Addendum-2 to the book **Davidrajuh, R. "Modeling Discrete-Event Systems with GPenSIM: An Introduction", Springer, 2018**

Using GPenSIM Resources

By Reggie Davidrajuh, reggie.davidrajuh@uis.no, © 16 June 2018.

In engineering systems, there are always resources. For example, in a manufacturing system, humans are needed to operate some machines and robots for some activities; printers are common resources in a computer Local Area Network (LAN). In a P/T Petri Net, resources are usually represented by places, with some initial token equaling to the available instances of the resource. 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 a print function called '**prnschedule**' to print details of the resource usage.

1. Resources

There are different issues involving resources:

- *Instances* of a resource: This means, a resource has many indistinguishable copies ('instances'). For example, in a bank, there are three cashiers, all of them process all kind of transactions. For a casual customer, it doesn't make sense to prefer one cashier over the others. In this case, we could generalize all the cashiers into a group, name this resource as 'cashier,' and say that cashier is a single resource with three instances.
- *Generic* resources: this means, there are some 'named' resources available, but all of them are same for some generic applications. For example, for a mathematical modeling project, we have three modellers called 'Ada,' 'Richelle,' and 'Nathania.' Though they are the specialist in some areas of mathematical modeling, when it comes to basic modeling, they all are the same. Thus for a simple mathematical modeling work, we can pick any (generic) modeller, without naming anyone.
- *Specific* resources: When the resources are named, for some applications, we may prefer specific named resources. For example, for a mathematical modeling project, 'Ada' is a specialist in discrete mathematics, 'Richelle' is an expert in signal processing, and 'Nathania' is in HCI. Hence, when we need to specialist to work with HCI, we prefer using the specific resource 'Nathania.' Only if Nathania is not available, we may go for another resource.
- *Write access*: a resource may contain many instances (e.g., cashier resource with three instances), and a transition may try to acquire one or more of these instances. Write access means, the resource will be locked, and **all the instances** will be made available to a requesting (single) transition.

1.1 Declaring Resources

The resources are to be declared in the MSF. For example, if a LAN network has five identical printer resources (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, represented by the infinity 'inf' symbol. In MSF:

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

Let us declaring three mechanics (one instance each) named 'Al,' 'Bob,' and 'Chuck.' Let us also assume that the mechanics can work up to 5, 8, and 10 hours, respectively in a day (also known as **cycle time** in scheduling algorithms). In MSF:

```
dynamicpart.re = {'Al',1,5, 'Bob',1,8, 'Chuck',1,10};
```

1.2 The functions for resource usage

There are some functions available in GPenSIM for resource management, such as checking the availability of resources and reserving them; these functions are called in COMMON_PRE or specific _pre files. Whereas, releasing the resources are done in the COMMON_POST or specific _post files.

Table 1-1: GPenSIM functions for resource management.

Function	*Description*
`availableInst`	Checking whether any instances of a resource are available.
`availableRes`	Checking whether any resources are available in the system.
`requestSR`	Request some instances from specific ('named') resources.
`requestGR`	Request some resource instances, without naming any resource.
`requestAR`	Request some resource instances among many alternative resources.
`requestWR`	Request all the instances of a specific resource (exclusive access): this means, if the resource has many instances, either all the instances are granted (if available), or none is granted.
`release`	Release all the resources and its resource instances held by a transition.
`prnschedule`	Prints useful information on resources usage.
`plotGC`	Plots Gantt Chart, showing how the resources were used (by which transition and how long).
`occupancy`	Prints summary of resource usage.

Detailed usage of the functions:

Function 'availableInst':

This function returns info about the available (free) instances of a given resource. Input parameter to this function is the name of the resource (or the index of the resource). The output parameter of this function is a structure consisting of the following fields:

- avINS.r_index: index of the resource.
- avINS.n: the number of free (available) instances of the resource.
- avINS.instance_indices: the set of indices of the available instances.

Usage: in COMMON_PRE or specific _pre:

```
avInst = availableInst('printer');
```

CAUTION: It is important to note the following: when an instance is checked for availability, GPenSIM will not verify whether the time this instance has been used so far added with the firing time of the

requesting transition will be within the cycle time (or 'max cap') of the instance. This verification is not possible because of the stochastic firing times that can be anything during run-time.

Function 'availableRes':

This function returns info about all the available resources. As with the function `availableInst`, the instances are checked for not exceeding their cycle times, if they were to be allocated to this requesting transition. The input parameter is the set of resource names (or resource indices); if the input parameter is not given, then all the resources will be checked for available instances. The output parameter is an **array of structures** (called resource availability info 'RAI'), where each structure has the following elements:

- avINS.r_index: resource index.
- avINS.n: the number of free (available) instances of this resource.
- avINS.instance_indices: the set of indices of the available instances

Usage: in COMMON_PRE or specific _pre:

```
avRes = availableRes({'printer','CPU'}); % check available prn, cpu
avRes = availableRes();  % check all available resources
```

Function 'requestGR':

Firstly, this function checks whether the required number of free instances can be composed of all the resources. If this is possible, then these free instances will be marked reserved for the requesting transition. On the other hand, if the requested number of free instances cannot be composed, then none of the free ones are reserved ("all the required or none"). The input parameter of this function is the required number of instances. The output parameter is a Boolean indicating the reservation was successful or not.

Usage: in COMMON_PRE or specific _pre:

```
Status_of_reservation = requestGR(2); % request ANY 2 instances
```

Function 'requestSR':

This function reserves a set of specific resource instances. Only if all the required number of instances of the specific resource are found free, then all of them will be reserved for the requesting transition; otherwise, none will be reserved ("all the required instances or none"). The input parameter is a set of resource instances. The output parameter is Boolean status info indicating whether the reservation was successful or not.

Usage: in COMMON_PRE or specific _pre:

```
% request 2 cashiers and one manager
sts = requestSR({'cashier',2,'mgr',1});
```

Function 'requestAR':

This function reserves some instances from any of the specified resources. As with the other two request functions, "all or none" policy applies. The input parameters are 1) a pool of resources, and 2) the number of instances required.

E.g., A car rental company wants to check whether two cars are available from the German-made cars, and three from the Japanese made cars. Usage: in COMMON_PRE or specific _pre:

```
r1 = requestAR({'Benz','BMW','Audi'}, 2); % request any two German
r2 = requestAR({'Toyota','Honda','Mazda'}, 3);% any 3 Japanese cars

if and(r1, r2),
     % yes, 2 German and 3 Japanese cars are free
…
```

Function 'requestWR':

This function tries to reserve **all the instances of a specific resource**; "all or none" policy applies. The input parameter is the name (or index) of the resource. Note that, only one resource can be assigned as the input parameter.

E.g., Let us assume that a production facility has five identical robots and three identical milling machines. Thus, in MSF, the declaration of these resources:

```
dyn.re = {'robot',5,'milling M/C',3}; % systems resources
```

After a long production run, *all* the robots and *all* the milling machines must be stopped for maintenance; thus, in COMMON_PRE or specific _pre:

```
rAR = requestWR('robot'); % request all robots
rAM = requestWR('milling M/C'); % request all milling M/Cs

if and(rAR, rAM),
     % yes, all the robots and all the milling M/Cs are acquired
     % Start maintenance cycle of robots & milling M/Cs
…
```

Note-1: Request for resources and resource instances by a transition can only happen in COMMON_PRE or the transition's specific _pre.

Note-2: A transition may request resources in its specific 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 (reserved) resources will be taken away from the transition.

1.3 Releasing the Resources (after usage)

After firing, a transition may release ***all the resources*** it used (held). After firing, **releasing only some of the resources (not all of them) is not possible**. Releasing the resources is done in the post-processor files (specific post or COMMON_POST).

The function '**release**' behaves very differently depending on whether an input parameter was provided or not.

No input parameter: In this case (the usual case), the transition that has just fired releases all the resources it was holding and using. E.g., in COMMON_POST or the specific_post file:

```
% release all the resources held by the transition just fired
release();
```

The name of a transition as the input parameter: In this case (special case), the transition just fired, **is not releasing the resources held by it**; the system is only releasing the resources used by some other transition indicated as the input parameter.

E.g., **t1** is the transition that has just finished firing. Thus, the system is executing the post-processor file **t1_post**. In the post processor file, the system is instructed to release all the resources acquired by another transition **t99** and not the resources that may have been obtained by **t1**.

```
function [] = t1_post(transition)
% this is the post-processor file for t1

% release all the resources (if any) used by another transition t99
release('t99');
```

1.4 Functions for printing and plotting resource usage

Function '**prnschedule**' is exclusively used when resources are involved. After simulations, this function prints useful information, such as how much (how long) each of the resources and their instances was used, and by whom (which transitions). This function is useful for scheduling applications. Function '**plotGC**' plots the Gantt chart showing the resource usage graphically, which transitions were using each resource and for how long. Function '**occupancy**' prints summary of the resource usage.

1.4.1 Example-50: Using Resources to realize the critical section

This example is to force two transitions to fire alternatingly. Though it is possible to realize alternating firing with a binary semaphore, we make use of 'resources' in this example.

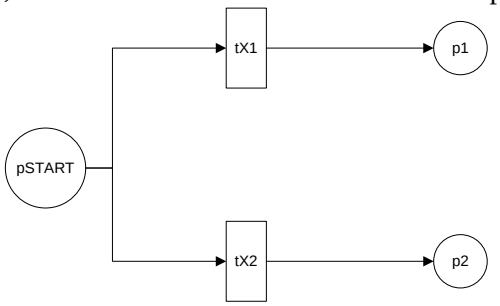

Figure 1-1: Alternating firing, achieved by using resources.

In figure-1-1, transitions **tX1** and **tX2** are supposed to fire alternatingly. To make this happen, the two transitions seek a common *resource*. The common resource behaves like a critical region such that only one transition can 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-50: Resource as Critical Region
function [png] = critical_region_pdf()

png.PN_name = 'Example-50: Resource as a 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-50: 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'});
plotGC(sim); % plot the Gantt Chart
```

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

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

granted = requestSR({'Resource-X',1}); %alternatively:granted=requestGR(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 resources used by the fired transition will be released.

```
function [] = COMMON_POST(transition)
% release all resources used by transition
release(); %
```

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

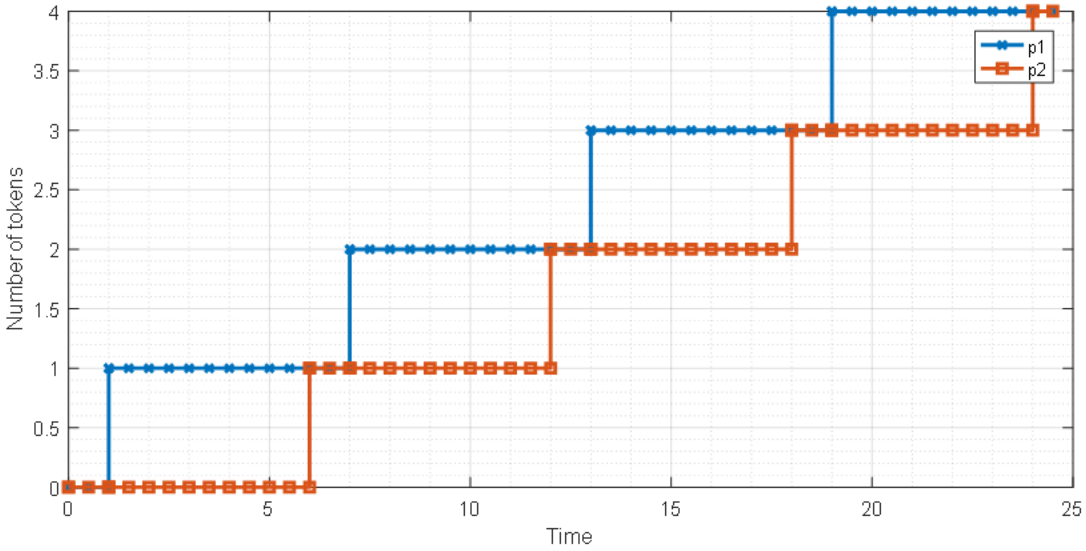

Figure 1-2: Tokens in p1 and p2 indicate alternating firing

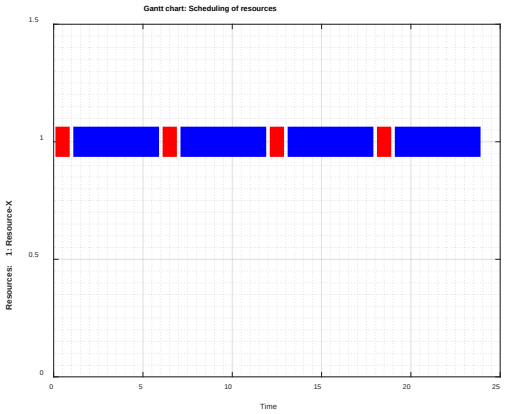

Figure 1-3: Gantt Chart also shows alternating firing.

The Gantt chart shows that the resource Resource-X was alternatingly used by the transitions **tX1** (represented by the red colored short bars of 1 TU length) and **tX2** (represented by blue colored the long bars of 5 TU length). 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%
```

1.4.2 Example-51: Using Specific Resources

This example shows how GPenSIM handles resources. This example also shows that by using resources, the size of the Petri Net model can be minimized.

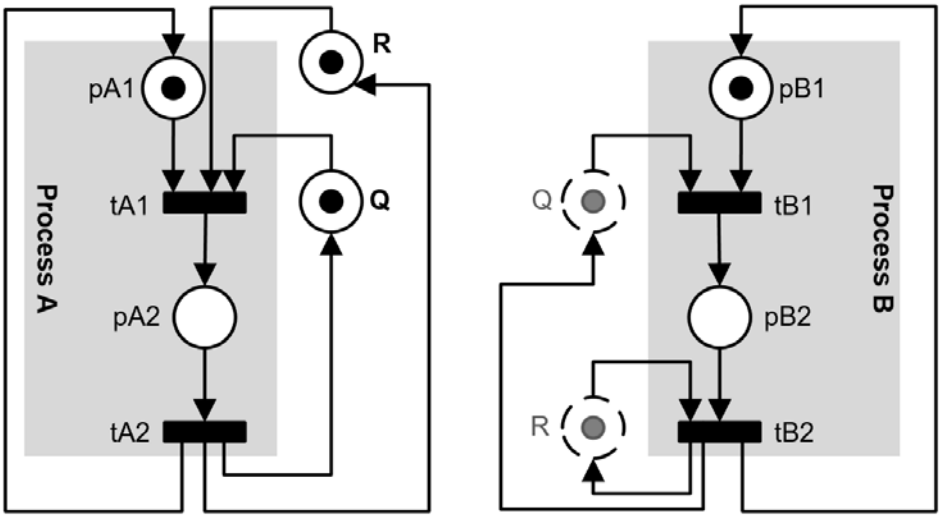

Figure 1-4: Two processes A and B using two resources Q and R

Figure-1-4 given above shows a system with two processes (**A** and **B**) that share two different resources (**Q** and **R**). The details:

- Resources **Q** and **R** have one instance each.
- The process **A** has two steps **A1** and **A2**. Step-**A1** needs both of the resources to start. When **A1** is completed, **A1** will not release the resources as they are needed for the step-**A2**. When **A2** is completed, both of the resources will be released.
- The process **B** has two steps too, **B1** and **B2**. Step-**B1** needs the resource **Q** to start. When **B1** is completed, **Q** will not be released as it is needed for step-**B2**. However, **B2** needs **R** too. When **B2** is completed, **Q** and **R** will be released together.
- Processes **A** and **B** run cyclically.

The standard P/T Petri Net model shown in figure-1-4. This model reveals that the resources are represented by tokens inside the places **Q** and **R**. This means, allocation and retrieval of resources are shown by the arcs connecting transitions (like **tA1**, **tB2**, etc.) to the places **Q** and **R**. Shown below in the figure-1-5 is the GPenSIM version of the same system.

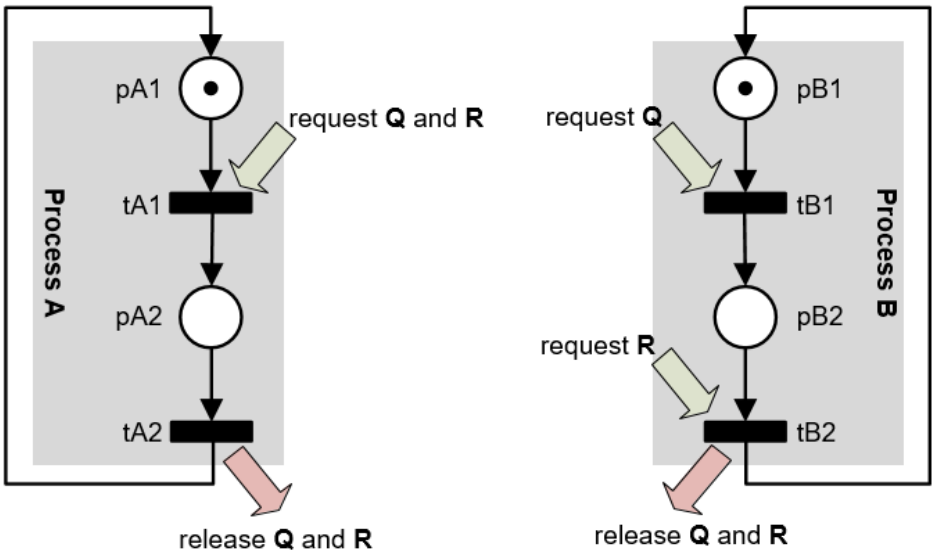

Figure 1-5: Simplified model, GPenSIM style.

In the GPenSIM version shown in figure-1-5, the resources **Q** and **R** are treated as variables and hence will be not be shown in the Petri Net model. Thus, there will be no arcs from the transitions requesting resources to the places that represent the resources. Lack of these arcs makes the GPenSIM version is much simpler.

PDF:

```
% Example-51: 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};
```

MSF:

```
% Example-51: Two processes A and B using two resources Q and R
global global_info
global_info.STOP_AT = 20;

pns = pnstruct('procAB_resQR_pdf');

dyn.m0 = {'pA1',1, 'pB1',1};
dyn.ft = {'allothers',1};
dyn.re = {'Q',1,inf, 'R',1,inf};

pni = initialdynamics(pns, dyn);

sim = gpensim(pni);

prnschedule(sim);
plotGC(sim, [1 1 2 2]);
```

```
occupancy(sim, {'tA1','tA2', 'tB1','tB2'});
```

COMMON_PRE:

```
% Example-51: Two processes A and B using two resources Q and R
function [fire, transition] = COMMON_PRE(transition)
switch transition.name
    case 'tA1'
        granted = requestSR({'Q',1, 'R',1});
    case 'tA2'
        granted = 1;          % do nothing

    case 'tB1'
        granted = requestSR({'Q',1});
    case 'tB2'
        granted = requestSR({'R', 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 the resource that acquired by itself, as there aren't any resources acquired by it. Similarly, after firing, **tB2** must release resources acquired by both **tB1** and **tB2**.

```
function [] = COMMON_POST(transition)
switch transition.name
    case 'tA1'
        % do nothing
    case 'tA2'
        release('tA1'); % release all the resouces acquired by tA1

    case 'tB1'
        % do nothing
    case 'tB2'
        release('tB1'); % release resources acquired by tB1
        release('tB2'); %  and by tB2
    otherwise
end
```

Simulation result:

```
RESOURCE USAGE:

RESOURCE INSTANCES OF ***** Q *****
tB1 [0 : 2]
tB1 [2 : 4]
tA1 [4 : 6]
tA1 [6 : 8]
tA1 [8 : 10]
tA1 [10 : 12]
tA1 [12 : 14]
tA1 [14 : 16]
tB1 [16 : 18]
Resource Instance: Q:: Used 9 times. Utilization time: 18

RESOURCE INSTANCES OF ***** R *****
tB2 [1 : 2]
```

```
tB2 [3 : 4]
tA1 [4 : 6]
tA1 [6 : 8]
tA1 [8 : 10]
tA1 [10 : 12]
tA1 [12 : 14]
tA1 [14 : 16]
tB2 [17 : 18]
Resource Instance: R:: Used 9 times. Utilization time: 15

RESOURCE USAGE SUMMARY:
Q:  Total occasions: 9   Total Time spent: 18
R:  Total occasions: 9   Total Time spent: 15

*****  LINE EFFICIENCY AND COST CALCULATIONS: *****
  Number of servers:  k = 2
  Total number of server instances:  K = 2
  Completion = 20
  LT = 40
  Total time at Stations: 33
  LE = 82.5 %
  **
  Sum resource usage costs: 0   (NaN% of total)
  Sum firing costs: 0   (NaN% of total)
  Total costs:  0
  **

Occupancy analysis ....
Simulation Completion Time: 20
occupancy tA1:
  total time: 6
  Percentage time: 30%
occupancy tA2:
  total time: 6
  Percentage time: 30%
occupancy tB1:
  total time: 4
  Percentage time: 20%
occupancy tB2:
  total time: 3
  Percentage time: 15%
>>
```

1.5 Scheduling using Resources

Scheduling is 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 with GPenSIM.

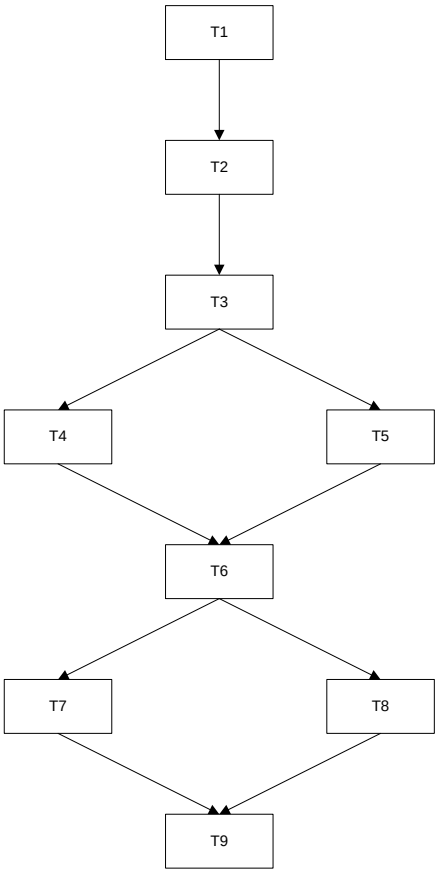

Figure 1-6: Scheduling of a software developement project

1.5.1 Example-52: Scheduling of a Software Development Project

A team of software developers is 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-1-6 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-1-2 given below shows the stages in the software development, probable time (in months) each stage will take and the resources demanded by each stage (task).

Table 1-2: 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**

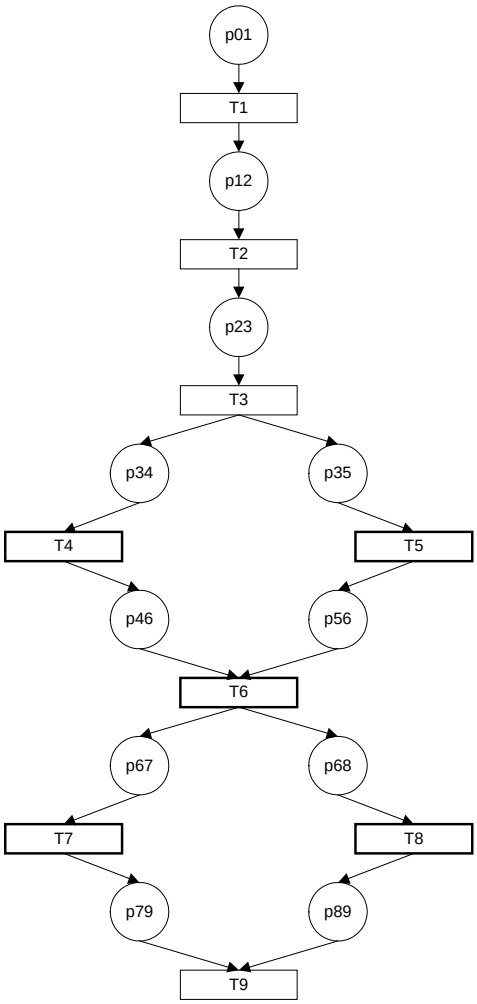

Figure 1-7: Partial Petri Net model of the SW development project

Petri Net model: The Petri Net model of the project can be easily built from the digraph shown in figure-1-6. First of all, we represent each task (square box) in the digraph by a transition. And then, between any two transitions, we inject a place as a buffer. Also, just to start the system, we put a place with an initial token at the top above **T1**. Figure-1-7 shows the ***partial*** Petri Net model.

To make a complete Petri Net model, we have to add the resource requirements to each transition. For example, for Stage-1 (task **T1**) to start, there must be at least two developers (resources) available. This means, there must be an arc with weight two, from the place representing the resource pool (**pRP**) to the transition representing the task **T1**. Figure-1-8 below shows the complete Petri Net model.

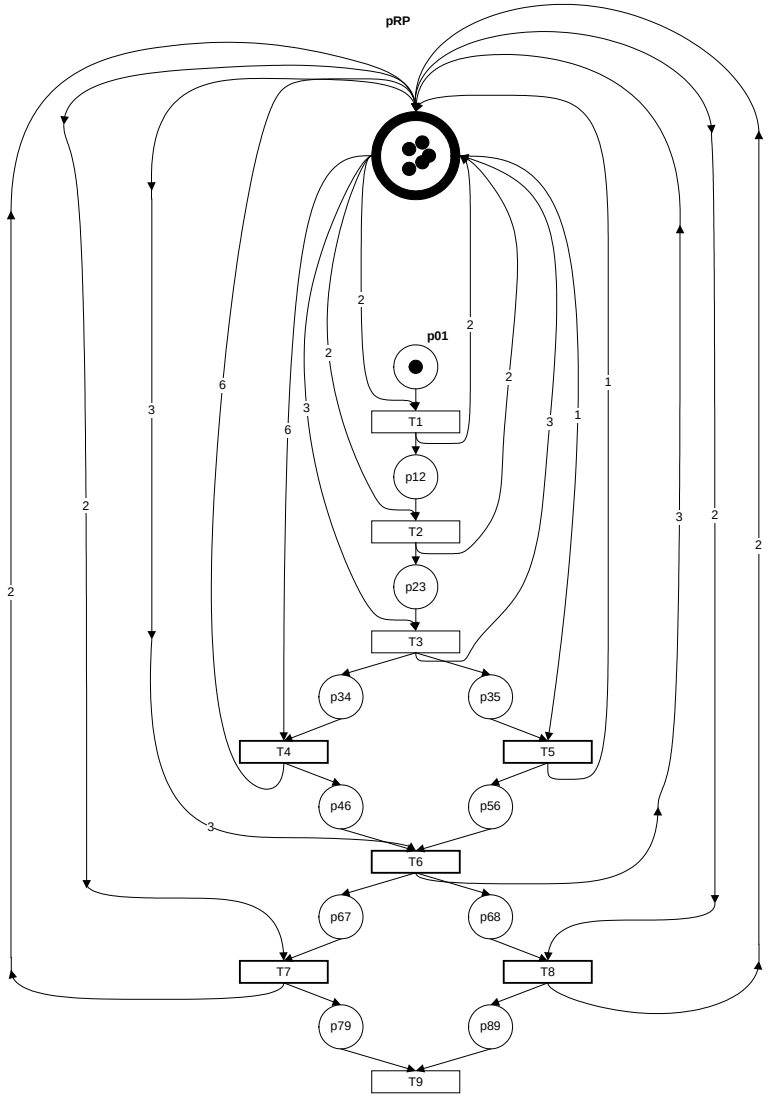

Figure 1-8: (Complete) Petri Net model for software development project

The Petri Net model shown in figure-1-8 is unnecessarily complex, and it is rudimentary. The model is complex because of the resource requirements imposed by the arcs from place **pRP** to all the transitions. The model is also rudimentary 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 represented by 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 remove resources altogether from the Petri Net model: let us remove the place **pRP** (which represent the developers) from the model, and push it into the background as a 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 figure-1-7 as the partial Petri Net model. However, this model shown in figure-1-7 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-1-7).

```
% Example-52: Scheduling with Resources
function [png] = schedule_w_resources_pdf()
png.PN_name   = 'example-52: schedule w resources';
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-52: 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};% 9 developers with infinite cycle time

pni = initialdynamics(pns, dp);

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

COMMON_PRE: all the transitions request resources through COMMON_PRE.

```
% Example-52: Scheduling with rescources
function [fire, trans] = COMMON_PRE(trans)
global global_info

switch trans.name
    case {'T1', 'T2', 'T7', 'T8'}
        granted = requestGR(2); % request 2 instances of resource

    case {'T3', 'T6'}
        granted = requestGR(3); % request 3 instances of resource

    case 'T4'
        granted = requestGR(6); % request 6 instances of resource

    case 'T5'
        granted = requestGR(1); % request 1 instance of resource

    otherwise % 'T9'
    global_info.STOP_SIMULATION = 1;  % Time to 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 result:

```
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 (workforce) are not utilized fully; there are too many developers, as developer-8 and developer-9 are not utilized at all. Also, developers-5, 6, and 7 are used only in one stage and developer-4 only in two stages. Thus, resource (workforce) utilization is only 45%.

1.5.2 Example-53: Scheduling with Specific Resources

For a realistic software development project, table-1-2 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-1-3 instead, as table-1-3 is more advanced as it shows the specific resource requirements for each stage.

Let us assume that the project team consists of nine members as before: a project manager, a chief developer, two senior developers, three developers and two trainees. The table-1-3 given below shows specific resources needed for each stage, otherwise same as the table-1-2.

Table 1-3: The specific resources needed in each stage

	Stage (or task)	**Time (months)**	**Resources Needed**
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, even though the digraph permits parallel execution of these two stages.

PDF: (same as before, since the resource is not part of the static Petri Net static structure)

MSF: 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:

```
% Example-53: Scheduling using Specific Resources
function [fire, trans] = COMMON_PRE(trans)
global global_info

switch trans.name
    case {'T1', 'T2'}
        granted = requestSR({'project manager',1, ...
            'chief developer',1});

    case 'T3'
        granted = requestSR({'project manager',1, ...
            'chief developer',1, 'senior developers',1});
```

```
    case 'T4', granted = requestGR(6); % T4 seeks any 6 resources
    case 'T5', granted = requestGR(1); % T5 seeks any 1 resource
    case 'T6', granted = requestGR(3); % T6 seeks any 3 resources

    case {'T7', 'T8'}
        % T7 and T8 seek 'chief developer' and any 1 resources
        granted1 = requestSR({'chief developer',1});
        granted2 = requestGR(1);
        granted = and(granted1, granted2);

    otherwise % 'T9'
    global_info.STOP_SIMULATION = 1; % stop simulations
    granted = 1;
end
fire = granted;
```

COMMON_POST: same as before

Simulation result:

```
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).

1.5.3 Example-54: Project Completion Time

The following example is taken from **James D. Stein (2008) "How Math Explains the World", page-3.** Figure-1-9 is a digraph showing the order of the tasks (and the time taken by the tasks) to be done to complete the job.

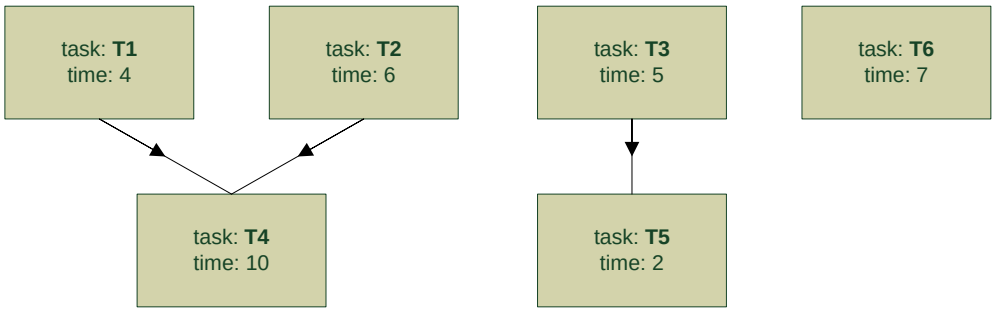

Figure 1-9: Digraph showing order of tasks to be completed.

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:

The Petri Net model (figure-1-10) is easily obtained from the digraph, by substituting transitions for the tasks and injecting buffer places between the transitions.

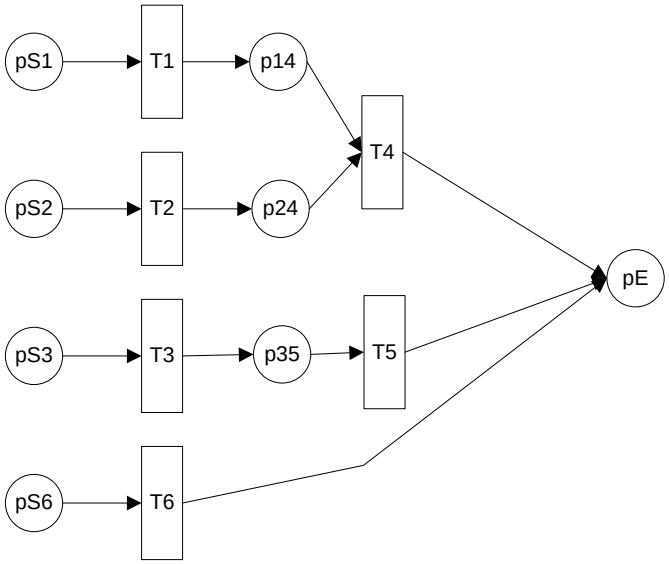

Figure 1-10: Petri Net model

PDF:

```
% Example-54: 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-54: Project Completion Time
% Problem taken from James D: Stein, "How Math Explains the World", page.3
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 = requestGR(1); % 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**.

```
% Example-54: Project Completion Time
function [] = COMMON_POST(transition)
global global_info

release();

% check if the three transitions T4, T5, and T6 has already fired.
% if so, stop the simulations, immediately.
if ismember(transition.name, {'T4', 'T5', 'T6'})
    n4 = timesfired('T4'); % how many times T4 has fired
    n5 = timesfired('T5'); % how many times T5 has fired
    n6 = timesfired('T6'); % how many times T6 has fired

    if n4*n5*n6   % T4, T5, and T6 has fired at least once
        global_info.STOP_SIMULATION  = 1;
    end
end
```

Simulation result: 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 %
 **
```

'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%

2. Acitivity-Oriented Petri Net (AOPN)

GPenSIM facilitates compact Petri Net models by removing the resources (usually represented as places) from the Petri Nets and calling them as variables during run-time. This approach is useful for modeling systems that employ a large number of resources (e.g., manufacturing systems). Usually, even for a system with few activities competing for a few resources, the resulting Petri Net model can be huge (Davidrajuh 2012, Skolud et al. 2016).

Activity-oriented Petri Nets (AOPN) is an approach for obtaining compact Petri Net models of discrete-event systems where resource sharing and resource scheduling dominate (Davidrajuh, 2018). GPenSIM is a realization of AOPN on the MATLAB platform, supporting run-time resource management. In other words, AOPN is integrated into GPenSIM.

2.1 The Two Phases of the AOPN Approach

AOPN is a two-phase modeling approach (Davidrajuh, 2012; Skolud et al., 2016):

- In the Phase-I, the static Petri Net graph is created, focusing mainly on the activities. The activities are represented by transitions in the static Petri Net graph, and the transitions are separated from each other by the buffering places. The resources are grouped into two groups such as 'focal' resources and 'utility' resources. In addition to the activities, the focal resources are too included in the static Petri Net. In this case, focal resources are represented by places. The utility resources will not be considered in the phase-I. Thus, a compact Petri Net model is obtained in phase-I, with only the transitions representing the activities and, if there are any focal resources, they will be represented by places. In GPenSIM, coding the static Petri Net in phase-I will result in the Petri Net definition file (PDF).
- In the phase-II, the run-time dynamic model is developed. In the phase-II, the dynamic details that are not considered in the phase-I are added to the Petri Net model. E.g., transitions (activities) requesting, using, and releasing of the utility resources are coded in the run-time model. In GPenSIM, the run-time details in the phase-II will result in the files COMMON_PRE and COMMON_POST (and perhaps, specific pre- and post files too). The initial dynamics (initial markings of the places and the firing times of the transitions) are also added in the second phase; these details of initial dynamics goes into the Main Simulation File (MSF).

In the next section (section 2.2), we model a flexible manufacturing system with Timed Petri Net. In the subsequent section (section 2.3), we model the same flexible manufacturing system with the AOPN approach, just to show the advantages of this approach.

2.2 Example: Modeling an FMS with the AOPN approach

A simple Flexible Manufacturing System (FMS) is shown in figure-2-1. This FMS is to make only one type of product. The FMS consists of two conveyor belts (C1 and C2), three industrial robots (R1, R2, and R3), two CNC machines (M1 and M2), an assembly station (AS), one painting station (PS), and a painting robot (R4). The operational specifications of the FMS:

- The input raw material of type-1 arrives on the conveyor belt C1. Similarly, the input raw material of type-2 arrives on the conveyor belt C2. C1 and C2 transport only a unit of input material at a time.
- Robot R1 picks the raw material type-1 and places into the machine M1. Similarly, robot R2 picks the raw material from conveyor belt C2 and places into the machine M2.
- Machine M1 makes the part P1, and M2 makes the part P2. As soon as the machines M1 and M2 make the parts, they are placed on the assembly station by the robots R1 and R2, respectively.
- The assembly station AS to join the two parts P1 and P2 together to form the product. The robot R2 does the part assembly at AS.

- Robot R3 picks the product from the assembly station and places it on the painting (and polishing) station PS.
- Painting robot R4 performs the painting.
- Once the painting is completed, robot R3 the put the completed product into the output buffer (cartridge) OB.

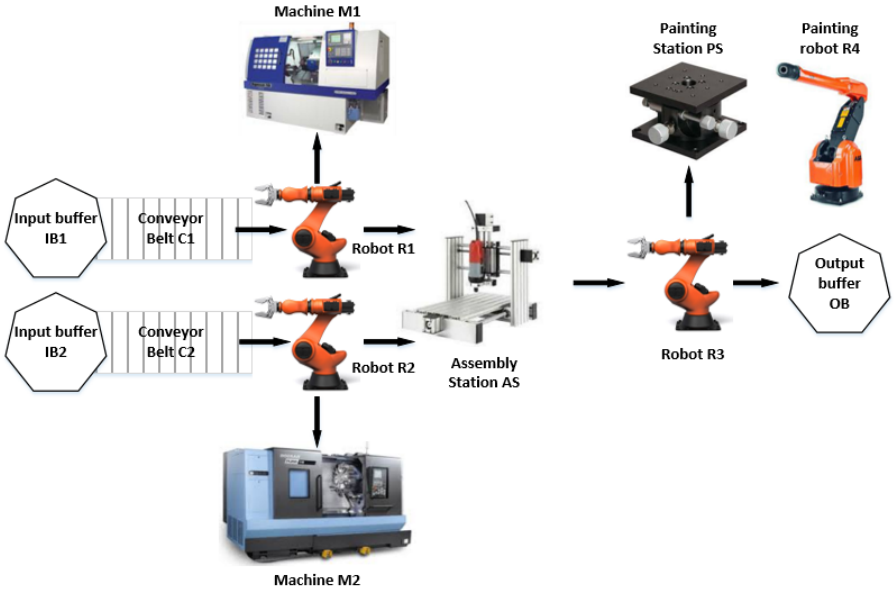

Figure 2-1: The Flexible Manufacturing System

The following activities explain the FMS operations (the firing time is given in TU):

- **tC1** (2): conveyor belt C1 brings the input material type-1 into the FMS, one item at a time.
- **tC2** (2): conveyor belt C2 brings the input material type-2 into the FMS, one item at a time.
- **tC1M1** (2): robot R1 moves raw material from conveyor belt C1 to M1.
- **tC2M2** (2): robot R2 moves raw material from conveyor belt C2 to M2.
- **tM1** (20): machining of Part-1 at machine M1.
- **tM2** (35): machining of Part-2 at machine M2.
- **tM1AS** (2): robot R1 moves part P1 from machine M1 to assembly station AS.
- **tM2AS** (2): robot R2 moves part P2 from machine M2 to assembly station AS.
- **tAS** (8): robot R2 assembles parts P1 and P2 together at the assembly station AS.
- **tAP** (2): robot R3 picks the product from the assembly station and places on the painting station PS.
- **tPS** (13): robot R4 performs painting and surface polishing on the product. When the painting job is completed, robot R3 places the product into the output buffer OB.

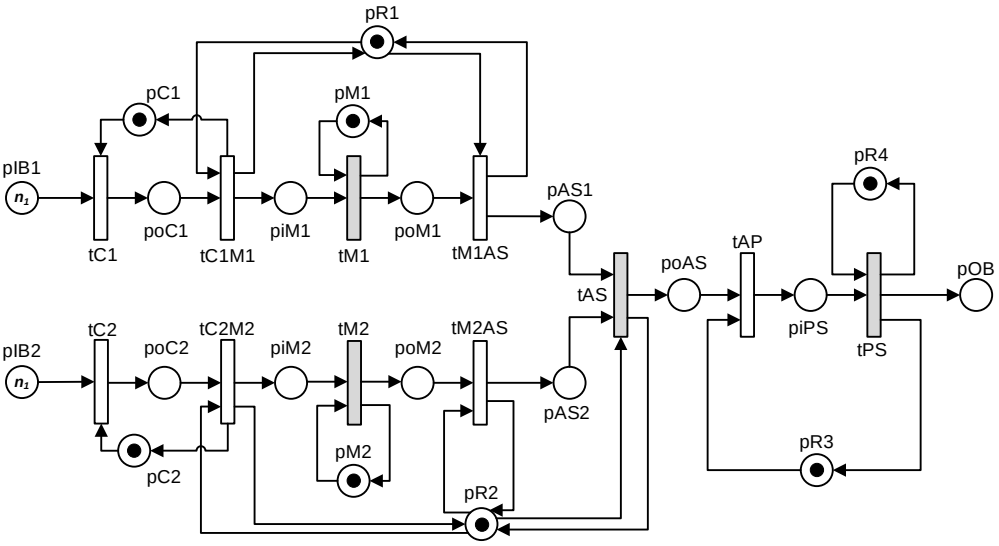

Figure 2-2: Petri Net model of the FMS

2.2.1 Example-55: The Timed Petri Net model

The Timed Petri Net model of the FMS is shown in figure-2-2. The Petri Net model is obtained by combining all the operations listed above.

Figure-2-2 clearly shows that even for a small FMS with a few manufacturing resources, the Petri Net model becomes complex. As real-world FMS usually has a large number of manufacturing resources, we can expect that modeling these FMS will result in huge Petri Net model and that simulation and analysis of these Petri Nets will not be easy. Thus, we will reduce the size of these Petri Nets with the AOPN approach.

MSF:

```
% Example-55: Timed Petri Net model of a Flex. Manufacturing System
global global_info
global_info.STOP_AT = 100;

pns = pnstruct('fms_timedPN_pdf');

dp.m0 = {'pIB1',1,'pIB2',1, 'pC1',1,'pC2',1,...
         'pR1',1,'pR2',1,'pR3',1,'pR4',1, 'pM1',1,'pM2',1};
dp.ft = {'tM1',20,'tM2',35,'tAS',8,'tPS',13, 'allothers',2};
pni = initialdynamics(pns, dp);

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

COMMON_PRE: There is no need for COMMON_PRE

COMMON_POST: (just to print the names of the completing transitions)

```
function [] = COMMON_POST(transition)
disp(['At time=', num2str(current_time), ', transition ', ...
      transition.name, ' has just fired.']);
```

Simulation result:

```
At time=2, transition tC2 has just fired.
At time=2, transition tC1 has just fired.
```

```
At time=4, transition tC2M2 has just fired.
At time=4, transition tC1M1 has just fired.
At time=24, transition tM1 has just fired.
At time=26, transition tM1AS has just fired.
At time=39, transition tM2 has just fired.
At time=41, transition tM2AS has just fired.
At time=49, transition tAS has just fired.
At time=51, transition tAP has just fired.
At time=64, transition tPS has just fired.
>>
```

2.2.2 Example-56: Modeling an FMS with the AOPN approach

In the Petri Net model shown in figure-2-2, let us assume the following:

- All the manufacturing resources are utility resources, thus need not be shown in the Petri Net model.

This means, by the AOPN approach, the places (**pC1**, **pC2**, **pM1**, **pM2**, **pR1**, **pR2**, **pR3**, and **pR4**) that represent these resources will be removed from the static Petri Net. Also, the arcs that connect these places to the transitions will also be eliminated from the Petri Net model, resulting in a simple and skinnier model. The resulting static Petri Net model is shown in the figure-2-3.

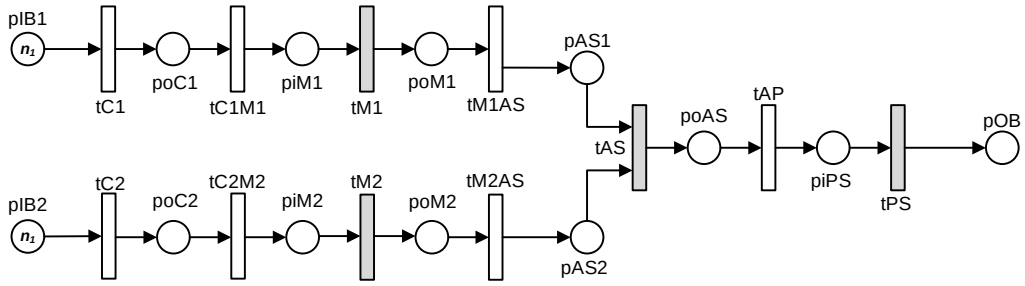

Figure 2-3: Skinnier static Petri Net by the AOPN approach.

During run-time, **tC1** will request the resource C1 to start transporting the input material type-1. The resource C1 will be released upon completion of **tC1M1**. **tC1M1** will request R1 to start the movement of input material type-1 from input conveyor belt to M1. Once **tC1M1** is complete R1 will be released. Figure-2-4 shows all the resource requests (inward arrows in red color) and releases (outward arrows in green color).

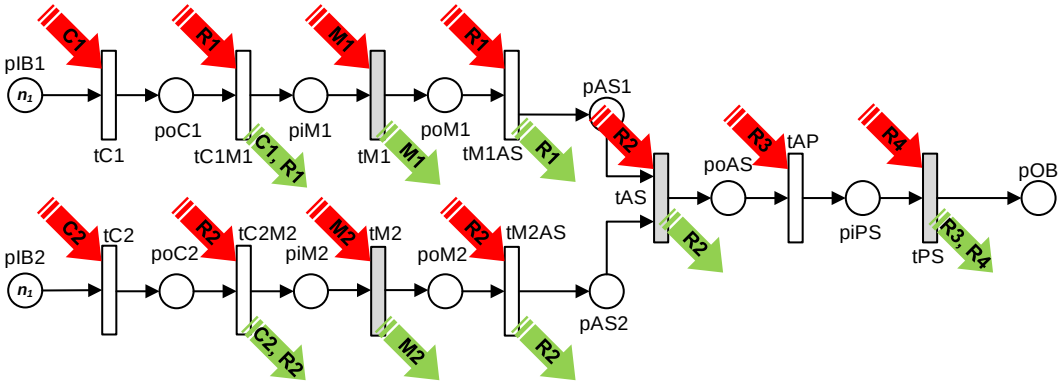

Figure 2-4: Petri Net model by the AOPN approach.

The following GPenSIM code (especially the COMMON_PRE) shows how elegantly the problem can be simulated and analyzed.

MSF:

```
% Example-56: AOPN model of a Flexible Manufacturing System
global global_info
global_info.STOP_AT = 300;

pns = pnstruct('fms_AOPN_pdf');

dp.m0 = {'pIB1',3,'pIB2',3};
dp.ft = {'tM1',20,'tM2',35,'tAS',8,'tPS',13, 'allothers',2};
dp.re = {'C1',1,inf,'C2',1,inf, 'M1',1,inf,'M2',1,inf, ...
         'R1',1,inf,'R2',1,inf,'R3',1,inf,'R4',1,inf};
pni = initialdynamics(pns, dp);

sim = gpensim(pni);
plotp(sim, {'pOB'});
prnschedule(sim);
```

COMMON_PRE: In this file, the enabled transition requests the necessary resources to start firing:

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

switch transition.name
    case 'tC1'
        granted = requestSR({'C1',1}); % conveyor belt "C1" required
    case 'tC2'
        granted = requestSR({'C2',1}); % conveyor belt "C2" required
    case {'tC1M1', 'tM1AS'}          % robot "R1" required
        granted = requestSR({'R1',1});
    case {'tC2M2', 'tM2AS', 'tAS'}   % robot "R2" required
        granted = requestSR({'R2',1});
    case 'tM1'    % machine "M1" required
        granted = requestSR({'M1',1});
    case 'tM2'    % machine "M2" required
        granted = requestSR({'M2',1});
    case 'tAP'    % robot "R3" required
        granted = requestSR({'R3',1});
    case 'tPS'    % painting robot "R4" required
        granted = requestSR({'R4',1});
    otherwise
        error('Cant come here');
end % switch
fire = granted; % fire if the required resource is granted
```

COMMON_POST: In this file, the fired transition releases the resources it has acquired. Note that the three transitions **tC1**, **tC2**, and **tAP** do not release the resources they acquired as the resource will be used by their successor. On the other hand, the three transitions **tC1M1**, **tC2M2**, and **tPS**, not only releases the resource they acquired but also the resource acquired by the predecessor.

```
function [] = COMMON_POST(transition)
global global_info
switch transition.name
    case {'tC1', 'tC2', 'tAP'}
        % do not release any resources as they are
        % required by the sunsequent transition
    case 'tC1M1'
        release('tC1');     % release "C1" acquired by tC1
```

```
        release;              % also, release "R1" acquired by itself
    case 'tC2M2'
        release('tC2');       % release "C2" acquired by tC2
        release;              % also, release "R2" acquired by itself
    case 'tM1', release;      % release "M1" acquired by tM1
    case 'tM2', release;      % release "M2" acquired by tM2
    case 'tM1AS', release;      % release "R1" acquired by tM1AS
    case 'tM2AS', release;      % release "R2" acquired by tM2AS
    case 'tAS', release;      % release "R2" acquired by tAS
    case 'tPS'
        release('tAP'); % release "R3" acquired by tAP
        release;          % also, release "R4" acquired by itself
    otherwise
        error('Cant come here');
end % switch

disp(['At time=', num2str(current_time), ', transition ', ...
      transition.name, ' has just fired.']);

n = ntokens('pOB'); % get the number of tokens in pOB
if eq(n,3), global_info.STOP_SIMULATION = 1; end
```

The simulation result present some useful information on resource usage, thanks to the function **prnschedule**.

```
RESOURCE USAGE:

RESOURCE INSTANCES OF ***** C1 *****
tC1 [0 : 4]
tC1 [4 : 8]
tC1 [8 : 12]
Resource Instance: C1:: Used 3 times. Utilization time: 12

RESOURCE INSTANCES OF ***** C2 *****
tC2 [0 : 4]
tC2 [4 : 8]
tC2 [8 : 12]
Resource Instance: C2:: Used 3 times. Utilization time: 12

RESOURCE INSTANCES OF ***** M1 *****
tM1 [4 : 24]
tM1 [24 : 44]
tM1 [44 : 64]
Resource Instance: M1:: Used 3 times. Utilization time: 60

RESOURCE INSTANCES OF ***** M2 *****
tM2 [4 : 39]
tM2 [39 : 74]
tM2 [74 : 109]
Resource Instance: M2:: Used 3 times. Utilization time: 105

RESOURCE INSTANCES OF ***** R1 *****
tC1M1 [2 : 4]
tC1M1 [6 : 8]
tC1M1 [10 : 12]
tM1AS [24 : 26]
tM1AS [44 : 46]
tM1AS [64 : 66]
Resource Instance: R1:: Used 6 times. Utilization time: 12
```

```
RESOURCE INSTANCES OF ***** R2 *****
tC2M2 [2 : 4]
tC2M2 [6 : 8]
tC2M2 [10 : 12]
tM2AS [39 : 41]
tAS [41 : 49]
tM2AS [74 : 76]
tAS [76 : 84]
tM2AS [109 : 111]
tAS [111 : 119]
Resource Instance: R2:: Used 9 times. Utilization time: 36

RESOURCE INSTANCES OF ***** R3 *****
tAP [49 : 64]
tAP [84 : 99]
tAP [119 : 134]
Resource Instance: R3:: Used 3 times. Utilization time: 45

RESOURCE INSTANCES OF ***** R4 *****
tPS [51 : 64]
tPS [86 : 99]
tPS [121 : 134]
Resource Instance: R4:: Used 3 times. Utilization time: 39

RESOURCE USAGE SUMMARY:
C1:  Total occasions: 3   Total Time spent: 12
C2:  Total occasions: 3   Total Time spent: 12
M1:  Total occasions: 3   Total Time spent: 60
M2:  Total occasions: 3   Total Time spent: 105
R1:  Total occasions: 6   Total Time spent: 12
R2:  Total occasions: 9   Total Time spent: 36
R3:  Total occasions: 3   Total Time spent: 45
R4:  Total occasions: 3   Total Time spent: 39

*****  LINE EFFICIENCY AND COST CALCULATIONS: *****
  Number of servers:  k = 8
  Total number of server instances:  K = 8
  Completion = 134
  LT = 1072
  Total time at Stations: 321
  LE = 29.944 %
  **
  Sum resource usage costs: 0   (NaN% of total)
  Sum firing costs: 0   (NaN% of total)
  Total costs:  0
  **
```

References

- Davidrajuh, R. (2012) “Activity-oriented Petri Net for scheduling of resources.” In: IEEE International Conference on Systems, Man, and Cybernetics (SMC), pp. 1201–1206. IEEE
- James D. Stein, How Math Explains the World. Smithsonian. 2008.
- Davidrajuh, R. *Modeling Discrete-Event Systems with GPenSIM: An Introduction*; Springer, 2018.
- Skolud, B.; Krenczyk, D.; Davidrajuh, R. Solving Repetitive Production Planning Problems. An Approach Based on Activity-oriented Petri Nets. In *Advances in Intelligent Systems and Computing, Proceedings of the International Joint Conference SOCO’16, San Sebastián, Spain, 19–21 October 2016*; Springer: Cham, Switzerland, 2017; Volume 527, pp. 397–407.