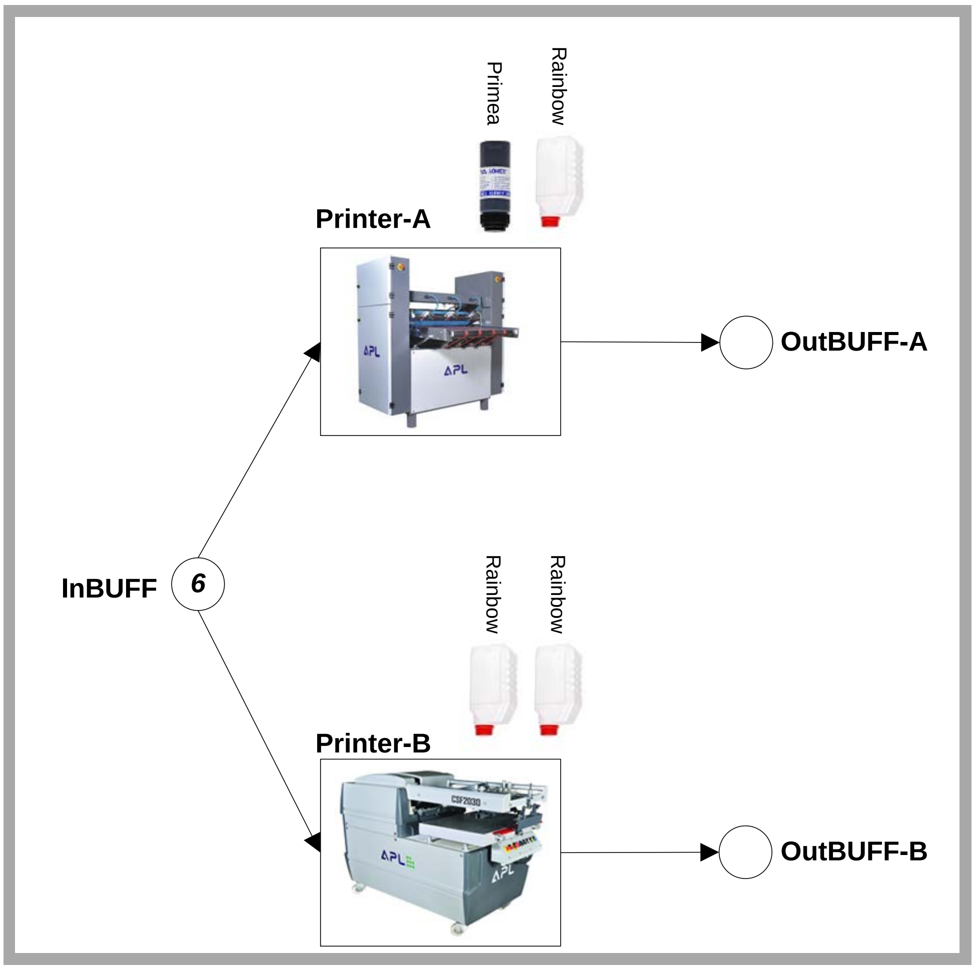


Figure 38-1: Petri Net model of the production facility to calculate production costs

MSF:

```
% Example-38-1: machine and resource Usage Cost
clear all; clc;
global global_info;
global_info.STOP_AT = 80;

pns = pnstruct('mruc_pdf');

dyn.m0 = {'In',6}; % tokens initially
dyn.ft = {'Printer-A',10, 'Printer-B',20};
dyn.fc_fixed =  {'Printer-A',1000, 'Printer-B',500};
dyn.fc_variable={'Printer-A',100,  'Printer-B',50};

dyn.re = {'Premia',1,inf, 'Rainbow',3,inf};
dyn.rc_fixed = {'Premia',1000, 'Rainbow',500};
dyn.rc_variable = {'Premia',100, 'Rainbow',50};

pni = initialdynamics(pns, dyn);
sim = gpensim(pni);

prnfinalcolors(sim);
prnschedule(sim);
```

COMMON_PRE:
In COMMON_PRE, Printer-A reserves once instance of each Premia and Rainbow, whereas Printer-B reserves two instances of Rainbow:

```
% Example-38-1: resource Usage Cost
function [fire, transition] = COMMON_PRE(transition)

if strcmpi(transition.name, 'Printer-A'),
    granted = request(transition.name, {'Premia',1, 'Rainbow',1});
elseif strcmpi(transition.name, 'Printer-B'),
    granted = request(transition.name, {'Rainbow',2});
end;

fire = granted;
```

COMMON_POST:
In addition to releasing the resources used by a transiton, COMMON_POST has another important function to perform: when pInBUFF becomes empty, it is better to stop the simulations; otherwise, simulation runs longer, resulting in lower value for the efficiency of resource usage.

```
% Example-37-1: resource Usage Cost
function [] = COMMON_POST(transition)
global global_info;

release(transition.name);

% if p0 has no tokens then stop simulations immediately
pIB = get_place('In');

if not(pIB.tokens),
    global_info.STOP_SIMULATION = 1;
end;
```

<u>Simulation results</u>:
Function prnfinalcolors(sim) gives the following results:

First of all, there are 4 tokens (product-As) in pA, each with a cost of 9250.

```
'Time:'   '10'   ' Place:'   'pA'   ' *** NO COLOR ***'   ' Cost: '   '9250'
'Time:'   '20'   ' Place:'   'pA'   ' *** NO COLOR ***'   ' Cost: '   '9250'
'Time:'   '30'   ' Place:'   'pA'   ' *** NO COLOR ***'   ' Cost: '   '9250'
'Time:'   '40'   ' Place:'   'pA'   ' *** NO COLOR ***'   ' Cost: '   '9250'
```

There are 2 tokens (product-Bs) in pB, each with a cost of 13500.

```
'Time:'   '20'   ' Place:'   'pB'   ' *** NO COLOR ***'   ' Cost: '   '13500'
'Time:'   '40'   ' Place:'   'pB'   ' *** NO COLOR ***'   ' Cost: '   '13500'
```

Since there are no other tokens anywhere, the total cost of production is:

(4 * 9250) + (2*13500) = 64000

The total production cost (64000) is confirmed the print out of **prnschedule(sim)**:

```
*****  LINE EFFICIENCY AND COST CALCULATIONS: *****
  Number of servers:  k = 2
  Total number of server instances:  K = 4
  Completion = 40
  LT = 160
  Total time at Stations: 160
  LE = 100 %
  **
  Sum resource usage costs: 53000   (82.8125% of total)
  Sum firing costs: 11000   (17.1875% of total)
  Total costs:  64000
  **
```

The info above while confirming the total production cost (64000), it also states that the total cost can be divided into 53000 as resource usage cost and 11000 as machining (firing) costs.
This is because:

- tA fired 4 times (because there are 4 tokens in pA) costing (4 * 1000) + (4 * 10 * 100) = 8000
- tB fired 2 times (because there are 2 tokens in pB) costing (2 * 500) + (2 * 20 * 50) = 3000
- Thus, total machining costs (firing costs) is 11000

Resource usage given below, it is clear that:

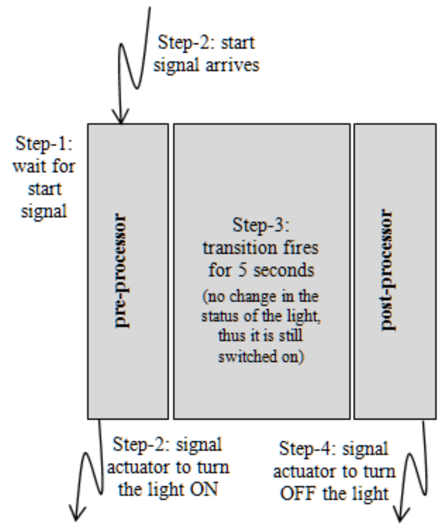
- R1 is used in 4 firings, for a total time of 40, amounting to costs: (4 * 750 + 40 * 300) = 15000
- R2 is used in 8 firings, for a total time of 120, amounting to costs: (8 * 1000 + 120 * 250) = 38000
- Thus, total resource costs is 53000!

```
RESOURCE USAGE:

RESOURCE INSTANCES OF ***** R1 *****
tA [0 : 10]
tA [10 : 20]
tA [20 : 30]
tA [30 : 40]
Resource Instance: R1:: Used 4 times. Utilization time: 40

RESOURCE INSTANCES OF ***** R2 *****
(R2-1):   tA [0 : 10]
(R2-1):   tA [10 : 20]
(R2-2):   tB [0 : 20]
(R2-3):   tB [0 : 20]
(R2-1):   tA [20 : 30]
(R2-1):   tA [30 : 40]
(R2-2):   tB [20 : 40]
(R2-3):   tB [20 : 40]
Resource Instance: (R2-1):: Used 4 times.  Utilization time: 40
Resource Instance: (R2-2):: Used 2 times.  Utilization time: 40
Resource Instance: (R2-3):: Used 2 times.  Utilization time: 40

RESOURCE USAGE SUMMARY:
R1:  Total occasions: 4   Total Time spent: 40
R1:  Total resource usage cost: 15000
R2:  Total occasions: 8   Total Time spent: 120
R2:  Total resource usage cost: 38000
```

Part-V: Internal Data Structures and Interacting with External Environment

39. Real-Time Application

GPenSIM supports interacting with external devices through processors (pre and post). In GPenSIM, places are generally assumed passive and the transitions are assumed active. Thus, GPenSIM maps transitions to activate or deactivate external devices. Figure below shows how a transition can be used to control an external device. Specifically, the transition is to be used to switch on a light for 5 seconds:



Step-2: start signal arrives
Step-1: wait for start signal
pre-processor
Step-3: transition fires for 5 seconds
(no change in the status of the light, thus it is still switched on)
post-processor
Step-2: signal actuator to turn the light ON
Step-4: signal actuator to turn OFF the light

Fig.5. Using transition to interact with the external environment

- The pre-processor (Pre-processor) of the transition, as usual, first checks whether the transition can fire by going through (if any) additional conditions for firing. The pre-processor can also run any commands coded in the Pre-processor file: the relevant command for the light switching example will be to **switch the light on**.
- Followed by the pre-processor, the firing transition enters sleeping in the "firing_transition" queue; firing transition is just a passive time delay that will not consume CPU. **A firing transition sleeping during the firing time (5 seconds, in this example),** will automatically awaken after the firing is complete;
- When the firing is complete, the post-processor (Post-processor) will be executed, executing all the commands on it; in the example, Post-processor will switch off the light.
- Summary: (Pre-processor switching on the light – transition sleeps (delayed) for 2 seconds – and the Post-processor switching off the light): results in the light switched on for 5 seconds.

IMPORTANT!!!:
Remember to set the REAL_TIME flag to 'on' in the main simulation file.

```
Global_info.REAL_TIME = 1
```

39.1 Example-39-1: Alternating Lights

This example is the same as the one described in example-09-01: Alternative firing using binary semaphore. But this time, we will use real hardware, a LEGO NXT, to alternatively 'ON' and 'OFF' Red and Green lights. Of course, we need the following software and hardware installed in the system:

- The RWTH–Mindstorms NXT Toolbox

- LEGO Mindstorms NXT building kit 2.0; LEGO Mindstorms NXT firmware v1.26 or higher
- For Bluetooth wireless connection: Bluetooth 2.0 adapter recommended by LEGO
- For USB connections: Mindstorms NXT Driver “Fantom” v1.02

The RWTH–Mindstorms NXT Toolbox [19] is a software package available for the MATLAB environment that allows control of LEGO Mindstorms NXT robots from a PC; a LEGO robot can be controlled from a PC either through USB connection or through Bluetooth wireless connection.

By using GPenSIM on top of RWTH–Mindstorms NXT Toolbox, various discrete control applications (especially, supervisory control applications) can be developed using LEGO robot as a specimen. Figure-38-1 shows the various layers of software that can be used to develop applications for discrete robotic control.

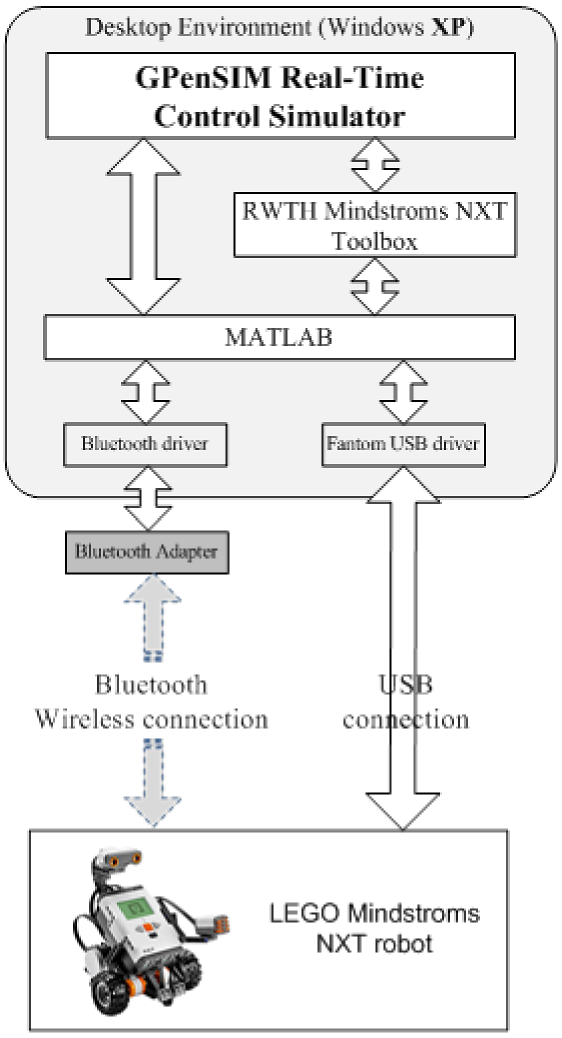

Desktop Environment (Windows XP)
GPenSIM Real-Time Control Simulator
RWTH Mindstroms NXT Toolbox
MATLAB
Bluetooth driver
Fantom USB driver
Bluetooth Adapter
Bluetooth Wireless connection
USB connection
LEGO Mindstroms NXT robot

Figure 38-1: Using LEGO NXT Mindstroms with GPenSIM

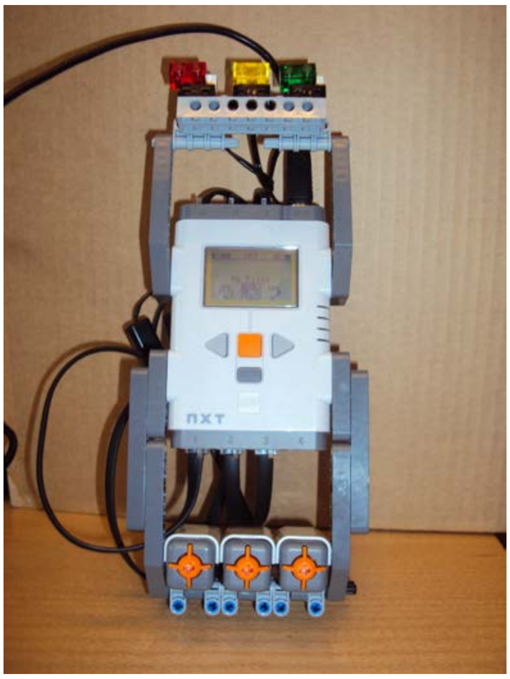


Figure 38-2: LEGO NXT Mindstrom robot setup with lights and switches

Figure-38-2 shows the LEGO Mindstorms NXT robot setup with lights (red, green, and yellow) and switches (‘buttons’).

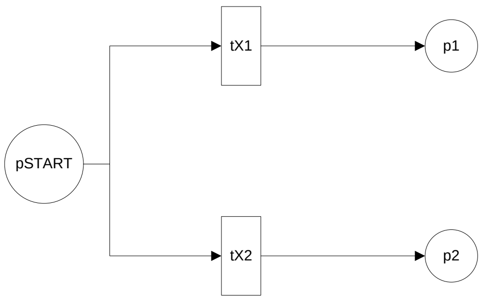


Figure 38-3: The Petri Net model for alternative firing

PDF: (NO change)

```
% Example-39-01: Rea-Time Load balance (with LEGO NXT)
function [png] = loadbalance_pdf()

png.PN_name = 'Load Balancer';
png.set_of_Ps = {'p0', 'p1', 'p2'};
png.set_of_Ts = {'t1', 't2'};
png.set_of_As = {'p0','t1',1, 't1','p1',1,'p0','t2',1, 't2','p2',1};
```

MSF: the only changes are the hardware initialization and real-time settings.

```
% Example-39-01: Rea-Time Load balance (with LEGO NXT)
clear all; clc;

global global_info;
global_info.REAL_TIME = 1;    % set REAL_TIME flag
global_info.STOP_AT = current_clock(3)+[0 0 30];%stop after 30 secs
global_info.semaphore = 't1'; % GLOBAL DATA: binary semafor

init_TL_NXT();  % Initialize LEGO H/W

png = pnstruct('loadbalance_pdf');

dyn.m0 = {'p0',9};
dyn.ft = {'t1',2, 't2',2};
pni = initialdynamics(png, dyn);

sim = gpensim(pni);
plotp(sim, {'p1', 'p2'});

close_TL_NXT();% Important: clear memory (and hardware) after use
```

init_TL_NXT: Init LEGO NXT hardware

```
% Example-39-01: Rea-Time Load balance (with LEGO NXT)
function [] = init_TL_NXT()
% initialize the lights in NXT

global global_info;

% initializ NXT
warning('off', 'MATLAB:RWTHMindstormsNXT:noEmbeddedMotorControl');

COM_CloseNXT all
hNXT = COM_OpenNXT('bluetooth.ini');       % look for USB devices
COM_SetDefaultNXT(hNXT);    % sets global default handle

global_info.NXT_handle = hNXT;

global_info.lightRED = MOTOR_A;
global_info.lightGREEN = MOTOR_C;

NXT_PlayTone(440, 500); % just play a music to indicate ready
```

Close_TL_NXT: clear memory and hardware after use.

```
function [] = close_TL_NXT()
% function [] = close_TL_NXT()

global global_info;

SwitchLamp(global_info.lightRED, 'off');
SwitchLamp(global_info.lightGREEN, 'off');

% Never forget to clean up after your work!!!
COM_CloseNXT(global_info.NXT_handle);
```

COMMON_PRE:

```
function [fire, trans] = COMMON_PRE(trans)
% COMMON_PRE file codes the enabling conditions
% Here, the current value of the semaphore
% indicate which transiton can fire.
%
% After firing, the fired transition must
% set the value of semaphore to the other
% transition: this is done in the COMMON_POST
% file.

global global_info;

if strcmpi(trans.name, 't1'),
    fire = strcmp(global_info.semaphore, 't1');
    if (fire), SwitchLamp(global_info.lightRED, 'on'); end;

else strcmp(trans.name, 't2'),
    fire = strcmp(global_info.semaphore, 't2');
    if (fire), SwitchLamp(global_info.lightGREEN, 'on'); end;
end;
```

COMMON_POST:

```
function [] = COMMON_POST(transition)
% COMMON_POST file codes the post firing actions.
% Here, right after firing, the fired transition
% set the value of semaphore to the other
% transition so that the other one fires next

global global_info;

if strcmp(transition.name, 't1'),
    SwitchLamp(global_info.lightRED, 'off');
    global_info.semaphore = 't2'; % t1 releases semafor to t2
else % transition.name 't2'
    SwitchLamp(global_info.lightGREEN, 'off');
    global_info.semaphore = 't1'; % t2 releases semafor to t1
end;
```

39.2 Example-39-2: Alternating Lights without Real-Time Hardware

This is the same example as example-31-1. The previous example, we used some hardware to see the real-time action. In this example, we willnot use any hardware; we will simply echo some outputs to the screen.

In this example, just as in the example-31-1, we will let tX1 and tX2 fire alternatively. However, we want tX1 to fire for just 1 second, and tX2 to fire for 2 seconds. Before and after each firing, we want echos on the screen so that we can see the real-time action on the screen.

PDF file will be the same as before:

```
% ex-39-02: Real-time without hardware
% file: loadbalance_pdf.m

function [png] = loadbalance_pdf()

png.PN_name='Real-time without hardware';
png.set_of_Ps = {'pSTART', 'p1', 'p2'};
png.set_of_Ts = {'tX1','tX2'};
png.set_of_As = {'pSTART','tX1',1, 'tX1','p1',1,...
                 'pSTART','tX2',1, 'tX2','p2',1};
```

In the MSF, we let the system know that we want real-time action, by setting on the REAL_TIME flag. In addition, the firing times of tX1 and Tx2 will be 1 and 2 seconds, respectively.

```
% example-39-2: Real-time simulation without hardware

clear all; clc;

global global_info;

global_info.semafor = 1;    % GLOBAL DATA: binary semafor
global_info.STOP_AT = current_clock(3) + [0 0 15]; % after 15 secs
global_info.REAL_TIME = 1;  % This is a Real-Time run

pns = pnstruct('loadbalance_pdf');
dyn.m0 = {'pSTART', 10};

dyn.ft={'tX1',1,'allothers',2};% all trans timed
pni = initialdynamics(pns, dyn);
sim = gpensim(pni);
plotp(sim, {'p1', 'p2'});
```

In the COMMON_PRE, we will insert code to echo the real-time before the start of firing:

```
% COMMON_PRE.m
function [fire, trans] = COMMON_PRE(trans)
%

global global_info;

if strcmp(trans.name, 'tX1'),
    fire = eq(global_info.semafor, 1);
elseif strcmp(trans.name, 'tX2'),
    fire = eq(global_info.semafor, 2);
else
    disp('transition name is neither "tX1", nor "tx2" ?');
    disp('this can never happen');
end;

if fire,
    disp(['transition "', trans.name, ...
      '" is starting to fire ', 'at ', rt_clock_string(), ' !']);
end;
```

Similarly, in the COMMON_POST, we will insert code to echo the real-time after the firing completion:

```
function [] = COMMON_POST(transition)
global global_info;

disp(['transition "', transition.name, ...
  '" completed firing at ', rt_clock_string(), '!']);
disp(' ');

if strcmp(transition.name, 'tX1'),
    global_info.semafor = 2; % tX1 releases semafor to tX2
else % transition.name 'tX2'
    global_info.semafor = 1; % tX2 releases semafor to tX1
end;
```

The simulation results show that the transitons tX1 and tX2 fires alternatively (NO overlapping) and that they fire for 1 and 2 seconds, respectively. Also, the simulation was started at hour 18:52.

```
transition "tX1" is starting to fire at 18:52:21 !
transition "tX1" completed firing at 18:52:22!

transition "tX2" is starting to fire at 18:52:22 !
transition "tX2" completed firing at 18:52:24!

transition "tX1" is starting to fire at 18:52:24 !
transition "tX1" completed firing at 18:52:25!

transition "tX2" is starting to fire at 18:52:25 !
transition "tX2" completed firing at 18:52:27!

transition "tX1" is starting to fire at 18:52:27 !
transition "tX1" completed firing at 18:52:28!

transition "tX2" is starting to fire at 18:52:28 !
>>
```

40. Internal Data Structures of the basic PN Elements

In this section, we will look into the data structures of some of the basic elements in a Petri net model such as places, transitions and resources. As usual, we will inspect the data stuctures through an example given below.

40.1 Example-40-1: Data structures

Figure-40-1 shows a simple Petri net. The model has 4 places (**p1**, **p2**, **p3**, and **p4**) 2 transitions (**tA** and **tB**), and 3 resources that are not shown in the model (**rX**, **rY**, and **rZ**).

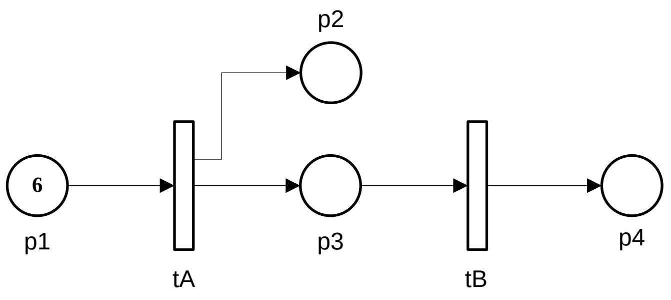


Figure 40-1: A simple Petri Net model for inspecting the internal data structures

```
% Example-40-1: data structures of basic PN elements
function [png] = ds_pdf()
png.PN_name = 'ex-40-1: data structures of basic PN elements';
png.set_of_Ps = {'p1', 'p2', 'p3', 'p4'};
png.set_of_Ts = {'tA', 'tB'};
png.set_of_As = {'p1','tA',1, 'tA','p2',1, 'tA','p3',1,... % tA
                 'p3','tB',1, 'tB','p4',1}; % tB
```

Initial dynamics: **p1** has 6 initial tokens; firing times of **tA** and **tB** are 1 and 2 TU, respectively. For firing, **tA** needs 1 instance of **rX** and 1 instance of **rY;** wheras, **tB** needs 1 instance **rY**, and 2 instances of **rZ**.

```
% Example-40-1: data structures of basic PN elements
clear all; clc;
global global_info;
global_info.check_up_time = false;
global_info.STOP_AT = 18;

pns = pnstruct('ds_pdf');

dyn.m0 = {'p1',6}; % tokens initially
dyn.ft = {'tA',1, 'tB',2};
dyn.fc_fixed =  {'tA',1000, 'tB',500};
dyn.fc_variable={'tA',100,  'tB',50};

dyn.re = {'rX',1,inf, 'rY',2,inf, 'rZ',3,inf};
dyn.rc_fixed = {'rX',110, 'rY',120, 'rZ',130};
dyn.rc_variable = {'rX',10, 'rY',20, 'rZ',30};

pni = initialdynamics(pns, dyn);
sim = gpensim(pni);
```

Probing the data structures:
When the time becomes 4 TU **tA** becomes enabled again and the COMMON_PRE will be run for **tA**. At this point, we will interrupt the COMMON_PRE and printout some of the data structures before **tA** starts firing. When **tA** completes firing COMMON_POST post will be run, and also here we will printout some of the data structures.

```
function [fire, transition] = COMMON_PRE(transition)

global PN;
global global_info;

ctime = current_time;
check_up_time = eq(ctime, 4);

%%%%%%%%%%%%%%%%%%
% Print the data structure of p2, p3, tA
if check_up_time,
    disp(['Enabled transitions is: ', transition.name]);
    disp(' '); disp('The global PN dynamic structure:'); disp(PN);
    disp(' '); disp('Structure of the place "p2":');
    disp(PN.global_places(2));
    disp('Structure of the place "p3":'); disp(PN.global_places(3));
    disp('Structure of the token bank at "p2":');
    for i = 1:length(PN.global_places(2).token_bank),
        disp(PN.global_places(2).token_bank(i));
    end;
    disp(' ');
    disp('Structure of the transition "tA" before aquiring resources:');
    disp(PN.global_transitions(1));
end;
%%%%%%%%%%%%%%%%%%

if strcmpi(transition.name, 'tA'),
    granted = request(transition.name, {'rX',1, 'rY',1});
elseif strcmpi(transition.name, 'tB'),
    granted = request(transition.name, {'rY',1, 'rZ',2});
end;

if check_up_time,
    disp(' ');
    disp('Structure of "tA" after aquiring resources:');
    disp(PN.global_transitions(1));
end;

global_info.check_up_time = check_up_time;

fire = granted;
```

```
function [] = COMMON_POST(transition)

global PN;
global global_info;

check_up_time = global_info.check_up_time;
check_up_time = and(check_up_time, strcmp(transition.name, 'tA'));

%%%%%%%%%%%%%%%%%%
% Print the data structure of resources
```

```
if check_up_time,
    ctime = current_time;
    disp(' ');
    disp(['This is COMMON_POST, at ', num2str(ctime)]);
    disp(' ');
    disp('Before releasing the resources by tA: Structure of "tA":');
    disp(PN.global_transitions(1));
    disp(PN.system_resources(1));
    disp(PN.system_resources(1).instance_usage);
    disp(PN.system_resources(2));
    disp(PN.system_resources(2).instance_usage);
    disp(PN.system_resources(3));
    disp(PN.system_resources(3).instance_usage);
end;
%%%%%%%%%%%%%%%%%%

release(transition.name);

%%%%%%%%%%%%%%%%%%
% Print the data structure of resources
if check_up_time,
    disp(' ');
    disp('After releasing the resources: Structure of "tA":');
    disp(PN.global_transitions(1));
    disp(PN.system_resources(1));
    disp(PN.system_resources(1).instance_usage);
    disp(PN.system_resources(2));
    disp(PN.system_resources(2).instance_usage);
    disp(PN.system_resources(3));
    disp(PN.system_resources(3).instance_usage);
end;
%%%%%%%%%%%%%%%%%%
```

40.2 Data structures

The simulation results of this example show data structures of some of the basic elements.

40.2.1 The data structure of PN – the global Petri Net run-time structure

Given below is the data structure of **PN** – the global Petri Net run-time structure. This structure is available everywhere as long as we declare it as a global parameter (“global **PN**”) in the beginning of the file (in any specific pre, COMMON_PRE, specific post, and COMMON_POST). This structure is big and contains all the static as well as dynamic information:

```
The global PN dynamic structure:
                            name: 'Data structures of basic PN elements'
                   global_places: [1x4 struct]
                    No_of_places: 4
              global_transitions: [1x2 struct]
               No_of_transitions: 2
                  global_Vplaces: [1x4 struct]
                incidence_matrix: [2x8 double]
           Inhibited_Transitions: [0 0]
                Inhibitors_exist: 0
                inhibitor_matrix: []
                         delta_T: 0.2500
                       REAL_TIME: 0
                        HH_MM_SS: 0
```

```
                     current_time: 4
                        STOP_TIME: 18
                                X: [2 4 2 1]
                               VX: [0 0 1 0]
                token_serial_numer: 15
          Set_of_Firing_Costs_Fixed: [1000 500]
       Set_of_Firing_Costs_Variable: [100 50]
              Firing_Costs_Enabled: 1
                 COST_CALCULATIONS: 1
               Set_of_Firing_Times: [1 2]
                     priority_list: [0 0]
                  system_resources: [1x3 struct]
            No_of_system_resources: 3
                Resource_usage_LOG: [11x5 double]
                         PRE_exist: [0 0]
                        POST_exist: [0 0]
                        COMMON_PRE: 1
                       COMMON_POST: 1
                Firing_Transitions: [0 1]
               Enabled_Transitions: [1 1]
                             State: 5
```

40.2.2 The data structure of place

At time equals to 4 TU, the places **p2** and **p3** have 4 and 2 tokens, respectively. Given below is the data structure for a place, with values filled in for **p2** and **p3**, at the time 4 TU:

```
 Structure of the place "p2":
          name: 'p2'
        tokens: 4
    token_bank: [1x4 struct]

Structure of the place "p3":
          name: 'p3'
        tokens: 2
    token_bank: [1x2 struct]
```

The structure for **p2** has 4 tokens and the token bank contains the details about the 4 tokens, as shown below. The values shown below states that the first token was put into **p2** at time equals to 1 TU ('creation_time = 1'), the second at time equals to 2 TU, and so on. All the tokens were made with the same cost of 680 cost units, and none of them have colors.

```
Structure of the token bank at "p2":
            tokID: 7
    creation_time: 1
            color: {}
             cost: 680

            tokID: 9
    creation_time: 2
            color: {}
             cost: 680

            tokID: 11
    creation_time: 3
            color: {}
             cost: 680

            tokID: 14
    creation_time: 4
            color: {}
```

```
              cost: 680
```

40.2.3 The data structure of transiton

The data structure of the transiton **tA** at time equal to 4 TU is shown below, right before **tA** requesting the resources; **tA** needs 1 instance **rX** and 1 instance of **rY**.

```
Structure of the transition "tA" before aquiring resources:
                    name: 'tA'
             firing_time: 1
             firing_cost: 0
             times_fired: 4
         absorbed_tokens: [0 0 0 0]
       firing_cost_fixed: 1000
    firing_cost_variable: 100
      resources_reserved: [0 0 0]
         resources_owned: [0 0 0]
```

After requesting and successfully acquiring the needed resources, in the data structure for **tA**, the field '`resources_reserved`' changes reflecting acquired resources; the field '`resources_reserved`' is a vector of dimension equal to the number of resources. The elements in this vector represent how many instances were acquired (1 **rX**, 1 **rY**, and 0 **rZ**). But still, **tA** has not started firing, as we are still inside COMMON_PRE.

```
Structure of "tA" after aquiring resources:
                    name: 'tA'
             firing_time: 1
             firing_cost: 0
             times_fired: 4
         absorbed_tokens: [0 0 0 0]
       firing_cost_fixed: 1000
    firing_cost_variable: 100
      resources_reserved: [1 1 0]
         resources_owned: [0 0 0]
```

Once we leave COMMON_PRE, **tA** starts firing. This means, all the reserved resources becomes resources under usage ('`resources_owned`') by **tA**:

```
Before releasing the resources by tA: Structure of "tA":
                    name: 'tA'
             firing_time: 1
             firing_cost: 0
             times_fired: 5
         absorbed_tokens: [0 0 0 0]
       firing_cost_fixed: 1000
    firing_cost_variable: 100
      resources_reserved: [0 0 0]
         resources_owned: [1 1 0]
```

At the completion of firing, if the command `release('tA')` is executed in COMMON_POST, then **tA** no longer owns any resources.

```
After releasing the resources: Structure of "tA":
                    name: 'tA'
             firing_time: 1
             firing_cost: 0
             times_fired: 5
         absorbed_tokens: [0 0 0 0]
       firing_cost_fixed: 1000
    firing_cost_variable: 100
      resources_reserved: [0 0 0]
```

```
          resources_owned: [0 0 0]
```

40.2.4 The data structure of resource

Given below are the data structures of the resources **rX**, **rY**, and **rZ**.

```
               name: 'rX'
      max_instances: 1
            MAX_CAP: Inf
           rc_fixed: 110
        rc_variable: 10
     instance_usage: [4x1 double]

               name: 'rY'
      max_instances: 2
            MAX_CAP: Inf
           rc_fixed: 120
        rc_variable: 20
     instance_usage: [4x2 double]

               name: 'rZ'
      max_instances: 3
            MAX_CAP: Inf
           rc_fixed: 130
        rc_variable: 30
     instance_usage: [4x3 double]
```

The field ‘instance_usage’ is a matrix of 4 rows. The number of column of this matrix is equal to the number of instances of the resource. For example, **rX** has only one instance thus its field ‘instance_usage’ will be 4X1 matrix, whereas **rY** has two instances thus its ‘instance_usage’ will be 4X2 matrix. The value shown below is for the field ‘instance_usage’ of **rZ** at time equals to 5 TU.

```
     2     2     0
     3     3     0
     2     2     0
     0     0     0
```

The rows of the field ‘instance_usage’:

- The first row of this matrix shows which instances are under use and by which transition: the first and second instances of the resource **rZ** are used by transition **tB** (**tA** has index 1 and **tB** has index 2), and the third instance is unused. Thus, the first row of the matrix is the vector [2 2 0].
- The second rwo shows that time of the start of use; as both the first and the second instances of **rZ** was taken into use by **tB** at time equals to 3 TU, the second row become the vector [3 3 0].
- The third row reveals how long each instance will be used; as **tB** takes 2 TU to fire, the third row becomes [2 2 0].
- The final rwo is unused.

41. PNML-2-GPenSIM Converter

This section shows how the PNML capability enhances the usefulness of GPenSIM. This section proposes an approach which eliminates the need for GUI with graphic editor of GPenSIM. The approach consists of the following three steps:

1. Basic Animations and Simulation: The initial Petri Net model can be defined using a graphical Petri Net editor like PIPE [PIPE]. Using the same tool, basic token game animations can also be done to verify whether the crude model behave as it should be. Once the modeler is satisfied with the crude model, he can save the model as a PNML file.
2. Importing the initial Petri Net model into GPenSIM: PNML file is imported into GPenSIM environment; using the PNML-2-GPenSIM converter, the static Petri Net structure is extracted and put in the PDF and the initial dynamics (initial markings and the firing times of the transitions) in the MSF.
3. Advanced modeling and simulation: Once the modeler is satisfied with the basics simulations in GPenSIM, enabling functions, transition priorities, resources, and any advanced facilities available in GPenSIM can be coded the processor files (pre and post) to make the advanced model.

41.1 PNML-2-GPenSIM Converter

Function **pnml2gpensim** is the converter that reads a PNML document describing a Petri Net model, extracts the Petri Net structure, and then creates PDF and MSF files representing the model. During this process, the graphical details coded in the PNML file are discarded.

Usage:

```
pnml2gpensim(PNMLFile);
```

The PNML-2-GPenSIM converter is built on MATLAB platform. MATLAB offers a set of functions for reading and interpreting XML files, starting with the function ‘xmlread’ which reads an XML document and returns Document Object Model (DOM) node. From the DOM node, the elements of the node (such as ‘place’, ‘transition’ and ‘arc’) can be visited recursively, extracting the names of the elements, the initial marking (in case of place element), the source and the target (in case of an arc element). The following steps are involved in the PNML-2-GPenSIM conversion:

1. Convert XML file to MATLAB structure and get the root of the DOM tree
2. From the root tree, recursively visit the child nodes to get the PNML structure
3. From the PNML structure, get the 'net' child and start extracting Petri Net structure (places, transitions, and arcs)
4. Write the Petri Net structure into the GPenSIM files MSF and PDF.

41.2 Example-41-01: Generating GPenSIM files from a PNML file

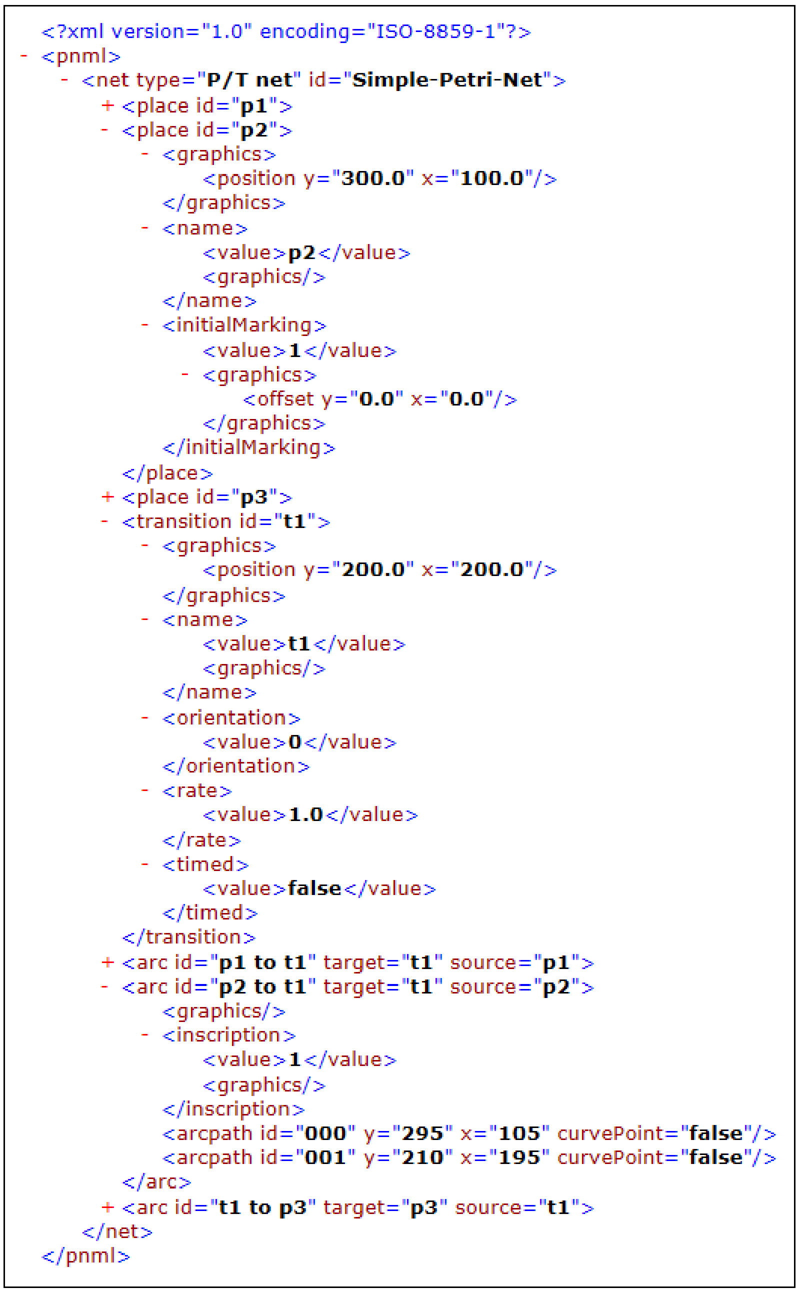
Let us assume that we have generated a PNML file, named “fms.xml” with the help of a Petri Net tool (for example, PIPE2 [xx]). The file “samplePNML1.xml” is shown is figure-39-1. Just passing this file to the function **pnml2gpensim** will produce all the necessary GPenSIM files (MSF, PDF); in addition, a template for COMMON_PRE and COMMON_POST will be also generated.

Generating GPenSIM files from PNML File:

```
% Example-41-1: Convert PNML file into GPenSIM files
clear all; clc;

PNMLfile = 'samplePNML1.xml';
pnml2gpensim(PNMLfile);
```

```
disp(' ');
disp(' ************** ');
disp('GPenSIM files are generated for the PNML file: ');
disp(['              ', PNMLfile]);
```

The generated MSF will be named “msf.m” and the PDF will be “pdf_pdf.m”.



```
  <?xml version="1.0" encoding="ISO-8859-1"?>
- <pnml>
  - <net type="P/T net" id="Simple-Petri-Net">
    + <place id="p1">
    - <place id="p2">
      - <graphics>
          <position y="300.0" x="100.0"/>
        </graphics>
      - <name>
          <value>p2</value>
          <graphics/>
        </name>
      - <initialMarking>
          <value>1</value>
        - <graphics>
            <offset y="0.0" x="0.0"/>
          </graphics>
        </initialMarking>
      </place>
    + <place id="p3">
    - <transition id="t1">
      - <graphics>
          <position y="200.0" x="200.0"/>
        </graphics>
      - <name>
          <value>t1</value>
          <graphics/>
        </name>
      - <orientation>
          <value>0</value>
        </orientation>
      - <rate>
          <value>1.0</value>
        </rate>
      - <timed>
          <value>false</value>
        </timed>
      </transition>
    + <arc id="p1 to t1" target="t1" source="p1">
    - <arc id="p2 to t1" target="t1" source="p2">
        <graphics/>
      - <inscription>
          <value>1</value>
          <graphics/>
        </inscription>
        <arcpath id="000" y="295" x="105" curvePoint="false"/>
        <arcpath id="001" y="210" x="195" curvePoint="false"/>
      </arc>
    + <arc id="t1 to p3" target="p3" source="t1">
    </net>
  </pnml>
```

Figure 41-1: PNML document describing the simple Petri Net model

Automatically generated “msf.m”:

```
% GPenSIM Main Simulation File
% this MSF is generated from PNML file "samplePNML1.xml"
% MSF: 'msf.m'

clear all; clc;

global global_info; % global user data attached to global_info
global_info.PRINT_LOOP_NUMBER = 1;

pns = pnstruct('pdf_pdf');
dyn.m0 = {'P0',5, 'P2',3, 'P3',2};

pni = initialdynamics(pns, dyn);
sim = gpensim(pni);

prnss(sim);
```

Automatically generated “pdf_def.m”:

```
% GPenSIM PDF file generated from PNML file "samplePNML1.xml"
% PDF: 'pdf_pdf.m'

function [png] = pdf_pdf()

png.PN_name = 'PDFxxx';
png.set_of_Ps = {'P0','P1','P2',...
          'P3','P4'};
png.set_of_Ts = {'T0','T1','T2',...
          'T3'};
png.set_of_As = {'P0','T0',1, 'P1','T1',1, ...
       'P2','T0',1, 'P2','T2',3, 'P3','T2',1, ...
       'P4','T3',1, 'T0','P1',1, 'T1','P0',1, ...
       'T1','P2',1, 'T2','P4',1, 'T3','P2',3, ...
       'T3','P3',1};
```

Automatically generated “COMMON_PRE.m”:

```
% COMMON_PRE file generated from PNML file "samplePNML1.xml"
% 'COMMON_PRE.m'

function [fire, transition] = COMMON_PRE(transition)
%function [fire,transition] = COMMON_PRE(transition)

if (strcmpi(transition.name, 'T0')),
elseif (strcmpi(transition.name, 'T1')),
elseif (strcmpi(transition.name, 'T2')),
elseif (strcmpi(transition.name, 'T3')),
else
    % error (['Error in the transition name: ', transition.name]);
end;

% fire = 1; % let it fire
```

Automatically generated “COMMON_PRE.m”:

```
% COMMON_POST file generated from PNML file "samplePNML1.xml"
% 'COMMON_POST.m'

function [] = COMMON_POST(transition)
%function [] = COMMON_POST(transition)

if (strcmpi(transition.name, 'T0')),

elseif (strcmpi(transition.name, 'T1')),

elseif (strcmpi(transition.name, 'T2')),

elseif (strcmpi(transition.name, 'T3')),

else
    % error (['Error in the transition name: ', transition.name]);
end;
```

REFERENCES

[1]. Cassandras, C. G. and Lafortune, S. (2007) *Introduction to Discrete Event Systems.* Boston, MA: Springer Science+Business Media, LLC.

[2]. Davidrajuh, R. (2003) Event-driven simulation, modeling, and analysis with GPenSIM. *Communications of the IIMA*, Vol. 3, pp. 53-71, 2003

[3]. David, R. and Alla, H. (1994) Petri nets for modeling of dynamic systems – a survey. Automatica, 30 (2), pp. 175-202

[4]. Davidrajuh, R. (2007). A Service-Oriented Approach for Developing Adaptive Distribution Chain. *International Journal of Services and Standards*, Vol. 3, No.1, pp. 64 – 78.

[5]. Davidrajuh, R. and Molnar, I. (2009) Designing a new tool for modeling and simulation of discrete event systems. *Issues in Information Systems*, vol. X, pp. 472-477, 2009

[6]. Fuzzy Logic Toolbox (2010) http://www.mathworks.com/products/fuzzylogic/

[7]. User's guide for Fuzzy Logic Toolbox (2010) The Mathworks Inc., Berkeley, CA.

[8]. GPenSIM (2011) web page: http://www.davidrajuh.net/gpensim/

[9]. Hruz, B. and Zhou, M. (2007) *Modeling and Control of Discrete-event Dynamic Systems: with Petri Nets and Other Tools*. Springer.

[10]. Jensen, K. (1997) *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use*, 2. ed. Vol. 1: Springer, 1997

[11]. Moody. J. O. and Antsaklis, P. J. (1998) *Supervisory Control of Discrete Event Systems Using Petri Nets*, Kluwer Academic Publishers.

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

[13]. Petri, C. and Reisig, W. (2008) Petri net. *Scholarpedia*, vol. 3, p. 6477, 2008

[14]. Wilkinson, D. J. (2006) *Stochastic Modelling for Systems Biology*. Chapman & Hall/CRC, NY, 2006. ISBN-10 1-58488-540-8.

[15]. Stateflow (2010) Stateflow 7.4 - Design and simulate state machines and control logic. The MathWorks Inc, http://www.mathworks.com/products/stateflow/, 2010.

[16]. Stein, J. (2008) *How Math Explains the World*. NY: HarperCollins

[17]. Zhou, M. and Robbi, A. (1994). Application of Petri net methodology to manufacturing systems. *Computer Control of Flexible Manufacturing Systems: Research and Development* . Ed.: Joshi, S. and Smith, J. Chapman & Hall, Hong Hong.

[18]. Ciardo, G. (1987). Toward a Definition of Modeling Power for Stochastic Petri Net Models. Proceedings International Workshop on Petri Nets and Performance Models, 1987, pp. 54—62

[19]. RWTH- Mindstorms NXT Toolbox User Manual v4.04 (2010). List of functions.

[20]. RWTH–Mindstorms NXT Toolbox: Available: http://www.mindstorms.rwth-aachen.de

[21]. PIPE2 [xx]).

[22].

Appendix-1: List of functions

A

[EIP] = add_to_events_queue(NEIQ, EIP) {global PN;}

B

[COTREE] = build_cotree(X0) {global PN;}

C

[x1] = check_for_dominance(x, COTREE, parent)
[px] = check_valid_place(px)
[tx] = check_valid_transition(tx)
[r_index] = check_valid_resource(resource)
[delta_X,index_OP,inherited_color_set] = consume_tokens(transition, selected_tokID)
[cotree_results] = cotree(pns, m0) {global PN; PN = pns;}
[new_event] = create_new_event_in_Q(transition1, delta_X, output_place)
[current_clock_HMS] = current_clock(secs_or_HHMMSS)
current_time()

D

[] = deposit_token(placeI, nr_tokens, t_color)

DELTA_TIME (OPTION)

E

[enabled] = enabled_transition(t)
[TOKEN_MATRIX] = extractp(set_of_places)
[DURATION_MATRIX] = extractt(sim, {'t1'})

F

[LOG, colormap, EIP, no_of_completions] = firing_complete(LOG, colormap, EIP, FTS_index) {global PN;}
[log_record, colormap_record] = firing_complete_one(current_event, FTS_index) {global PN;}
[EIP] = firing_start(EIP) {global PN;}
[sfc, new_color,override,selected_tokens] = firing_preconditions(t1) {global PN;}
[sfc, new_color, override, selected_tokens] = firing_preconditions_COMMON_PRE(t1) {global PN;}
[sfc, new_color, override, selected_tokens] = firing_preconditions_specific_pre(t1) {global PN;}
[] = firing_postactions(fired_transition) {global PN;}

```
sim = firingseq(pni, firing_seq, repeat_seq, allow_parallel);
```

G

[colormap_record] = get_current_colors ()

p1 = get_place('pNOI');

[PN] = gpensim (pni)	**{global PN; PN = pns; global global_info;}**
[Pre_A, Post_A] = gpensim_2_PNCT()	{global PN;}
[] = gpensim_ver()	

I

[p_index] = is_place(place_name)	{global PN;}
[t_index] = is_trans(trans_name)	{global PN;}
[logic_value] = is_enabled(trans_name)	{global PN;}

L

LOOP_NUMBER (OPTION)

M

[markings_str] = markings_string(markings)
MAX_LOOP
V3 = mincyctime(pni);

N

[x1] = new_marking(t)	{global PN;}

O

```
[OCCUPANCY_MATRIX, DURATION_MATRIX] = occupancy(PN,
set_of_transitions)
```

OPTIONS
DELTA_TIME
LOOP_NUMBER
MAX_LOOP
STARTING_AT
STOP_AT
STOP_SIMULATION

P

[] = pack_sim_results(Enabled_Trans_SET, Firing_Trans_SET, LOG, colormap)
PI = pinvariant(pns);

`pnml2gpensim(PNMLFile);`

classtype = pnclass(pns);
[PN] = pnstruct(fileNames)
[] = plot_cotree(X0)
[TOKEN_MATRIX] = plotp(sim_results, set_of_places, plotCOLOR, plotLINEWIDTH)
[G,T] = PNCT_graph(Pre,Post,X0)
[] = PNCT_plottree(T)
[global_places] = pns_places(set_of_places)
[global_transitions] = pns_trans(set_of_trans)
[A] = pns_incidencematrix(global_set_of_As)
[] = print_cotree(COTREE);

```
priorinc
priordec
priorcomp
priorset
get_priority
```

[] = prnss (PN)
[] = prnss_enabled_trans(enabled_trans, checking_time)
[] = prnss_firing_trans(firing_trans, checking_time)
[] = prnss_state(fired_event, finishing_time, current_markings, state)
[] = prnsys (PN)

`[LE] = print_schedule(PN)`

[ordered_enabledTs] = priority_enabled_trans(enabledTrans)
[X] = process_marking(sources)

```
prnfinalcolors(sim_results, {'pEarly', 'pIntv_E', 'pIntv_L',
'pLate'});
prncolormap(sim_results, {'pEarly', 'pIntv_E', 'pIntv_L',
'pLate'});
```

R

[Output_Set] = randomgen(Input_Set)
[] = resource_assign(t_index) {global PN;}
[] = resource_unreserve(t_index) {global PN;}

```
[acquired] = request(transition.name);
[acquired] = request(transition.name, {resource_name1,
                   nr_of_instances1, …});
[acquired] = release(transition.name);
```

```
[string] = rt_clock_string()
```

S

[set_of_tokID] = select_token_time(placeI, nr_tokens_wanted, FCFS_or_LCFS)
[set_of_tokID, nr_token_av] = select_token_with_colors(place, nr_tokens_wanted, t_color)

[set_of_tokID, nr_token_av] = select_token_without_colors(placeI, nr_tokens_wanted, t_color)
[set_of_tokID, nr_token_av] = select_token_colorless(placeI,nr_tokens_wanted)
[] = set_ft(ftimes)
[] = set_ft_delta_T(min_ft_but_not_zero)
[] = set_initial_priority(initial_priority)
[] = set_initial_dynamics(dynamicpart)
[] = set_initial_resources(dynamicpart)
[] = set_options()
[element_nr] = search_names(name, Sys1)
[SIM_COMPLETE] = simulations_complete(Loop_Nr,MAX_LOOP)

SM = siphons_minimal(pns);
S = siphons(pns);

[time_string] = string_HH_MM_SS(timex)

STARTING_AT (OPTION)
STOP_AT (OPTION)
STOP_SIMULATION (OPTION)

T

[] = timed_pensim()

TI = tinvariant(pns);

[Ts,EIP,LOG,colormap,Enabled_Trans_SET, Firing_Trans_SET,SIM_COMPLETE,Loop_Nr,…
ETS_index,FTS_index] = **timed_pensim_init_all**()
[timer_increased] = **timer_increment**(number_of_completions)
[source_tokens] = **tokens_for_places**(sources)
[set_of_tokID, nr_token_av] = **tokenAny**(placeI, nr_tokens_wanted)
[set_of_tokID, nr_token_av] = **tokenAllColor** (placeI, nr_tokens_wanted, t_color)
[set_of_tokID, nr_token_av] = **tokenAnyColor**(placeI, nr_tokens_wanted, t_color)
[set_of_tokID, nr_token_av] = **tokenEXColor**(placeI, nr_tokens_wanted, t_color)
[set_of_tokID, nr_token_av] = **tokenWOAllColor**(placeI, nr_tokens_wanted, t_color)
[set_of_tokID, nr_token_av] = **tokenWOAnyColor**(placeI, nr_tokens_wanted, t_color)
[set_of_tokID, nr_token_av] = **tokenWOEXColor**(placeI, nr_tokens_wanted, t_color)
[set_of_tokID, nr_token_av] = **tokenColorless**(placeI, nr_tokens_wanted)

[set_of_tokID, nr_token_av] = **tokenArrivedEarly**(placeI, nr_tokens_wanted)
[set_of_tokID, nr_token_av] = **tokenArrivedLate**(placeI, nr_tokens_wanted)
[set_of_tokID, nr_token_av] = **tokenArrivedBetween**(placeI, nr_tokens_wanted)
[set_of_tokID] = **tokIDs**(placeI, nr_tokIDs_wanted)

[set_of_tokID, nr_token_av] = **tokenCheap**(placeI, nr_tokens_wanted)
[set_of_tokID, nr_token_av] = **tokenExpensive**(placeI, nr_tokens_wanted)
[set_of_tokID, nr_token_av] = **tokenCostBetween**(placeI, nr_tokens_wanted)

TM = traps_minimal(pns);
T = traps(pns);

?

[] = _pre_post_files()   {global PN;}

PN -> V convertion:

```
V1 = convert_PN_V(pni);
V = cycles(V1);
print_minimum_cycle_time(V);
```