Reggie Davidrajuh

GPenSIM
Reference Manual

University of Stavanger

2024

University of
Stavanger

Contents

Dedication:

This book is dedicated to my dear friends

Koneswaran Tharmalingam
(Trondheim, Norway)

and

Tharmalingam Sivakumar
(Oslo, Norway)

for helping me during the difficult times.

Thank you, my friends!

Preface

General-purpose Petri net Simulator (**GPenSIM**) is a toolbox (set of functions) that runs on MATLAB. Some universities use GPenSIM as the tool for modeling, simulation, and performance analysis of discrete systems. Academics in these universities chose GPenSIM as it is easy to learn, use, and extend.

<u>**In this book**</u>:
There are around 150 GPenSIM built-in functions. This book exclusively presents these GPenSIM built-in functions. GPenSIM version 11 systems files (MATLAB files) are grouped into 26 folders (from '**Check - Valid - Functions**' to '**Utility - Functions**') based on their use. This reference manual allocates one chapter for each folder, and the functions in the folder are described in the corresponding chapter. Each chapter also starts with a summary of the functions described in that chapter. (some of the folders in the system files are for the GPenSIM compiler's internal use only (e.g., 'PDF - maker' and 'Token - Gaming'); hence, these folders do not have corresponding chapters in this book).

Suppose the reader wants to study the details of a specific function. In that case, it is advisable to check the index pages (the last pages of this reference manual) first, as all the functions are listed alphabetically on the index pages.

This reference manual and the newer GPenSIM Version 11 are released simultaneously. GPenSIM Version 11 is almost the same as Version 10 (upward compatible), but there are a few new functions. Also, some of the reported bugs are fixed.

<u>**What is not given in this reference manual**</u>
The author of this reference manual (and developer of GPenSIM) has published three books on GPenSIM before (for more details, see Appendix-C):

- Book I: "**Modeling Discrete-Event Systems with GPenSIM: An Introduction**," **Springer**, **2018**. This book is a simple introduction (a "user manual") to GPenSIM. (this book covers only the fundamen-

tals; topics such as coloured Petri nets and resources are excluded and discussed in Book III).

○ Book II: "**Petri Nets for Modeling of Large Discrete Systems**," **Springer**, **2021**. This book discusses modeling large discrete systems by introducing a new modular Petri net theory.

○ Book III: "**Colored Petri Nets for Modeling of Discrete Systems: A Practical Approach With GPenSIM**," **Springer**, **2023**. This book discusses modeling real-life discrete systems in which the coloring of tokens, resources, and cost calculations is inevitable. Also, data structures of elements in a Petri Net model are presented in this book.

This reference manual neither introduces GPenSIM nor Petri nets. For a detailed study of GPenSIM, the readers are encouraged to look at the three books mentioned above.

GPenSIM website

GPenSIM Website: **http://www.davidrajuh.net/gpensim/**
Users can download source code for the examples in this reference manual and GPenSIM source files from the GPenSIM website.

Acknowledgement

The book is a result of my teaching and research at the Department of Electrical Engineering and Computer Science (IDE), University of Stavanger. I am indebted to the department IDE and the University of Stavanger for offering ample time and research facilities to complete this book.

My sincere thanks also go to the **UiS Scholarly Publishing Services** (**USPS**) at the University of Stavanger for publishing this book as an open-access book. John David Didriksen at the university library patiently replied to all my emails regarding the publication of this book. A big thank you to John David Didriksen as well.

I am also thankful to my wife, Ruglin, and my daughter, Ada, for tolerating my frequent absence in their daily lives.

Reggie Davidrajuh
October 2024.

Chapter 1

Check Valid Functions

This chapter lists four useful functions for coding the processor files. For example, when we enter the name of an entity (file, place, transitions, or resource) into the program code, we need to check whether this entity exists during runtime. The functions listed in this chapter are for checking whether the input name represents a valid element (e.g., is 't1' a valid transition?). Table-1.1 presents a summary of these functions.

Function	Description
`check_valid_file`	Is the filename valid?
`check_valid_place`	Is the place name(s) valid?
`check_valid_resource`	Is the resource name valid?
`check_valid_transition`	Is the transition name(s) valid?

Table 1.1: Check-Valid functions.

```
CAUTION!!!:
These functions throw an error if the input name
is not valid and terminates the program (returns
control to the MATLAB Command Prompt).

If a 'soft' checking is needed (without
terminating the program), use "Is" functions
(e.g., "is_place", "is_trans") instead!
```

1.1   check_valid_file

Function name: `check_valid_file`
Purpose: checks whether the named file exists.
Note: This function throws an error if the input name is not a valid file name

and terminates the program (returns control to the MATLAB Command
Prompt).
Input Parameter: Name of the file (text string)
Output Parameter: Natural number; $0 =$ file does not exists; > 0 : file
exists
This functions uses: `search_names()`
This function is used by: [processor files, MSF]
Sample use:

```
% in the main simulation file or processor files
file_exists = check_valid_file('t1_pre.m'); % does t1_pre exists?
```

1.2   check_valid_place

Function name: `check_valid_place`
Purpose: checks whether one or more place names are valid and returns
their indices.
Note: This function throws an error if the input name is not a valid place
name(s) and terminates the program (returns control to the MATLAB Com-
mand Prompt). If a 'soft' checking is needed (without terminating the pro-
gram), use the function "`is_place`" instead!
Input Parameter: One or more names of places (set of text strings)
Output Parameter: Set of place indices, if valid
This functions uses: `search_names()`
This function is used by: [processor files, MSF]
Sample use:

```
% in the main simulation file or processor files
set_of_place_indices = check_valid_places({'p1','p2','pE'});
```

Application example: A simple example ("Example-14: Check valid func-
tions" in Section 1.5) is given at the end of this chapter.
Related functions: `is_place`, `check_valid_transition`

1.3   check_valid_resource

Function name: `check_valid_resource`
Purpose: checks whether a named resource exists and returns its resource
index.
Note: This function throws an error if the input name is not a valid re-
source name and terminates the program (returns control to the MATLAB
Command Prompt).

Input Parameter: Name of the resource (text string)
Output Parameter: Resource index, if valid.
This functions uses: `search_names()`
This function is used by: [processor files, MSF]
Further info: Chapter 8, "GPenSIM Resources: The Basics," in Davidrajuh (2023).
Sample use:

```
% in the main simulation file or processor files
[r_index] = check_valid_resource('CNC-1');  % does "CNC-1" exists?
```

1.4   check_valid_transition

Function name: `check_valid_transition`
Purpose: checks whether one or more named transitions exist and returns their indices.
Note: This function throws an error if the input name is not a valid transition name(s) and terminates the program (returns control to the MATLAB Command Prompt). If a 'soft' checking is needed (without terminating the program), use the function "`is_trans`" instead!
Input Parameter: One or more names of transitions (set of text strings)
Output Parameters: Indices, if valid.
This functions uses: `search_names()`
This function is used by: [processor files, MSF]
Sample use:

```
% in the main simulation file or processor files
% do "t1", "t2", "tE" exist? If yes, what are their indices?
[set_of_trans_indices] = check_valid_transition({'t1','t2','tE'});
```

Application example: A simple example ("Example-14: Check valid functions" in Section 1.5) is given at the end of this chapter.
Related functions: `is_trans`, `check_valid_place`

1.5   Example-14: Check valid functions

Let us imagine that a main simulation file has just completed a simulation. We may want to check the number of leftover tokens in various places or how many times the individual transitions have fired. Listing-1.1 shows that the simulation is complete in a main simulation file followed by two subroutines ('check_place' and 'check_trans').

Listing 1.1: **Part of the Main Simulation File** (Example-14)

```
%
% any main simulation file is fine!
%
...
...

sim = gpensim(pni);

% simulation if complete
check_place(); % number of tokens in a place
check_trans(); % number of times a trans has fired
```

Subroutine 'check_place' (Listing-1.2) repeatedly ask a user to input a valid place name and then checks how many tokens are left in that place. Similarly, subroutine 'check_trans' (Listing-1.3) repeatedly ask a user to input a valid transition name and then checks how many times this transition has fired. Both of these functions use the "check valid functions."

Listing 1.2: **check_place** (Example-14)

```
function [] = check_place()
prompt = ['\nEnter a valid *place* name without using single ...
    quotation marks \n', ...
    '(note that wrong place name can crash the program)\n',...
    'press return key to quit:  '];
reply = 'pSomething';

while not(isempty(reply))
    reply = input(prompt, 's');
    if isempty(reply), return; end

    % is this a valid place name?
    if check_valid_place(reply)
        % find the number of tokens
        ntok = ntokens(reply);
        disp(' ');
        disp(['"',reply, '" has ', int2str(ntok),...
            ' tokens now.']);
    end
end
```

Listing 1.3: **check_trans** (Example-14)

```
function [] = check_trans()

prompt = ['\nEnter a valid !transition! name without using ...
    single quotation marks \n', ...
  '(note that wrong transition name can crash the program)\n',...
```

```matlab
  'press return key to quit:  '];

reply = 'tSomething';

while not(isempty(reply))
    reply = input(prompt, 's');
    if isempty(reply), return; end

    % is this a trans name?
    if check_valid_transition(reply)
        % find the number of times fired
        nfired = timesfired(reply);
        disp(' ');
        disp(['"',reply, '" has fired ', int2str(nfired),...
            ' times.']);
    end
end
```

<u>Note</u>: This example is almost identical to example 22 (Section 8.7). In example 14, we use "Check Valid Functions," which throws an error and terminates the program (returns control to the MATLAB Command Prompt) if the input name is not valid. In example 22, we use 'softer' (invalid names will not cause termination) "Is functions" instead!

Bibliography

Davidrajuh, R. (2023). *Colored Petri Nets for Modeling of Discrete Systems: A Practical Approach With GPenSIM*, Springer Nature.

Chapter 2

Cotree

This chapter presents two functions, namely `cotree` and `cotreei`. These two functions are for plotting the **reachability tree** (coverability tree).

The first function `cotree` works for **P/T Petri Nets** and produces output in the form of a graphical plot and ASCII text. Note that function `cotree` uses a file from the "Petri Net Control Toolbox" from the University of Cagliari.

The second function `cotreei` works only for a specific Petri Net extension known as **Petri Nets with Inhibitor Arcs**. The function `cotreei` only outputs ASCII text as a result.

Table-2.1 presents a summary of these two functions.

Function	Description
cotree	plot and print the reachability tree (coverability tree).
cotreei	print the coverability tree of a Petri Net with *inhibitor arcs*. Note that the graphic plot is not possible.

Table 2.1: Functions for Reachability (Coverability) Tree.

2.1   cotree

Function name: `cotree`
Full name: Reachability tree and Coverability tree.
Purpose: Creates the reachability tree of a Petri net and then plots it and prints it (ASCII listing).
Input Parameter - mandatory: Petri Net structure with initial dynamics (**pni** - the output of the function `initialdynamics`).
Input Parameter - optional: second input: whether the graphical plot is

needed; third input: whether ASCII (text) output is needed.

Output Parameter: `COTREE` matrix - a structure containing all the states and the firing transitions; `COTREE` matrix consists of several rows equal to the number of states. **In each row**:

1. The first element (a vector of natural numbers) is the state (marking), and the length will equal the number of places.
2. The second element (a natural number) is the index of the fired transition.
3. The third element (a natural number) is the state number of the parent state.
4. Finally, the fourth element (a natural number) is an indicator of the state:

    ○ 82 is the ASCII value of the character "R," meaning this is a Root (initial) state.
    ○ 84 is the ASCII value of the character "D," meaning this is a Duplicate state.
    ○ 68 is the ASCII value of the character "T," meaning this is a Terminal (dead) state.

Output to screen: printout and a plot of reachability tree.

This functions uses:

`print_cotree` : prints reachability tree as a text printout;

`plot_cotree` : this is a modified code of the function `PNCT_graph` from the University of Cagliari; this function outputs a graphical tree;

`gpensim_2_PNCT`: converts a Petri Net from GPenSIM format to PNCT format.

This function is used by: [in main simulation file (MSF)]

Further info: Chapter 4 "Analysis of Petri nets" in Davidrajuh (2018).

Sample use:

```
spng = pnstruct('cotree_example_pdf');
dyn.m0 = {'p1',2, 'p4', 1};  % initial markings
pni = initialdynamics(spng, dyn);
% two inputs: graphic plot only;
% output is the COTREE matrix
COTREE = cotree(pni, 1);

% three inputs: graphic plot & ASCII display
cotree(pni, 1, 1);

% NOTE: function "cotree" is for Petri net without
%   INHIBITOR arcs; For Petri net with INHIBITOR arcs,
%   use the function "cotreei"
```

Application example: A simple example ("Example-15: cotree" in Section 2.3) is given at the end of this chapter.
Related functions: `cotreei`

2.2 cotreei

Function name: `cotreei`
Full name: Reachability tree and Coverability tree for **Petri Net with Inhibitor Arcs**.
Purpose: Creates the reachability tree of a Petri net with Inhibitor Arcs and then displays it (ASCII listing).
Input Parameter - mandatory: Petri Net structure with initial dynamics (**pni** - the output of the function `initialdynamics`.
Input Parameter - optional: second input: the maximum size of the reachability tree.
Output Parameter: `COTREE` matrix - a structure containing all the states and the firing transitions; see Section 2.1.
Output to screen: printout of reachability tree.
This functions uses:
`build_cotree_i` : creates the reachability tree;
`print_cotree` : prints (displays) the reachability tree on screen.
This function is used by: [in main simulation file (MSF)]
Further info: Chapter 6.3 "Inhibitor Arcs" and 6.4 "Coverability Tree for Petri Nets with in Inhibitor Arcs" in Davidrajuh (2018).
Sample use:

```
% in Main Simulation File
spng = pnstruct('cotree_i_pdf'); % contains inhibitor arcs
dyn.m0 = {'p1',2, 'p4', 1};  % initial markings
pni = initialdynamics(spng, dyn);

% find the cotree, max size 10 states
cotreei(pni, 10);
```

Application example: A simple example ("Example-16: cotreei" in Section 2.4) is given at the end of this chapter.
Related functions: `cotreei` **Related functions**: `cotree`

2.3 Example-15: cotree

Fig.2.1 shows a simple Petri net for which we are going to generate the reachability graph. Listings 2.1 and 2.2 show the PDF and the main simulation file. The resulting reachability tree is shown in Fig.2.2.
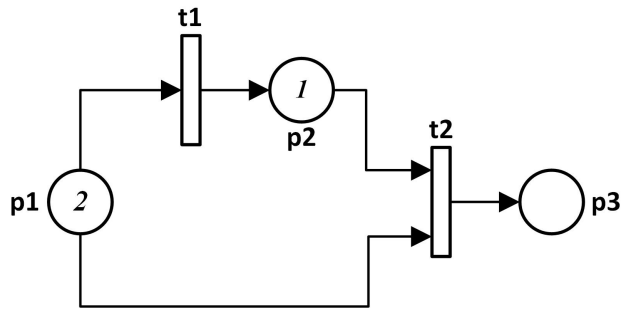
Figure 2.1: Petri net for generating reachability tree.

Listing 2.1: **PDF** (Example-15)

```
function [png] = ex_15_pdf()

png.PN_name = 'Example-15: cotree example';
png.set_of_Ps = {'p1', 'p2', 'p3'};
png.set_of_Ts = {'t1','t2'};
png.set_of_As = ...
    {'p1','t1',1, 't1','p2',1, ... % t1
     'p1','t2',1, 'p2','t2',1, 't2','p3',1}; % t2
```

Listing 2.2: **Main Simulation File** (Example-15)

```
clear all; clc; close all;

spng = pnstruct('ex_15_pdf');
dyn.m0 = {'p1',2, 'p2',1};

pni = initialdynamics(spng, dyn);
cotree(pni, 1, 1); % plot and text print
```

The function `cotree` also prints out the reachability tree in ASCII text format, which is given below. Note that the plot (Fig.2.2) and the text printout possess the same info. However, the printout shown below also provides the boundness (maximum tokens) of each place. Also, the acronyms for the states, ' ', 'D', and 'T', stand for Normal (or Tree), Duplicate, and Terminal.
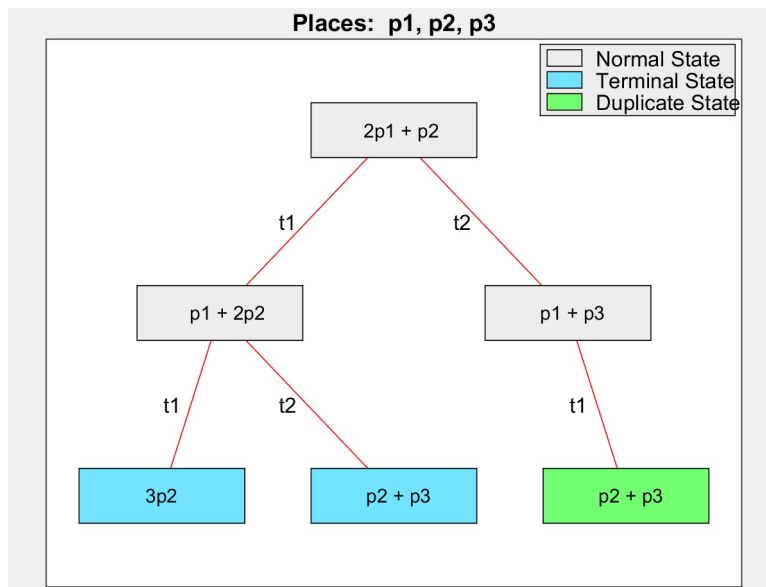
Figure 2.2: Generated reachability tree.

```
Firing times of transitions:  NOT given ...


======= Coverability Tree =======
State no.:  1 ROOT node
2p1 + p2


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


State no.:  3 Firing event:  t2
State:  p1 + p3
Node type:  '  '  Parent state:  1


State no.:  4 Firing event:  t1
State:  3p2
Node type:  'T' Parent state:  2


State no.:  5 Firing event:  t2
State:  p2 + p3
Node type:  'T' Parent state:  2


State no.:  6 Firing event:  t1
State:  p2 + p3
Node type:  'D' Parent state:  3



Boundedness:
p1 :  2
p2 :  3
p3 :  1


Liveness:
```

2.4   Example-16: cotreei
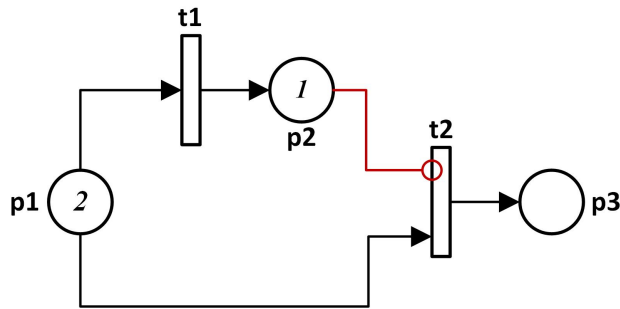
Figure 2.3: Petri net with Inhibitor Arc for generating reachability tree.

Fig.2.3 shows a simple Petri net with an inhibitor arc for which we are going to generate the reachability graph. Listings 2.3 and 2.4 show the PDF and the main simulation file. Note that the PDF file (Listing-2.3) also includes the declaration of the set of inhibiting arcs (set_of_Is).

Listing 2.3: **PDF** (Example-16)

```
function [png] = ex_16_pdf()

png.PN_name = 'Example-16: cotreei example';
png.set_of_Ps = {'p1', 'p2', 'p3'};
png.set_of_Ts = {'t1','t2'};
png.set_of_As = ...
    {'p1','t1',1, 't1','p2',1, ... % t1
     'p1','t2',1, 't2','p3',1}; % t2

%%%%%%% Inhibiting Arcs %%%%%%%
png.set_of_Is = {'p2','t2',1};
```

Listing 2.4: **Main Simulation File** (Example-16)

```
clear all; clc; close all;

spng = pnstruct('ex_16_pdf');
dyn.m0 = {'p1',2, 'p2',1};

pni = initialdynamics(spng, dyn);

% printout max 10 states of reachablity tree
```

```
cotreei(pni, 10);
```

The function `cotreei` only prints out the reachability tree in text format (and no graphical plot), which is given below.

```
Firing times of transitions:  NOT given ...

======= Coverability Tree =======
State no.:  1 ROOT node
2p1 + p2

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

State no.:  3 Firing event:  t1
State:  3p2
Node type:  'T' Parent state:  2
```

Bibliography

Davidrajuh, R. (2018). *Modeling Discrete-Event Systems with GPenSIM*, Springer International Publishing, Cham.

Chapter 3

Firing Sequence

The function `firingseq` is for executing (firing) a sequence of a pre-defined set of transitions in the strict order of the sequence. The function `firingseq` forces the simulator to allow only the enabled transitions in the pre-defined firing sequence to fire, and the other enabled transitions will be blocked from firing; for more info, see Chapter 4.2 "Firing Sequence" in Davidrajuh (2018).

Using 'firingseq': In the MSF, we assign the firing sequence (e.g., {'t1', 't2', 't3'}) to the OPTION "FIRING_SEQ"; Also, two other OPTIONs (such as "FS_REPEAT" and "FS_ALLOW_PARALLEL") are related to this function. Finally, the function `firingseq` is called by assigning this function as the only input for 'fire' in COMMON_PRE.

3.1   firingseq

Function name: `firingseq`
Full name: Firing Sequence.
Purpose: To fire only the transitions declared in the OPTION "global_info.FIRING_SEQ". Also, the transitions declared in this sequence will be allowed to fire in the strict sequence order.
Input Parameter - mandatory: (none)
Outputs: (none)
This functions uses:
`is_firing` : to check whether any other transitions are currently firing.
This function is used by: [in COMMON_PRE]
Further info: Chapter 4.2 "Firing Sequence" in Davidrajuh (2018).
Sample use: See the following example (Example-17: "Firing Sequence").
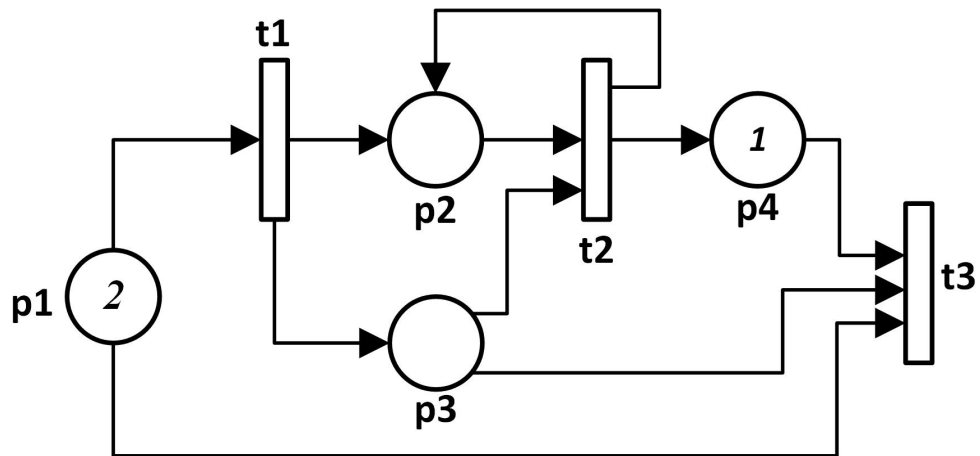
Figure 3.1: Petri net for checking the function 'firing sequence'.

3.2   Example-17: Firing Sequence

Fig.3.1 shows a Petri net with an initial marking of two tokens in **p1** and
one in **p4**, as shown in the Listing-3.1. The coverability tree (Fig.3.2) for
this initial marking includes two dead states if the firing sequence from the
initial state is either {**t1**, **t3**} or {**t1**, **t1**, **t2**, **t2**}. Let us check whether it is
true using the firing sequence.

   Listings 3.1, 3.2, and 3.3 show the main simulation file, PDF, and COM-
MON_PRE, respectively.

Listing 3.1: **Main Simulation File** (example-17)

```
clear all; close all; clc;
global global_info
global_info.STOP_AT = 80;
global_info.FIRING_SEQ = {'t1','t1','t2','t2'};

spng = pnstruct('ex_17_pdf');
dyn.m0 = {'p1',2, 'p4',1};
dyn.ft = {'allothers',1};
pni = initialdynamics(spng, dyn);

results = gpensim(pni);
cotree(pni, 1, 0); % plot graphically; no ASCII text
prnss(results);
```

Listing 3.2: **PDF** (Example-17)

```
function [png] = ex_17_pdf()
```
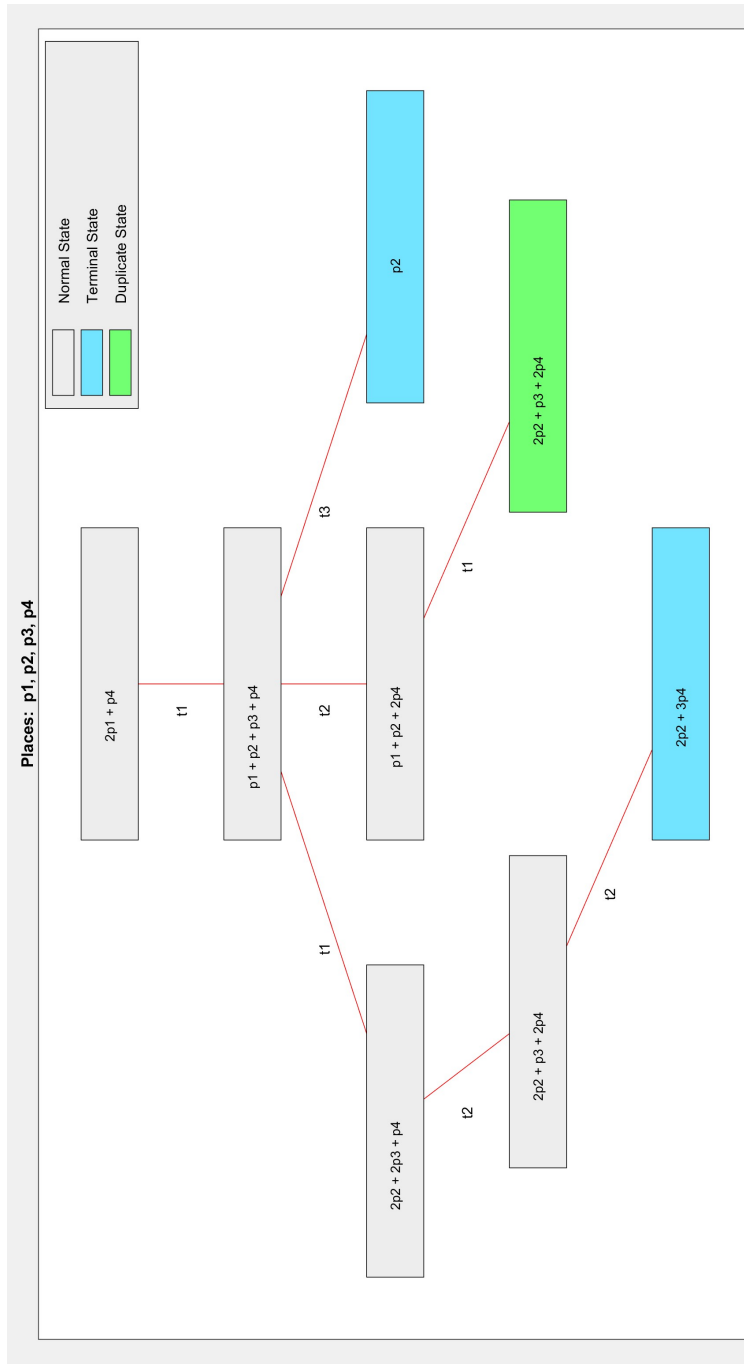
Figure 3.2: Coverability tree for the Petri net shown in Fig.3.1.

```matlab
png.PN_name = 'Example-17: Firing sequence';
png.set_of_Ps = {'p1', 'p2', 'p3', 'p4'};
png.set_of_Ts = {'t1','t2', 't3'};
png.set_of_As =  {...
    'p1','t1',1, 't1','p2',1, 't1','p3',1,... % t1
    'p2','t2',1, 'p3','t2',1,... % t2
    't2','p2',1, 't2','p4',1,... % t2
    'p1','t3',1, 'p3','t3',1, 'p4','t3',1}; % t3
```

Listing 3.3: **COMMON_PRE** (example-17)

```matlab
function [fire, trans] = COMMON_PRE(trans)
% assign firingseq directly to 'fire'
% otherwise program malfunctions
fire = firingseq();
```

The screen dump given below verifies that one of the firing sequences, {**t1**, **t1**, **t2**, **t2**}, indeed leads to a dead state.

```
 ** Time:  0 **
State:0 (Initial State):  2p1 + p4
At time:  0, Enabled transitions are:  t1
At time:  0, Firing transitions are:  t1


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


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


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


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


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


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


** Time:  4 **
State:  4
Fired Transition:  t2
Current State:  2p2 + 3p4
Virtual tokens:  (no tokens)


Right after new state-4 ....
At time:  4, Enabled transitions are:
At time:  4, Firing transitions are:
```

Bibliography

Davidrajuh, R. (2018). *Modeling Discrete-Event Systems with GPenSIM*, Springer International Publishing, Cham.

Chapter 4

Get Functions

This chapter describes a set of utility functions for extracting diverse information from the GPenSIM environment; for example, information such as token color or a place's output transitions can be extracted.

Tables 4.1 and 4.2 summarize these "get" functions. Some of these functions extract the colors of tokens. For a complete study on token colors in GPenSIM, Chapter 2, "Colored Petri net: The Basics," in Davidrajuh (2023) is suggested.

Function	Description
get_all_tokens	returns all the tokens in all the places.
get_color	returns a token's colors.
get_cost	returns token's cost.
get_current_colors	returns a summation of all the colors of all the tokens anywhere.
get_firingtime	return the firing time of a transition.
get_inputplace	returns a transition's input places.
get_inputtrans	returns a place's input transitions.
get_outputplace	returns a transition's output places.
get_outputtrans	returns a place's output transitions.
get_place	returns a place's the complete data structure.
get_priority	returns a transition's the current priority.
get_tokCT	returns a token's creation time.
get_token	returns a token's complete information.
get_tokens	returns complete information about a set of tokens in a place.
get_trans	returns a transition's complete data structure.

Table 4.1: Get-functions (part I).

Function	Description
`nplaces`	returns the number of places in the Petri net (same as `PN.No_of_places`, where `PN` is the Petri net run-time structure).
`nresources`	returns the number of resources in the Petri net (same as `PN.No_of_system_resources`).
`ntokens`	returns the current number of tokens in a place.
`ntrans`	returns the number of transitions in the Petri net (same as `PN.No_of_transitions`).
`pname`	returns a place's name.
`rname`	returns a resource's name.
`timesfired`	returns the number of times a transition has fired already.
`tname`	returns a transition's name.

Table 4.2: Get-functions (part II).

4.1 get_all_tokens

Function name: `get_all_tokens`
Full name: Get all the token info in all the places.
Purpose: Collect token info (tokID, creation_time, color, cost) of all the tokens in all the places.
Inputs: (none)
Output Parameter: A structure of token info (tokID, creation_time, color, cost) of all the tokens in all the places.
This function uses: `get_tokens`: to get the token info of a single place.
Sample use:

```
global_token_bank = get_all_tokens();
```

4.2 get_color

Function name: `get_color`
Purpose: Get the colors of a specific token in a specific place.
Input Parmeters: 1) a place's name or index; 2) tokID, the ID of the token.
Output Parmeter: A set of colors of the token.

This function uses: `check_valid_place`: to check whether the given name of the place or the pi is valid.
Sample use:

```
colors = get_color('p3', tokIDi);
disp_str = ['colors of token (', int2str(tokIDi), '): '];
for i = 1:numel(colors)
   disp_str = [disp_str,  colors{i}, ' '];
end
disp(disp_str);
```

4.3   get_cost

Function name: `get_cost`
Purpose: Get the cost of a specific token in a specific place.
Input Parameters: 1) a place's name or index; 2) tokID, the ID of the token.
Output Parameter: The token's cost (a real number).
This function uses: `check_valid_place`: to check whether the given name of the place or the pi is valid.
Sample use:

```
token_cost = get_cost('p3', tokIDi);
disp(['Cost of the token (', int2str(tokIDi),...
       ') is: ', num2str(token_cost)]);
```

Application example: A simple example ("Example-18: get_cost" in Section 4.24) is given at the end of this chapter.

4.4   get_current_colors

Function name: `get_current_colors`
Purpose: This function outputs all the colors of all the tokens that reside in various places.
Inputs: (none)
Output Parameter: All the colors of all the tokens in all the places.
This function uses: (none)
Sample use:

```
[set_of_all_colors] = get_current_colors();
```

4.5   get_firingtime

Function name: `get_firingtime`
Purpose: This function returns the firing time of a specific transition.
Input Parameter: a transition's name or index.
Output Parameter: the firing time of the transition.
This function uses: `check_valid_transition`: to check whether the input name or index of the transition is valid.
Sample use:

```
[firingTime] = get_firingtime('tRobot_1');
```

4.6   get_inputplace

Function name: `get_inputplace`
Purpose: This function returns the set of input places of a transition.
Input Parameter: a transition's name or index.
Output Parameters: 1) Set of indices of places; 2) Set of names (text strings) of places.
This function uses: `check_valid_transition`: to check whether the input transition name or index is valid.
Sample use:

```
% get input places of transition "t1"
[pIndices1, pNames1] = get_inputplace('t1');
% get input places of transition "t2", which has index "5"
[pIndices2, pNames2] = get_inputplace(5);
```

Related functions: `get_inputtrans`, `get_outputtrans`, `get_outputplace`

4.7   get_inputtrans

Function name: `get_inputtrans`
Purpose: This function returns the set of input transitions of a place.
Input Parameter: a place's name or index.
Output Parameters: 1) Set of indices of transitions; 2) Set of names (text strings) of transitions.
This function uses: `check_valid_place`: to check whether the input place name or index is valid.
Sample use:

```matlab
% get input transitions of place "p1"
[tIndices1, tNames2] = get_inputtrans('p1');
% get input transitions of place "p2", which has index "7"
[tIndices2, tNames2] = get_inputtrans(7);
```

Related functions: get_inputplace, get_outputtrans,
get_outputplace

4.8   get_outputplace

Function name: get_outputplace
Purpose: This function returns the set of output places of a transition.
Input Parameter: a transition's name or index.
Output Parameters: 1) Set of indices of places; 2) Set of names (text strings) of places.
This function uses: check_valid_transition: to check whether the input transition name or index is valid.
Sample use:

```matlab
% get output places of transition "t1"
[pIndices1, pNames1] = get_outputplace('t1');
% get output places of transition "t2", which has index "5"
[pIndices2, pNames2] = get_outputplace(5);
```

Related functions: get_inputplace, get_inputtrans,
get_outputtrans

4.9   get_outputtrans

Function name: get_outputtrans
Purpose: This function returns the set of output transitions of a place.
Input Parameter: a place's name or index.
Output Parameters: 1) Set of indices of transitions; 2) Set of names (text strings) of transitions.
This function uses: check_valid_place: to check whether the input place name or index is valid.
Sample use:

```matlab
% get output transitions of place "p1"
[tIndices1, tNames2] = get_outputtrans('p1');
% get output transitions of place "p2", which has index "7"
[tIndices2, tNames2] = get_outputtrans(7);
```

Related functions: get_inputplace, get_inputtrans,
get_outputplace.

4.10   get_place

Function name: `get_place`
Purpose: This function extracts the complete info about a place (place structure) from the Petri net run-time structure (PN.global_places).
Input Parameter: a place's name or index.
Output Parameter: Place structure.
This function uses: `check_valid_place`: to check whether the input place name or index is valid.
Sample use:

```
% get the complete info about "p1"
pStrcture1 = get_place('p1');
% get the complete info about place "p2", which has index "7"
pStrcture2 = get_place(7);
```

Related functions: `get_trans`

4.11   get_priority

Function name: `get_priority`
Purpose: This function returns the priority of a transition.
Input Parameter: a transition's name or index.
Output Parameter: priority (natural number: 0, 1, 2, . . .).
This functions uses: `check_valid_trans`: to check whether transition name or index is valid.
Further info: See Chapter 6.5, "Prioritizing Transitions" in Davidrajuh (2018).
Sample use:

```
% get priority of transition is named 't1'
prior1 = get_priority('t1');
% get priority of transition t2 which has index 10
prior2 = get_priority(10);
```

4.12   get_tokCT

Function name: `get_tokCT`
Purpose: This function returns the tokCT (token creation time) of a token.
Input Parameters: 1) a place's name or index; 2) token ID (tokID).
Output Parameter: token creation time (a positive real number).
This function uses: `get_tokens`: to get all the details of a token.
Further info: See Chapter 5.1, "Functions for Time-based Token Selection,"

in Davidrajuh (2023).
Sample use:

```
% tokCT of the token (tokID = 121) in the place 'p1'
[tokCreationTime] = get_tokCT ('p1', 121);
```

4.13 get_token

Function name: `get_token`
Purpose: This function returns complete info (token bank) of a token.
Input Parameters: 1) a place's name or index; 2) token ID (tokID).
Output Parameter: token bank (complete info about the token).
This function uses: `check_valid_place`: to check whether the input place name (or index) is valid.
Further info: See Chapter 2, "Colored Petri net: The Basics," in Davidrajuh (2023).
Sample use:

```
% token with tokID 101 in the place 'p1'
[token101_info] = get_tokens('p1', 101);
```

Related functions: `get_tokens`

4.14 get_tokens

Function name: `get_tokens`
Purpose: This function returns complete info (token bank) of a set of tokens in a place.
Input Parameters: 1) a place's name or index; 2) set of token IDs (tokIDs).
Output Parameter: token bank (complete info about the set of tokens).
This function uses: `check_valid_place`: to check whether the input place name (or index) is valid.
Further info: See Chapter 2, "Colored Petri Net: The Basics," in Davidrajuh (2023).
Sample use:

```
% tokens with tokIDs [20, 77, 101] in the place 'p1'
[set_of_tokens_info] = get_tokens('p1', [20, 77, 101]);
```

Related functions: `get_token`

4.15 get_trans

Function name: `get_trans`
Purpose: This function extracts the complete info about a transition (transition structure) from the Petri net run-time structure (PN.global_transitions).
Input Parameter: a transition's name or index.
Output Parameter: Transition structure.
This function uses: `check_valid_transition`: to check whether the input transition name or index is valid.
Sample use:

```matlab
% get the complete info about transition "t1"
tStructure1 = get_trans('t1');
% get the complete info about transition "t2", which has index "7"
tStructure2 = get_trans(7);
```

Related functions: `get_place`

4.16 nplaces

Function name: `nplaces`
Full name: Number of places in the Petri net.
Purpose: This function returns the total number of places in the Petri net.
Inputs: (none)
Output Parameter: The total number of places in the Petri net.
This function uses: (none)
Sample use:

```matlab
% get the total number of places in this Petri net
totalPlaces = nplaces();
```

Related functions: `nresources`, `ntrans`

4.17 nresources

Function name: `nresources`
Full name: Number of resources in the system.
Purpose: This function returns the total number of resources in the system.
Inputs: (none)
Output Parameter: The total number of resources in the system.
This function uses: (none)
Sample use:

```
% get the total number of resources in the system
totalRes = nresources();
```

Related functions: `ntrans, nplaces`

4.18   ntokens

Function name: `ntokens`
Full name: Number of tokens
Purpose: This function returns the total number of tokens in a place.
Input Parameter: a place's name or index.
Output Parameter: the total number of tokens in the place.
This function uses: `get_place`: to get the complete place info so that the number of tokens in this place can be extracted.
Sample use:

```
% get the total number of tokens in the place 'p1'
number_of_tokens = ntokens('p1');
```

Related functions: `get_place`

4.19   ntrans

Function name: `ntrans`
Full name: Number of transitions in the Petri net.
Purpose: This function returns the total number of transitions in the Petri net.
Inputs: (none)
Output Parameter: The total number of transitions in the Petri net.
This function uses: (none)
Sample use:

```
% get the total number of transitions in this Petri net
totalTrans = ntrans();
```

Related functions: `nplaces, nresources`

4.20   pname

Function name: `pname`
Full name: Place Name.
Purpose: This function returns the name of the place, which is identified by an index.

Input Parameter: a place's index.
Output Parameter: The name (ASCII text string) of the place.
This function uses: (none)
Sample use:

```
% get the name of the place identified by index 33
name_of_place = pname(33);
```

Related functions: tname, rname

4.21   rname

Function name: rname
Full name: Resource Name.
Purpose: This function returns the name of the resource, which is identified
by an index.
Input Parameter: a resource's index.
Output Parameter: The name (ASCII text string) of the resource.
This function uses: (none)
Sample use:

```
% get the name of the resource identified by index 5
name_of_resource = rname(5);
```

Related functions: pname, tname

4.22   timesfired

Function name: timesfired
Full name: Number of times fired.
Purpose: This function returns the number of times a transition has fired
already.
Input Parameter: a transition's name or index.
Output Parameter: The number of times (natural integer) the transition
has fired so far.
This function uses: get_trans: to get the complete info about the
transition so that the number of times it has fired so far can be extracted.
Sample use:

```
% how many times "t1" has fired so far?
number_of_times_fired = timesfired('t1');
```

4.23   tname

Function name: `tname`
Full name: Transition Name.
Purpose: This function returns the name of the transition, which is identified by an index.
Input Parameter: a transition's index.
Output Parameter: The name (ASCII text string) of the transition.
This function uses: (none)
Sample use:

```
% get the name of the transition identified by index 11
name_of_transition = tname(11);
```

Related functions: `pname`, `rname`

4.24   Example-18: get_cost
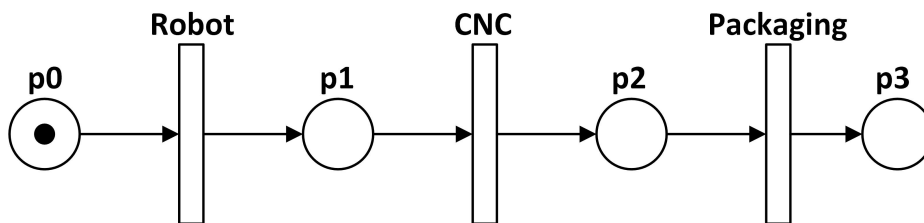
Figure 4.1: Petri net model of three machines in action.

This simple exercise is to practice the function 'get_cost'. Fig.4.1 shows a production facility in which material passes through a series of three machines: a robot, a CNC machine, and a packaging machine. We want to study the development of the product cost as the material/product flows through the machines.

Listing-4.1 shows the PDF.

Listing 4.1: **PDF** (Example-18)

```
% Example-18: PDF (function get_cost)
function [png] = ex_18_pdf()
png.PN_name = 'Example-18: testing "get_cost"';
png.set_of_Ps = {'p0','p1','p2','p3'};
png.set_of_Ts = {'Robot', 'CNC', 'packaging'};
png.set_of_As = {...
    'p0','Robot',1, 'Robot','p1',1,... % Robot
```

```
'p1','CNC',1, 'CNC','p2',1,... % CNC
'p2','packaging',1, 'packaging','p3',1}; % packaging
```

Listing-4.2 shows the MSF. In MSF, we first create an empty global array `global_info.COSTS` so that this array can be stuffed with costs of tokens in **p1**, **p2**, and **p3**. We assume that raw material costs are negligible (in other words, the cost of the initial token in **p0** is zero).

Listing 4.2: **MSF** (Example-18)

```
% Example-18: MSF (get_ost)
clear all; clc; close all;
global global_info
global_info.STOP_AT = 80;
% global variable to store the costs
global_info.COSTS = [];

spng = pnstruct('ex_18_pdf');
dyn.m0 = {'p0',1}; % tokens initially
dyn.ft = {'allothers',10};
% fixed & variable costs of machine operations
dyn.fc_fixed = {'Robot',3, 'CNC',250....
                'packaging',70};
dyn.fc_variable = {'Robot',5, 'CNC',50,...
                'packaging',22};

pni = initialdynamics(spng, dyn);
sim = gpensim(pni);

% plot the cost development as a bar-chart
X = categorical({'After Robot','After CNC',...
    'After packaging'});
X = reordercats(X,{'After Robot','After CNC',...
    'After packaging'});
bar(X,global_info.COSTS);
title('Development of the product cost');
```

Listing-4.3 shows the COMMON_POST. Note that we use COMMON_POST to extract the costs of tokens as soon as these tokens are deposited into the output places. If we don't extract the costs of tokens using COMMON_POST (that is, immediately after the firing of transitions), the tokens will disappear, consumed by the next transitions. We are not using COMMON_PRE as there is no need for it.

Listing 4.3: **COMMON_POST** (Example-18)

```
function [] = COMMON_POST(transition)

global global_info
fired_trans = transition.name;
```

```
switch fired_trans
    case 'Robot'
        tokIDr = tokenAny('p1',1);
        cost_p1R = get_cost('p1', tokIDr);
        global_info.COSTS = [global_info.COSTS cost_p1R];

    case 'CNC'
        tokIDc = tokenAny('p2',1);
        cost_p2C = get_cost('p2', tokIDc);
        global_info.COSTS = [global_info.COSTS cost_p2C];

    case 'packaging'
        tokIDp = tokenAny('p3',1);
        cost_p3 = get_cost('p3', tokIDp);
        global_info.COSTS = [global_info.COSTS cost_p3];
end
```

The result of this simulation - the bar chart depicting the cost development of the product as it passes through the three machines - is shown in Fig.4.2.
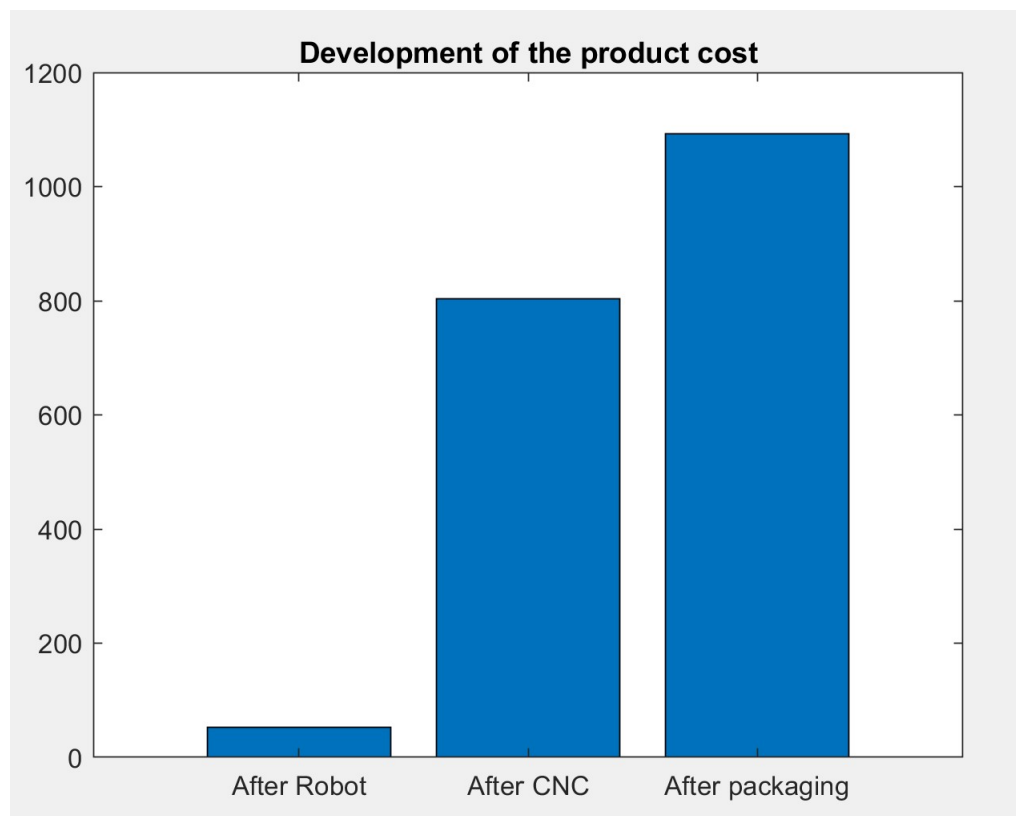
Figure 4.2: Cost development of the product.

Bibliography

Davidrajuh, R. (2018). *Modeling Discrete-Event Systems with GPenSIM*, Springer International Publishing, Cham.

Davidrajuh, R. (2023). *Colored Petri Nets for Modeling of Discrete Systems: A Practical Approach With GPenSIM*, Springer Nature.

Chapter 5

gpensim

Function "`gpensim`" is the main function for simulation. Table-5.1 presents a summary of these two functions.

Function	Description
gpensim	executes simulation on the Petri net structure with initial dynamics (`pni`).
gpensim_ver	prints the GPenSIM version number.

Table 5.1: gpensim functions.

5.1 gpensim

Function name: `gpensim`
Purpose: Once the Petri net structure with initial dynamics (**pni**) is constructed by the function `initialdynamics`, pni is passed to `gpensim` for simulation. Hence, `gpensim` is the main function for simulation.
Input Parameter: `pni` - the Petri net structure with initial dynamics; if the input is omitted `gpensim` will output the current version number.
Output Parameter: Simulation results as a structure; the simulation results can be plotted using the function `plotp` or listed on the screen by the function `prnss`.
This function uses: (many sub-functions).
This function is used by: [in Main Simulation File (MSF)]
Further info: Ref. Davidrajuh (2018) introduces `gpensim`.
Sample use:

```
spng = pnstruct('simple_pn_pdf');
dyn.m0 = {'p1',3, 'p2',4};
pni = initialdynamics(spng, dyn);
```

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

Related functions: gpensim_ver, pnstruct, initialdynamics,
plotp, prnss

5.2   gpensim_ver

Function name: gpensim_ver
Full name: gpensim version
Purpose: This function prints the current version of GPenSIM, which is
version 11, 01 December 2023.
Inputs: (none).
Output Parameter: (none).
Output on screen: the version will be printed on the screen.
This function uses: (none)
This function is used by: [in Main Simulation File (MSF)]
Sample use:

```
%%% print the current version of gpensim
gpensim_ver();
%%%%% alternatively
gpensim();
```

The following message will be displayed (output) on the MATLAB Command Window:

```
-----
GPenSIM version 11; Last update:  December 01,
2023

(C) Reggie.Davidrajuh@uis.no

http://www.davidrajuh.net/gpensim
-----
```
Related functions: gpensim

Bibliography

Davidrajuh, R. (2018). *Modeling Discrete-Event Systems with GPenSIM*,
    Springer International Publishing, Cham.

Chapter 6

Graphs and Cycles

Petri Net is a bipartite graph containing two types of elements: places and transitions. Hence, being a graph, all graph algorithms can be applied on a Petri Net, if we could convert the bipartite Petri net into a directed homogeneous graph ('digraph'). This chapter presents some of the graph algorithms that are useful for the analysis of Petri Nets. Table-6.1 presents a summary of these functions.

Function	Description
`convert_PN_V`	converts a directed bipartite Petri net into a homogeneous directed graph (digraph).
`cycles`	detects the cycles (*aka* circuits) of a digraph.
`prncycles`	printout cycles detected by the function `cycles`.
`stronglyconn`	finds the strongly connected components of a digraph.
`stronglyconn_rader`	(same as `stronglyconn`) finds the strongly connected components of a graph; however, it uses a faster Rader's algorithm.
`prnscc`	printout the connected components detected by the functions `stronglyconn` or `stronglyconn_rader`.

Table 6.1: Graphs and Cycles functions.

6.1   convert_PN_V

Function name: `convert_PN_V`
Purpose: To convert a bipartite Petri Net into a homogeneous directed

graph to apply the usual graph algorithms.

Input Parameter: Static Petri net graph (`spng`, output of the function `pnstruct`) or Petri net structure with initial dynamics (`pni`, output of the function `initialdynamics`).

Output Parameter: A homogeneous directed graph (**V**) on which the usual graph algorithms can be run. **V** is a MATLAB structure with two fields: 1) **V.nodes** is a set of node names (text strings). 2) **V.A** is the adjacency matrix of the digraph.

This function uses: (none).

This function is used by: [graph algorithms such as depth-first-search (DFS), strongly connected components (SCC)]

Further info: Section 8.5, "Interfacing with Graph Algorithms Toolbox," in Davidrajuh (2018).

Sample use:

```
spng = pnstruct('simple_pn_pdf');
V = convert_PN_V(spng);
% V.A: Incidence Matrix;  V.nodes: nodes
```

Application example: Two simple examples ("Example-20: convert_PN_V" in Section 6.6 and "Example-21: cycles" in Section 6.7) are given at the end of this chapter.

Related functions: `stronglyconn`, `cycles`, `dfs`

6.2   cycles

Function name: `cycles`

Purpose: To detect all the elementary circuits (cycles) in a directed graph (V).

Input Parameter: Directed graph (V, output of function `convert_PN_V`).

Output Parameter: The cycles in the graph (V.cycles), where each row represents a circuit.

For example, for a Petri net of seven elements, V.cyles:

2 6 0 0 0 0 0

3 5 1 7 0 0 0

means there are two cycles; cycle 1 consists of second and sixth elements; cycle 2 consists of third, fifth, first, and seventh elements. Note that each row is padded with trailing zeros.

CAUTION: This function uses an inefficient algorithm for finding cycles and seems to miss some cycles!!!

This function uses: `cycle_detection`.

This function is used by: [`mincyctime`, a function to detect the cycle times of an event graph (marked graph).]

Further info: Chapter 11, "Discrete Systems as Petri Modules," in Davidra-juh (2021).
Sample use:

```
spng = pnstruct('simple_pn_pdf');
[G] = convert_PN_V(spng);
V = cycles(G); % V.cycles are the cycles found
```

Application example: Two simple examples ("Example-20: convert_PN_V" in Section 6.6 and "Example-21: cycles" in Section 6.7) are given at the end of this chapter.
Related functions: `convert_PN_V`

6.3 prncycles

Function name: `prncycles` or `print_cycles`
Full name: Print cycles in a digraph.
Purpose: To printout all the cycles deteced by the function `cycles`.
Input Parameter: Directed graph (V) with the cycles - output of function `cycles`.
Output Parameter: (none).
Output on screen: The cycles and their elements.
This function uses: none.
This function is used by: [main simulation file.]
Sample use:

```
spng = pnstruct('simple_pn_pdf');
[G] = convert_PN_V(spng);
V = cycles(G); % V.cycles are the cycles found
prncycles(V); % same as print_cycles(V)
```

Application example: Two simple examples ("Example-20: convert_PN_V" in Section 6.6 and "Example-21: cycles" in Section 6.7) are given at the end of this chapter.
Related functions: `cycles`, `convert_PN_V`

6.4 stronglyconn

Function name: `stronglyconn`
Full name: Strongly Connected Components.
Purpose: To detect all the strongly connected components of a Petri Net.
Input Parameter: 1) Static Petri net graph (`spng`, the output of the function `pnstruct`) or Petri net structure with initial dynamics (`pni`, the output of the function `initialdynamics`).

2) (Optional input): '1' to suppress intermediate results.

Output Parameter: 1) Matrix SCC in which each row represents a strongly connected component. 2) V - the digraph output by the function `convert_PN_V`. SCC matrix has a number of rows, and each row represents a connected component. For example, for a Petri net of seven elements, SCC:

0 0 0 1 0 0 0

0 1 0 0 0 1 0

1 0 1 0 1 0 1

means there are three components; component 1 (row 1) consists of just one element (the fourth); component 2 (row 2) consists of two elements, second and sixth; component 3 (row 3) consists of four elements, the first, third, fifth, and seventh elements.

This function uses: `stronglyconn_rader`, `convert_PN_V`.

Further info: Chapter 11, "Discrete Systems as Petri Modules," in Davidrajuh (2021).

Sample use:

```
spng = pnstruct('simple_pn_pdf');
[SCC, V] = stronglyconn(spng, 1);  % each row of SCC is a ...
    component
```

Application example: Two simple examples ("Example-20: convert_PN_V" in Section 6.6 and "Example-21: cycles" in Section 6.7) are given at the end of this chapter.

Related functions: `convert_PN_V`, `cycles`

6.5   prnscc

Function name: `prnscc`

Full name: Printout Strongly Connected Components.

Purpose: To printout the strongly connected components detected by the function `stronglyconn` (or `stronglyconn_rader`).

Input Parameter: 1) SCC (strongly connected components, the output of the function `stronglyconn`). 2) V: digraph, the output of the function `convert_PN_V`.

Output Parameter: none.

Output on screen: Connected components and their elements.

This function uses: none.

Sample use:

```
spng = pnstruct('simple_pn_pdf');
[SCC, V] = stronglyconn(spng, 1);  % each row of SCC is a ...
    component
prn(SCC, V);
```

Application example: Two simple examples ("Example-20: convert_PN_V" in Section 6.6 and "Example-21: cycles" in Section 6.7) are given at the end of this chapter.
Related functions: `convert_PN_V`, `cycles`

6.6 Example-20: convert_PN_V
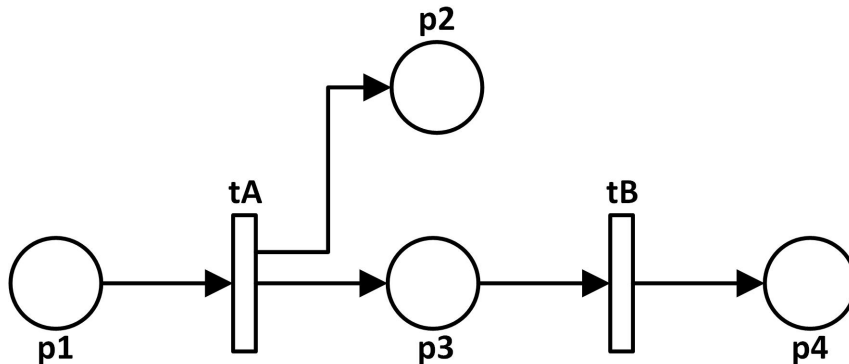
Figure 6.1: Applying graph algorithms to a Petri net.

This example is adapted from Davidrajuh (2018). Fig.6.1 shows a Petri net with four places (**p1** to **p4**) and two transitions (**tA** and **tB**).

In the main simulation file (Listing-6.1), we compute the following:

1. Convert the Petri net into a directed graph (V).
2. Feed V into the function `cycles` to find out the number of cycles in this Petri net.
3. Feed the static Petri net graph structure (`spng`) to the function `stronglyconn` to find out the number of strongly connected components (or 'connected components') in this Petri net.

Listing 6.1: **MSF** (Example-20)

```
clear all; clc; close all;

spng = pnstruct('ex_20_pdf');
disp('Petri net"s Extended Incidence Matrix (A): ');
disp(spng.incidence_matrix);

%%%%% 1: Convert Petri net into digraph V
V = convert_PN_V(spng);
disp(' ');
```

```matlab
disp('The Digraph"s Adajacency matrix: ');
disp(V.A);

%%%%% 2: Find the cycles in V
V = cycles(V);
V.cycles
prncycles(V); % print cycles

%%%%% 3: Find the Strongly Connected Comp.
SCC = stronglyconn(spng, 1); % each row is an SCC
SCC
prnscc(SCC, V);   % print SCCs
```

Digraph:
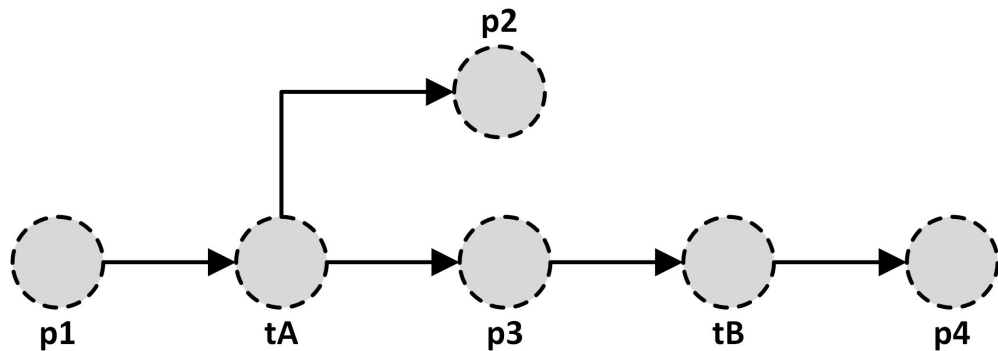
Figure 6.2: Digraph converted from the Petri net.

Fig.6.2 shows the digraph generated from the Petri net. The function `convert_PN_V` returns a structure **V**, which possesses two fields: 1) **V.nodes**: this is an array of node names (text strings) 2) **V.A**: the adjacency matrix of the digraph.

The digraph's adjacency matrix is basically a makeover of the Petri net's extended incidence matrix (A_e). A_e consists of two parts, namely the input incidence matrix (A_i) and the output incidence matrix (A_o). Digraph's adjacency matrix is also composed of (A_i) and the transpose of (A_o), in addition to two zero matrices, as shown in Fig.6.3.

Cycles:
The Petri net (Fig.6.1) does not have any cycles (acyclic). Hence, the function `cycles` returns an empty structure.

Strongly Connected Components:
Due to the lack of cycles in the Petri net (Fig.6.1), each element in the Petri net is a component on its own. Hence, the function `stonglyconn` returns a
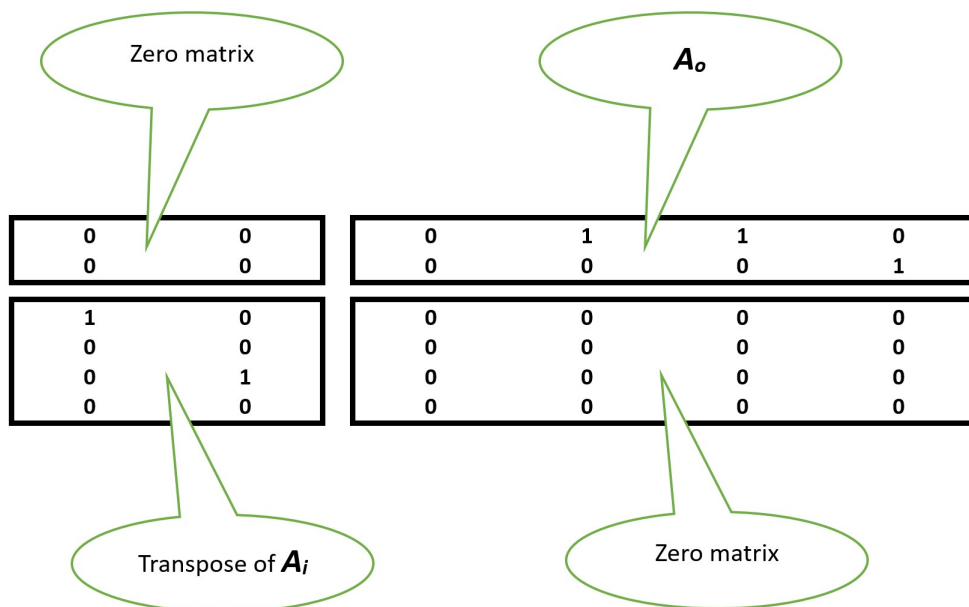
Figure 6.3: Digraph's Adjacency matrix as a composition of the Petri net's A_e.

matrix in which just one element in each raw is non-zero (non-zero elements
in each raw represent a connected component).

```
SCC =

6×6 logical array

0 0 0 0 0 1
0 0 0 0 1 0
0 0 0 1 0 0
0 0 1 0 0 0
0 1 0 0 0 0
1 0 0 0 0 0
```

6.7   Example-21: cycles
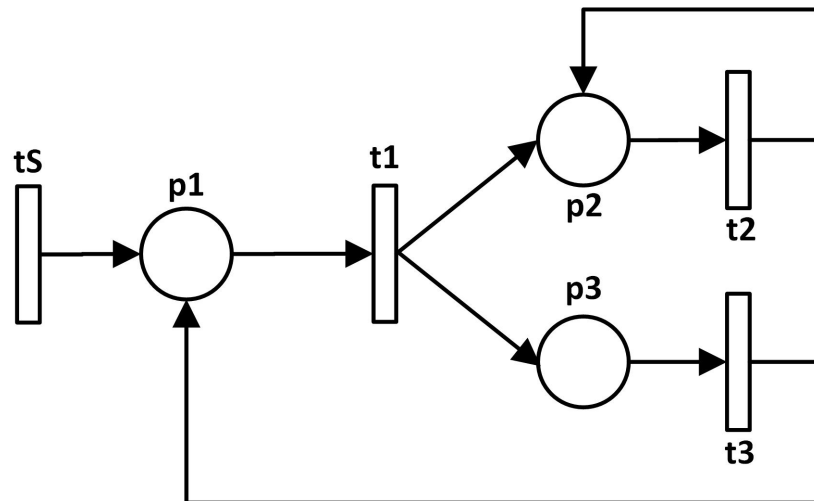
Figure 6.4: A Petri net with two cycles.

Fig.6.4 shows a Petri net with two cycles. Listing-6.2 presents the MSF.

Listing 6.2: **MSF** (Example-21)

```
clear all; clc; close all;
spng = pnstruct('ex_21_pdf');

%%%%% 1: Convert Petri net into digraph V
V = convert_PN_V(spng);
```

```
%%%%% 2: Find the cycles in V
V = cycles(V);
V.cycles
prncycles(V);

%%%%% 3: Find the Strongly Connected Comp.
SCC = stronglyconn(spng, 1); % each row of SCC is a
SCC
prnscc(SCC, V);
```

The printout below shows the **V.cycles** as a matrix of dimension $(2X7)$. The two rows indicate there are two cycles:

○ Cycle 1 (row 1) consists of second and sixth elements;
○ Cycle 2 (row 2) consists of third, fifth, first, and seventh elements.

Note that each row is padded with trailing zeros.

```
ans =

2 6 0 0 0 0 0
3 5 1 7 0 0 0
```

The printout shown below is by the function `prncycles`, which takes **V.cycles** and prints the cycles with the names of the cycle members.

```
Cycle No-1:   *******
-> t2 -> p2

Cycle No-2:   *******
-> t3 -> p1 -> t1 -> p3
```

The printout below shows the **SCC** as a matrix of dimension $(3X7)$. The three rows indicate there are three strongly connected components:

○ Component 1 (row 1) consists of just one element (fourth);
○ Component 2 (row 2) consists of two elements (second and sixth);
○ Component 3 (row 3) consists of four elements (first, third, fifth, and seventh elements).

```
SCC =

3×7 logical array

0 0 0 1 0 0 0
0 1 0 0 0 1 0
1 0 1 0 1 0 1
```

Finally, function `prnscc` takes the **SCC** matrix and printout the components with their elements names:

```
Component No-1:  *******
{ tS }

Component No-2:  *******
{ t2 - p2 }

Component No-3:  *******
{ t1 - t3 - p1 - p3 }
```

Bibliography

Davidrajuh, R. (2018). *Modeling Discrete-Event Systems with GPenSIM*, Springer International Publishing, Cham.

Davidrajuh, R. (2021). *Petri Nets for Modeling of Large Discrete Systems*, Springer.

Chapter 7

Initial Dynamics

The function `initialdynamics` combines the static Petri Net graph and the initial dynamics to create the Petri set structure with initial dynamics ('pni'). pni is the one that is used to start the simulation by the function `gpensim`. pni is also the input to `cotree` to create the reachability (coverability) tree.

7.1 initialdynamics

Function name: `initialdynamics`
Full name: Assign Initial Dynamics.
Purpose: create the Petri net structure with initial dynamics by combining the static Petri Net graph (`spng`) and the initial dynamics (e.g., initial tokens, firing times, resources, and costs).
Input Parameter: Static Petri Net graph (`spng`, output of the functions `pnstruct`) and initial dynamics.
Output Parameter: Petri net structure with initial dynamics (**pni**); pni can be input to the function `gpensim`, to start the simulation.
This function uses: (many sub-functions).
This function is used by: [in Main Simulation File (MSF)]
Further info: chapter 2, "Modeling with GPenSIM: Basic Concepts," in Davidrajuh (2018).
Sample use:

```
spng = pnstruct('simple_pn_pdf');
dyn.m0 = {'p1',3, 'p2',4};
pni = initialdynamics(spng, dyn);
Sim_Results = gpensim(pni); % perform simulation runs
```

Bibliography

Davidrajuh, R. (2018). *Modeling Discrete-Event Systems with GPenSIM*, Springer International Publishing, Cham.

Chapter 8

Is Functions

This chapter describes some utility functions that can help verify the type of an entity, e.g., is 'x1' a transition? Is the Petri net a strongly connected component? These functions are summarized in Table-8.1.

Function	Description
is_enabled	Is this transition currently enabled?
is_eventgraph	Is this Petri net an event graph (*aka* marked graph)?
is_firing	Is this transition curretly firing?
is_place	Is this element a place?
is_stronglyconn	Is this Petri net strongly connected?
is_trans	Is this element a transition?

Table 8.1: Summary of Is functions.

8.1   is_enabled

Function name: is_enabled
Full name: Is the transition enabled?
Purpose: to check whether a transition is enabled right now.
Input Parameter: transition's name or index.
Output Parameter: (Boolean) true or false.
This function uses: is_transition: whether the given entity is a transition or not.
Sample use:

```
function [fire, trans] = tBeta_pre(trans)
% in the specific pre-preprocessor of "tBeta"

% allow tBeta to fire only if tAlfa is not enabled
```

```
fire = not(is_enabled('tAlfa'));
```

Related function: `is_firing`

8.2   is_eventgraph

Function name: `is_eventgraph`
Full name: Is the Petri net an event graph?
Purpose: to check whether the Petri net is an Event Graph (*aka* Marked Graph).
Input Parameter: Static Petri net graph (`spng`, output of the function `pnstruct`) or Petri net structure with initial dynamics (`pni`, output of the function `initialdynamics`).
Output Parameter: (Boolean) true or false.
This function uses: (none).
Further info: Chapter 7, "Performance Evaluation of Discrete Event Systems," in Davidrajuh (2018).
Sample use:

```
spng = pnstruct('simple_pn_pdf');
if is_eventgraph(spng)
    disp('This Petri net is an Event (Marked) Graph!');
end
```

8.3   is_firing

Function name: `is_firing`
Full name: Is this transition currently firing?
Purpose: to check whether a transition is firing right now.
Input Parameter: transition's name or index.
Output Parameter: (Boolean) true or false.
This function uses: `is_transition`: whether the given entity is a transition or not.
Sample use:

```
function [fire, trans] = tBeta_pre(trans)
% in the specific pre-preprocessor of tBeta

% allow tBeta to fire only if tAlfa is not firing now
fire = not(is_firing('tAlfa'));
```

Application example: A simple example ("Example-23: is_firing" in Section 8.8) is given at the end of this chapter.
Related function: `is_enabled`

8.4 is_place

Function name: `is_place`
Full name: Is this entity a place?
Purpose: to check whether a named entity is a place.
Input Parameter: entity's name.
Output Parameter: 0: Not a place; > 0 : index of place.
This function uses: (none).
Sample use:

```
pIndex = is_place('p22'); % is "p22" a place?
if pIndex
   disp(['"p22" is a place, with place index ', int2str(pIndex)]);
end
```

Application example: A simple example ("Example-22: Is Functions" in Section 8.7) is given at the end of this chapter.
Related function: `is_trans`

8.5 is_stronglyconn

Function name: `is_stronglyconn`
Full name: Is this Petri net a Strongly Connected Component?
Purpose: To check whether the given Petri net is a Strongly Connected Component.
Input Parameter: Static Petri net graph (`spng`, output of the function `pnstruct`) or Petri net structure with initial dynamics (`pni`, output of the function `initialdynamics`).
Output Parameter: (Boolean) true or false.
This function uses: `stronglyconn`: to extract the connected components of the Petri net.
Further info: Chapter 11, "Discrete Systems as Petri Modules," in Davidrajuh (2021).
Sample use:

```
spng = pnstruct('simple_pn_pdf');
if is_stronglyconn(spng)
   disp('This Petri net is one component!');
end
```

Related functions: `stronglyconn`, `cycles`

8.6   is_trans

Function name: `is_trans`
Full name: Is this entity a transition?
Purpose: to check whether a named entity is a transition.
Input Parameter: entity's name.
Output Parameter: 0: Not a transition; > 0 : index of transition.
This function uses: (none).
Sample use:

```
tIndex = is_trans('t44'); % Is 't44' a transition
if tIndex
   disp(['"t44" is a transition with transition index ', ...
       int2str(tIndex)]);
end
```

Application example: A simple example ("Example-22: Is Functions" in Section 8.7) is given at the end of this chapter.
Related function: `is_place`

8.7   Example-22: Is Functions

This example is almost identical to example 14 (Section 1.5). In example 14, we used "Check Valid Functions," which throws an error and terminates the program (returns control to the MATLAB Command Prompt) if the input name is not valid. In example 22, we use 'softer' "Is functions" instead!

Let us imagine that a main simulation file has just completed a simulation. We may want to check the number of leftover tokens in various places or how many times the individual transitions have fired. Listing-8.1 shows that the simulation is complete in a main simulation file followed by two subroutines ('check_place' and 'check_trans').

Listing 8.1: **Part of the Main Simulation File** (Example-22)

```
%
% any main simulation file is fine!
%
...
...

sim = gpensim(pni);

% simulation if complete
check_place(); % number of tokens in a place
check_trans(); % number of times a trans has fired
```

Subroutine 'check_place' (Listing-8.2) repeatedly ask a user to input a valid place name and then checks how many tokens are left in that place. Similarly, subroutine 'check_trans' (Listing-8.3) repeatedly ask a user to input a valid transition name and then checks how many times this transition has fired. Both of these functions use the "Is functions."

Listing 8.2: **check_place** (Example-22)

```
function [] = check_place()
prompt = ['\nEnter a valid *place* name without using single ...
    quotation marks \n', ...
  '(note that wrong place name can crash the program)\n',...
  'press return key to quit:  '];
reply = 'pSomething';

while not(isempty(reply))
    reply = input(prompt, 's');
    if isempty(reply), return; end

    % is this a valid place name?
    if is_place(reply)
        % find the number of tokens
        ntok = ntokens(reply);
        disp(' ');
        disp(['"',reply, '" has ', int2str(ntok),...
            ' tokens now.']);
    end
end
```

Listing 8.3: **check_trans** (Example-22)

```
function [] = check_trans()

prompt = ['\nEnter a valid !transition! name without using ...
    single quotation marks \n', ...
  '(note that wrong transition name can crash the program)\n',...
  'press return key to quit:  '];

reply = 'tSomething';

while not(isempty(reply))
    reply = input(prompt, 's');
    if isempty(reply), return; end

    % is this a trans name?
    if is_trans(reply)
        % find the number of times fired
        nfired = timesfired(reply);
        disp(' ');
        disp(['"',reply, '" has fired ', int2str(nfired),...
            ' times.']);
```

```
    end
end
```

8.8   Example-23: is_firing
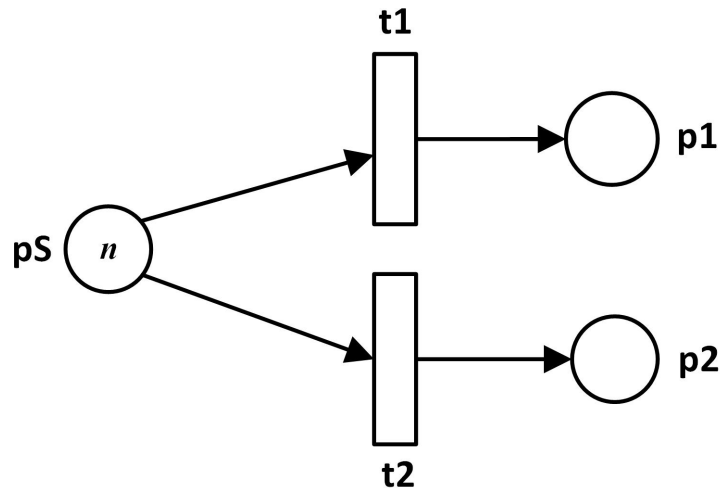
Figure 8.1: A Petri net with two transitions.

In Fig.8.1, only one transition fires at a time (either **t1** or **t2**):
In this tiny example, we shall make sure that either **t1** or **t2** fires, and
not both at the same time. This condition can be realized by using a global
variable ('semaphore') or by putting a place with a token for mutual exclusion
between the two transitions.

However, in this example, we shall attempt another technique using the
function 'is_firing.' Since both transitions are enabled, we will allow one
of these two to start firing randomly. After that, any enabled transition can
start firing only if the other is not firing.

Listing-8.4 shows the main simulation file, whereas Listings 8.5 and 8.6
shows the specific pre-processors for **t1** and **t2** (there is no need for post-
processors).

Listing 8.4: **MSF** (Example-23)

```
clear all; clc; close all;
global global_info
global_info.STOP_AT = 100;

spng = pnstruct('ex_23_pdf');
```

```
dyn.m0 = {'pSTART', 10};
dyn.ft = {'t2',20, 't1',10};
pni = initialdynamics(spng, dyn);

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

Listing 8.5: **t1_pre** (Example-23)

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

% t1 can fire if t2 is not firing
fire = not(is_firing('t2'));
```

Listing 8.6: **t2_pre** (Example-23)

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

% t2 can fire if t1 is not firing
fire = not(is_firing('t1'));
```

Bibliography

Davidrajuh, R. (2018). *Modeling Discrete-Event Systems with GPenSIM*, Springer International Publishing, Cham.

Davidrajuh, R. (2021). *Petri Nets for Modeling of Large Discrete Systems*, Springer.

Chapter 9

Performance Metrics

This chapter presents three functions for performance analysis. The function `mincyctime` can be applied only for Marked Graphs (*aka* Event Graphs); a Marked Graph is a special type of Petri net in which all the places have exactly one input and one output transition. The other two functions, `extractt` and `occupancy`, summarize the firings of transitions. Table-9.1 presents a summary of these functions.

Function	Description
`extractt`	returns a matrix with three columns: [firing-transition, start-time, stop-time].
`mincyctime`	finds the "minimum-cycle-time (MCT)" of a marked graph.
`occupancy`	extracts start-times and stop-times of transition firings from the simulation results.

Table 9.1: Summary of functions for Performance Metrics.

9.1   extractt

Function name: `extractt`
Full name: Extract transitions firings information.
Purpose: After simulation, extract the start and finish of firing times of different transitions.
Input Parameters: 1) Simulation results (output of the function `gpensim`); 2) set of transitions (names or indices).
Output Parameter: `DURATION` matrix : A matrix containing three columns:

1. Column-1: The firing transition (index of the transition).
2. Column-2: firing start time.

3. Column-3: firing finishing time.

This function uses: `check_valid_transition`: to check whether the input transition (name or index) is valid.
Further info: chapter 10, "Performance-Metrics," in Davidrajuh (2018)
Sample use:

```
...
sim_results = gpensim(pni);
[DURATION] = extractt(sim_results, {'trans1', 'trans2'})
```

Related function: `occupancy`

9.2   mincyctime

Function name: `mincyctime`
Full name: Minimum Cycle Time.
Purpose: This function is applicable only for marked graphs (*aka* event graphs). This function will analyze all the cycles and print the results pointing out the minimum cycle (the "bottleneck").
The Minim-Cycle-Time = max(cycle Delay/tokens in the cycle)
If an optional target for throughput is given as the second input parameter, this function will also suggest the remedies for achieving the requested throughput.
Input Parameter - compulsory: Petri net structure with initial dynamics (`pni`) - the output of the function `initialdynamics`
Input Parameter - optional: wanted throughput.
Output Parameter: structure (V) with all the cycles.
This function uses: `is_eventgraph`, `cycles`.
Further info: See chapter 7.2, "Minimum Cycle Time in Marked Graphs," in Davidrajuh (2018).
Sample use:

```
...
pni = initialdynamics(spng, dyn);
V = mincyctime(pni);
```

Application example: A simple example ("Example-24: mincyctime" in Section 9.4) is given at the end of this chapter.

9.3   occupancy

Function name: `occupancy`
Full name: Extract a summary of transitions firings information.

Purpose: After simulation, extract a summary of different transitions' firing times (total firing times and in percentage).

Input Parameter: 1) Simulation results (output of the function `gpensim`); 2) set of transitions (names or indices).

Output Parameter: Output: 1. `OCCUPANCY` matrix : 1st row [time taken by each transition] and 2nd row:[active time in percentage]

Output: 2. `DURATION` matrix (see 9.1): A matrix containing three columns: [firing-transition, start-time, stop-time].

This function uses: `extractt`: to extract `DURATION` matrix; `check_valid_transition`: to check whether the input transition (name or index) is valid.

Further info: chapter 10, "Performance-Metrics," in Davidrajuh (2018).

Sample use:

```
...
sim_results = gpensim(pni);
[OCCUPANCY, DURATION] = occupancy(sim_results, {'t1', 't2'});
```

Related function: `extractt`

9.4 Example-24: mincyctime

Fig.9.1 shows a marked graph (*aka* marked graph) as all the places in that Petri net have exactly one input and one output transition. This marked graph also has four cycles. Let us assume that the firing times of all transitions are 1 minute.

First, we will find the throughput of the Petri net (tokens/minute) using the function `mincyctime`. Then, suppose we want a higher throughput, say 0.5 tokens/minute; we can feed this value as the optional input and get some ideas from `mincyctime` on how this higher throughput can be achieved.

Listing-9.1 shows that main simulation file.

Listing 9.1: **MSF** (Example-24)

```
clear all; clc; close all;

spng = pnstruct('ex_24_pdf');
dyn.ft = {'allothers',1}; % firing times of all: 1 minute
dyn.m0 = {'p6',1, 'p4',1, 'p7',1,'p9',1};
pni = initialdynamics(spng, dyn);

% expected flowrate is 0.5 tokens/minute
mincyctime(pni, 0.5);
```

The function `mincyctime` prints out the following information on the number of cycles in the marked graph, the dominating cycle (minimum cy-
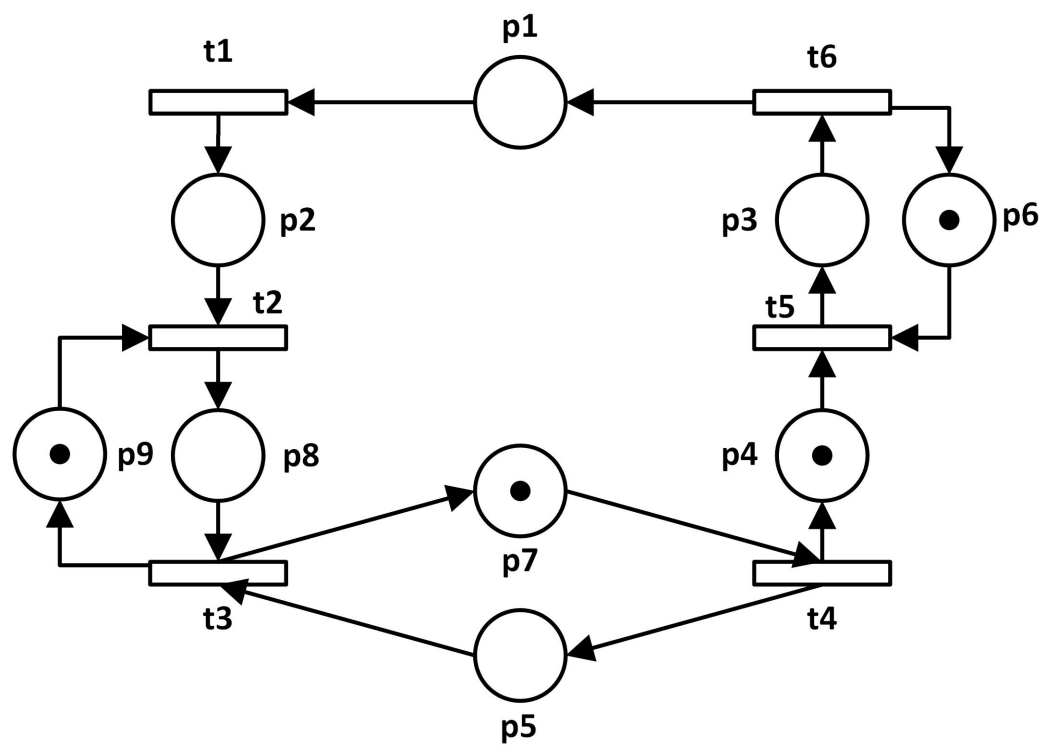
Figure 9.1: A Marked Graph with four cycles.

cle or the bottleneck), and the current throughput (which is 0.33 tokens/minute).

```
This is a Strongly Connected Petri net.

Cycle-1:  -> p8 -> t3 -> p7 -> t4 -> p4 -> t5
-> p3 -> t6 -> p1 -> t1 -> p2 -> t2
TotalTD = 6 TokenSum = 2 Cycle Time = 3

Cycle-2:  -> p3 -> t6 -> p6 -> t5
TotalTD = 2 TokenSum = 1 Cycle Time = 2

Cycle-3:  -> p7 -> t4 -> p5 -> t3
TotalTD = 2 TokenSum = 1 Cycle Time = 2

Cycle-4:  -> t2 -> p8 -> t3 -> p9
TotalTD = 2 TokenSum = 1 Cycle Time = 2


Minimum-cycle-time is:  3, in cycle number-1


*** Token Flow Rate:  ***
In a steady state, the firing rate of each
transition is:  1/C* = 0.33333
meaning, on average, 0.33333 tokens go through
any node in the Petri net per unit of time.
```

Further, `mincyctime` suggests the following on how we can increase the throughput from 0.33 to 0.5 tokens/minute.

```
*** We can increase the current flow rate to 0.5
tokens/TU by improving the critical circuit alone
...
In the circuit-1 either:
1.  increase the sum of tokens by one token, or
2.  decrease the total delay (firing times) by 2
TU.
```

Bibliography

Davidrajuh, R. (2018). *Modeling Discrete-Event Systems with GPenSIM*, Springer International Publishing, Cham.

Chapter 10

Petri Net Structure

The first three functions presented in this chapter, namely `createPDF`, `pnstruct`, and `matrixD`, are used to develop a static Petri Net graph. The final two functions, `postset` and `preset`, return the inputs and outputs of elements. Table-10.1 presents a summary of these functions.

Function	Description
`createPDF`	creates a Petri net Definition File (PDF) from the inputs - incidence matrices Ai and Ao.
`matrixD`	returns incidence matrices $[Dm, D, Dp]$ extracted from GPenSIM's extended incidence matrix (A_e, `PN.incidence_matrix`).
`pnstruct`	creates a MATLAB structure from the static Petri Net graph based on the Petri Net Definition File (PDF).
`postset`	returns the postset of an element (or set of elements).
`preset`	returns the preset of an element (or set of elements).

Table 10.1: Summary of functions for Petri Net Structure.

10.1   createPDF

Function name: `createPDF`
Full name: create a Petri net Definition File (PDF).
Purpose: This function will create a PDF file from input and output incidence matrices Ai and Ao.

Input Parameters - compulsory: 1) The input incidence matrix (A_i); 2) the output incidence matrix (A_o).

Input Parameters - optional: 3) Name of the output PDF file. 4) A label for the Petri Net.

Output Parameter: A Petri net Definition file (PDF) file will be created and stored in the current folder.

This function uses: (none).

Further info: See chapter 8.3, "Avoiding PDF Files," in Davidrajuh (2018)

Sample use:

```
Ai   =   [ 0      1      0      0      0      0; ...
           1      0      0      0      0      1; ...
           0      0      0      1      1      0; ...
           0      0      1      0      0      0];

Ao =     [ 1      0      0      0      1      0; ...
           0      1      0      0      0      0; ...
           0      0      1      0      0      0; ...
           0      0      0      1      0      1];
PDF_filename = 'mg1_pdf'; % without ending ".m"
PN_name = 'Marked Graph example';
createPDF(Ai, Ao, PDF_Filename, PN_name);
```

Application example: A simple example ("Example-25: createPDF" in Section 10.6) is given at the end of this chapter.

10.2   matrixD

Function name: `matrixD`

Full name: Matrices D^+, D, and D^-.

Purpose: This function will extract the incidence matrices (D^+: output incidence matrix; D^-: input incidence matrix; D: (net) incidence matrix) from GPenSIM's **extended incidence matrix $\mathbf{A_e}$**; Matrix D is used in textbooks on Petri Nets (e.g., Moody and Antsaklis (2012)).

Note that D^+ is the same as $\mathbf{A_o}$ and D^- is the same as $\mathbf{A_i}$ in the previous Section 10.1.

Input Parameters: (none).

Output Parameters: Three matrices: 1) Matrix D^+; 2) Matrix D; 3) Matrix D^-.

This function uses: (some sub-functions).

Sample use:

```
% in MSF:
spng = pnstruct('sample_pdf');
```

```
[Dp, D, Dm] = matrixD();
```

Application example: A simple example ("Example-26: matrixD" in Section 10.7) is given at the end of this chapter.

10.3 pnstruct

Function name: pnstruct
Full name: Static Petri Net Structure.
Purpose: This function creates the static Petri Net graph structure (spng) that represents the static Petri Net graph.
Input Parameters: One or more Petri Net Definition Files (PDFs).
Output Parameter: A MATLAB structure (spng) is a compact representation of the static Petri net graph.
There are 13 fields in spng:

1. name: a text string or label given in the PDF(s).
2. global_places: this is a structure array representing the places.
3. No_of_places (n_p): Total number of places in the Petri net.
4. global_transitions: this is a structure array of transitions.
5. No_of_transitions (n_t): Total number of transitions in the Petri net.
6. global_Vplaces: place structures for all the places.
7. incidence_matrix: the extended incidence matrix (A_e). Dimension is ($n_t \times 2 \cdot n_p$).
8. Inhibitors_exist: (Boolean value) Whether or not we have inhibiting arcs in the Petri net.
9. Inhibited_Transitions: A vector of Boolean values (length n_t) indicating which transitions are inhibited (identified by the transition index).
10. inhibitor_matrix: A matrix that represents the inhibiting arcs. Dimension: ($n_t \times n_p$). Each row represents a transition, and each column a place.
11. No_of_modules (n_M): Number of Petri Modules in the Petri net.
12. module_membership: A matrix of dimension ($2 \times n_t$). The first row of the matrix shows the membership of each transition. The second row indicates whether a transition is an input or output port.
13. module_names: Set of all the module names.

This function uses: (some sub-functions).
This function is used by: [in Main Simulation File]
Sample use:

```
% in MSF:
% create the structure for
%     static Petri net graph
spng = pnstruct('simple_pn_pdf');
```

10.4   postset (new in version 11)

Function name: `postset`
Full name: postset.
Purpose: This function returns the set of output elements of an element (or set of elements).
Input Parameter: set of elements (either places or transitions), identified by their name.
Output Parameter: 1) Set of postset elements' names (set of ASCII text string) 2) Set of postset elements' indices.
Output to Screen: The postset names will be displayed on the screen.
This function uses: `get_outputplaces`, `get_outputtrans`
Sample use:

```
postset(preset({'p1'}));
postset(postset({'t2'}));
```

Application example: A simple example ("Example-27: preset and post-set" in Section 10.8) is given at the end of this chapter.
Related functions: `preset`, `get_outputplaces`, `get_outputtrans`

10.5   preset (new in version 11)

Function name: `preset`
Full name: preset.
Purpose: This function returns the set of input elements of an element (or set of elements).
Input Parameter: set of elements (either places or transitions), identified by their name.
Output Parameter: 1) Set of preset elements' names (set of ASCII text string) 2) Set of preset elements' indices.
Output to Screen: The preset names will be displayed on the screen.
This function uses: `get_inputplaces`, `get_inputtrans`
Sample use:

```
% get the preset of "t5" and "t6"
```

```
[preset_Names, preset_Indices] = preset({'t5', 't6'});
% get the preset of preset of "t5" and "t6"
[preset_Names, preset_Indices] = preset(preset({'t5', 't6'}));
```

Application example: A simple example ("Example-27: preset and post-set" in Section 10.8) is given at the end of this chapter.
Related functions: `postset`, `get_inputplaces`, `get_inputtrans`

10.6   Example-25: createPDF
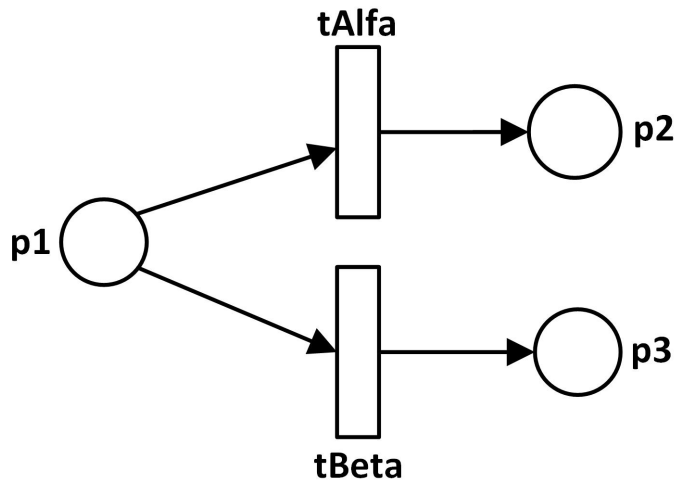
Figure 10.1: Petri net for testing the function `createPDF`.

Fig.10.1 shows a tiny Petri net with three places and two transitions. The input and output incidence matrices (A_i and A_o) of this Petri net:

$$\mathbf{A_i} = \begin{array}{c} \\ tAlfa \\ tBeta \end{array} \begin{array}{ccc} p1 & p2 & p3 \\ \left(\begin{array}{ccc} 1 & 0 & 0 \\ 1 & 0 & 0 \end{array} \right) \end{array}$$

$$\mathbf{A_o} = \begin{array}{c} \\ tAlfa \\ tBeta \end{array} \begin{array}{ccc} p1 & p2 & p3 \\ \left(\begin{array}{ccc} 0 & 1 & 0 \\ 0 & 0 & 1 \end{array} \right) \end{array}$$

Rather than creating a PDF file, we are going to use the incidence matrices and ask the function `createPDF` to create the PDF automatically, as shown in the MSF (Listing-10.1).

Listing 10.1: **Main Simulation File** (Example-25)

```matlab
clear all; clc; close all;

% input incidence matrix
Ai = [1 0 0; ...
      1 0 0];

% output incidence matrix
Ao = [0 1 0; ...
      0 0 1];

% optional: name for the PDF to be created
PDF_Filename = 'ex_25_pdf'; % without ending ".m"

% optional: name for the Petri net
PN_name = 'Example-25: Testing automatic PDF creation';

% call the functio to create the PDF file
createPDF(Ai, Ao, PDF_Filename, PN_name);
```

The PDF (named 'ex_25_pdf.m') that is automatically created by the function `createPDF` is given below.

```
 % This PDF file was generated by "createPDF"
function on
% On 09-Jul-2024 at 14:27:18
% PDF: ex_25_pdf.m


function [png] = ex_25_pdf.m()


png.PN_name = 'Example-25:  Testing automatic
PDF creation';
png.set_of_Ps = {'p1','p2','p3'};
png.set_of_Ts = {'t1','t2'};
png.set_of_As = {'p1','t1',1,'t1','p2',1,...  %t1
'p1','t2',1, 't2','p3',1, ...  %t2
};
```

10.7   Example-26: matrixD

Fig.10.2 presents a Petri net with 13 elements. Hence, we can use the function `matrixD` to compute the incidence matrices, as doing it by hand could be clumsy.

Listing-10.2 shows that the input incidence matrix (A_i) and the output incidence matrix (A_o) matrix can be directly extracted from the structure for static Petri net graph (`spng`) (or from `pni` - the Petri net structure with initial dynamics or `PN` - the Petri net run-time structure). However, the
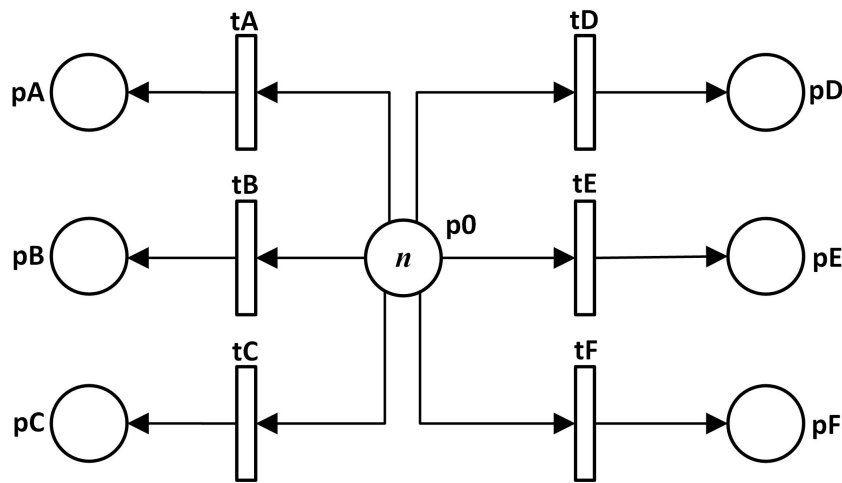
Figure 10.2: Petri net for testing the function `matrixD`.

function `matrixD` offers an easy high-level alternative.

Listing 10.2: **Main Simulation File** (Example-26)

```
clear all; clc; close all;
spng = pnstruct('ex_26_pdf');

%%%%% Extract the "D" matrices
[Dp, D, Dm] = matrixD();

np = nplaces(); % number of places
% Ae: extended incidence matrix
Ae = spng.incidence_matrix;
% Ai : input incidence matrix
Ai = Ae(:, np+1:end);
% Ao : output incidence matrix
Ao = Ae(:, 1:np);

if all(all(eq(Dm, Ai)))
    disp(['Input incidence matrices "Dm" ' ...
        'and "Ai" are the same!']);
end

if all(all(eq(Dp, Ao)))
    disp(['Output incidence matrices "Dp" ' ...
        'and "Ao" are the same!']);
end
```
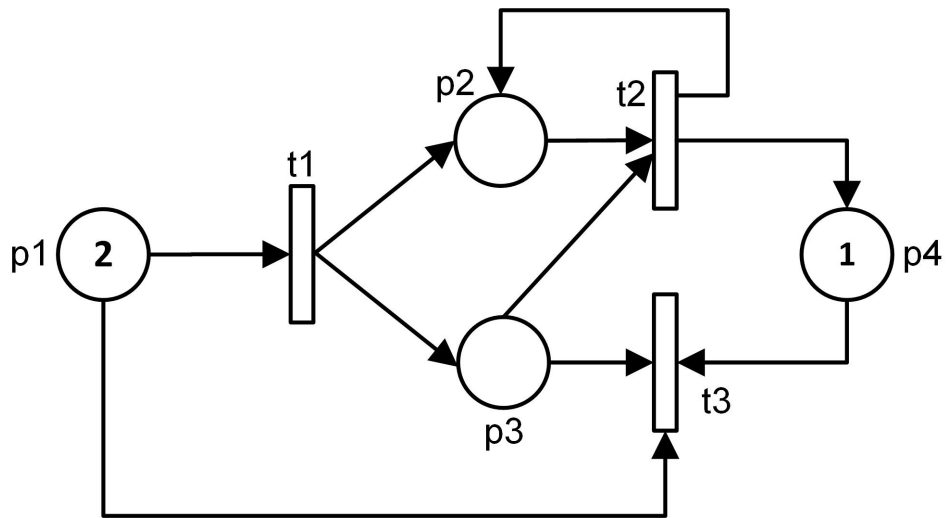
Figure 10.3: Petri net for testing functions postset.

10.8   Example-27: preset and postset

Fig.10.3 shows a Petri net for testing the `postset` and `preset` functions.

Listing 10.3: **Main Simulation File** (Example-27)

```
clear all; clc; close all;
spng = pnstruct('ex_27_pdf');

%%%%% Testing preset and postset   %%%%%
preset({'p1'});
preset({'t1','t2','t3'});
postset(preset({'p1'}));
postset(postset({'t2'}));
```

In this main simulation file (Listing-10.3), we are performing four sets of tests. The first statement finds the preset of **p1**, which is nothing, as **p1** is a source.

```
preset of "p1" :
-- none ---
```

The second statement finds the preset of three transitions: preset of **t1** is **p1**; preset of **t2** is **p2** and **p3**. preset of **t3** is **p1**, **p3**, and **p4**. Put together, the result is **p1**, **p2**, **p3**, and **p4**.

```
preset of "t1" "t2" "t3" :
"p1", "p2", "p3", "p4"
```

The third statement finds the postset of the preset of **p1**. Since preset is **p1** is nothing, nothing's postset is also nothing.

```
preset of "p1" :
-- none ---


postset of nothing is nothing ....
```

The final statement finds the postset of postset of **t2**. postset of **t2** is **p2** and **p4**. postset of **p2** is **t2** and postset of **p4** is **t3**. Hence, the result is **t2** and **t3**.

```
postset of "t2" :  "p2", "p4"

postset of "p2" "p4" :
"t2", "t3"
```

Bibliography

Davidrajuh, R. (2018). *Modeling Discrete-Event Systems with GPenSIM*, Springer International Publishing, Cham.

Moody, J. O. and Antsaklis, P. J. (2012). *Supervisory control of discrete event systems using Petri nets*, Vol. 8, Springer Science & Business Media.

Chapter 11

Plotp

This chapter presents only one function, namely `plotp`.

11.1   plotp

Function name: `plotp`
Full name: Plot tokens in places versus time.
Purpose: After simulation (by the function `gpensim`), this function will plot the number of tokens in various places.
Input Parameters: 1) Simulation results (output of the function `gpensim`); 2) Set of places.
Output Parameter: `TOKEN` matrix: A simpler matrix that shows the time and the state (marking). `TOKEN` matrix shows only time versus (partial) state info, the tokens in specific places (set of places in the input parameter to `plotp`). Note that to get the full state info (tokens in all the places), we must enter the complete set of place names as the input parameter to function `plotp`.
Output on screen: a graphical plot.
This function uses: `extractp`: To extract tokens from the simulation results structure.
This function is used by: [in Main Simulation File]
Further info: chapter Chapter 2, "Modeling with GPenSIM: Basic Concepts," in Davidrajuh (2018).
Sample use: See **Example-28 (Section 11.2)**.
Related function: `prnss`

11.2   Example-28: plotp

Let us say that in the Petri net shown in Fig.11.1, we are only interested in how the tokens in places **pStep2** and **pEnd** change in comparison to each other. Hence, in the main simulation file (Listing-11.1), we will plot the
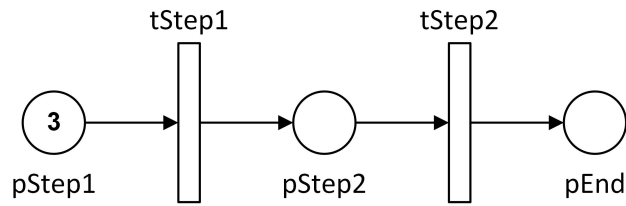
Figure 11.1: Graphical plot by the function `plotp`.

tokens in these two places using the function `plotp`; also, we will obtain the
TOKEN matrix from the function `plotp` for further analysis.

Listing 11.1: **Main Simulation File** (Example-28)

```
clear all; clc; close all;
global global_info
global_info.STOP_AT = 10;

spng = pnstruct('ex_28_pdf');
dyn.m0 = {'pStep1', 6};
dyn.ft = {'tStep1',2, 'tStep2',5};
pni = initialdynamics(spng, dyn);

sim = gpensim(pni);
% plot tokens and display TOKENs
TOKENS = plotp(sim, {'pStep2','pEnd'})
```

Fig.11.2 shows the plot, and the screen dump showing the contents of
the matrix TOKEN is given below. Note that the first row of TOKEN matrix
is the header; it indicates that **pStep2**'s index is three and **pEnd**'s index
is 1. For the rest of the rows, the first column represents the time series,
and the second and third columns are the tokens in **pStep2** and **pEnd**.
Section 2.1 gives more details on the composition of TOKEN matrix. TOKEN
matrix possesses the same information shown in Fig.11.2.
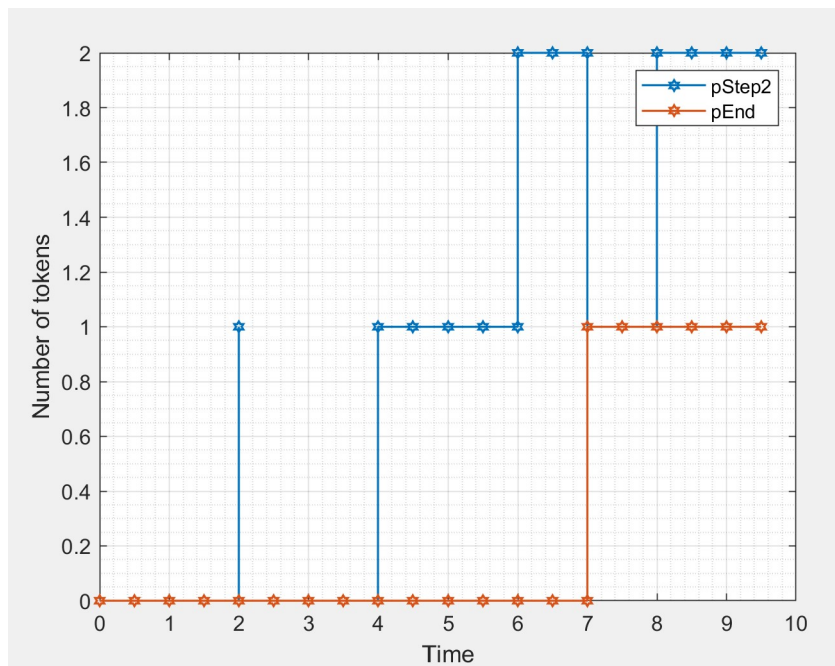
Figure 11.2: Graphical plot by the function `plotp`.

```
TOKEN matrix =
0 3.0000 1.0000
0 0 0
0 0 0
0.5000 0 0
1.0000 0 0
1.5000 0 0
2.0000 0 0
2.0000 1.0000 0
2.0000 0 0
2.5000 0 0
3.0000 0 0
3.5000 0 0
4.0000 0 0
4.0000 1.0000 0
4.0000 1.0000 0
4.5000 1.0000 0
5.0000 1.0000 0
5.5000 1.0000 0
6.0000 1.0000 0
6.0000 2.0000 0
6.0000 2.0000 0
6.5000 2.0000 0
...
```

Bibliography

Davidrajuh, R. (2018). *Modeling Discrete-Event Systems with GPenSIM*, Springer International Publishing, Cham.

Chapter 12

PNCT Functions

The Petri Net Control Toolbox (PNCT) was developed at the University of Cagliari. PNCT offers simple and crude functionality for plotting a reachability (section 2.1) tree and for structural analysis (chapter 20). GPenSIM extends five PNCT functions to provide more understandable information:

1. PNCT function *plottree* is used by GPenSIM function `cotree` to plot the reachability tree.
2. PNCT functions *siphons*, *tinvar*, *pinvar*, and *traps* are extended by GPenSIM to provide functions for structural analysis with the same function names.
3. GPenSIM function `gpensim_2_PNCT` converts GPenSIM's Extended Incidence Matrix A_e into PNCT incidence matrices Pre_A and Post_A.

12.1   gpensim_2_PNCT

Function name: `gpensim_2_PNCT`
Full name: GPenSIM to PNCT format.
Purpose: This function converts GPenSIM's extended incidence matrix A_e to PNCT incidence matrices "Pre_A" and "Post_A".
Input Parameter: GPenSIM's incidence matrix A_e (e.g., `spng.incidence_matrix` or `pni.incidence_matrix` or `PN.incidence_matrix`).
Output Parameters: Thee incidence matrices in PNCT format: [Pre_A, Post_A, D] (where D= Post_A, Pre_A).
This function uses: (none)
Further info: Chapter 9, "Structural Invariants," in Davidrajuh (2018) presents some basic material.
Sample use:

```
% in MSF:
spng = pnstruct('simple_pn_pdf');
```

```matlab
dyn.m0 = {'p1',15, 'p2',13};
dyn.ft = {'t1',10};
pni = initialdynamics(spng, dyn);

A = spng.incidence_matrix;
% or A = pni.incidence_matrix;

[Pre_A, Post_A, D] = gpensim_2_PNCT(A)
```

Application example: A simple example (Example-29) is given below.

12.2   Example-29: gpensim_2_PNCT
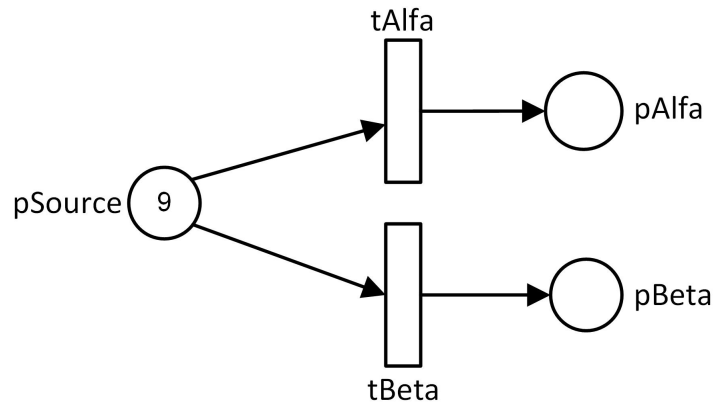
Figure 12.1: Petri net for testing the function `gpensim_2_PNCT`.

Fig.12.1 presents a simple Petri net in which we can either use the function `matrixD` to compute the incidence matrices or do it by hand.

Listing-12.1 shows that the transpose of input incidence matrix (A_i) is PNCT's 'Post_A' and transpose of output incidence matrix (A_o) is PNCT's 'Pre_A'.

Listing 12.1: **Main Simulation File** (Example-29)

```matlab
clear all; clc; close all;
spng = pnstruct('ex_29_pdf');

%%%%% extract the GPenSIM incidence matrices
[Ao, A, Ai] = matrixD();

%%%%% convert to the PNCT incidence matrices
[Pre_A, Post_A, D] = ...
```

```
    gpensim_2_PNCT(spng.incidence_matrix);

%%%% dispaly the incidence matrices
disp('PNCT "Pre_A":'); disp(Pre_A);
disp('PNCT "Post_A":'); disp(Post_A);
disp('GPenSIM "Ae":'); disp(spng.incidence_matrix);

%%%% now, compare GPenSIM and PNCT
if all(all(eq(Post_A, transpose(Ai))))
    disp(['PNCT "Post_A" is the same as ' ...
        ' GPenSIM "Ai" transposed!']);
end
if all(all(eq(Pre_A, transpose(Ao))))
    disp(['PNCT "Pre_A" is the same as ' ...
        ' GPenSIM "Ao" transposed!']);
end
```

The findings:

```
PNCT "Pre_A":
0 0
0 0
1 1

PNCT "Post_A":
1 0
0 1
0 0

GPenSIM ext.  incidence matrix "Ae" = (Ai|Ao):
0 0 1 1 0 0
0 0 1 0 1 0

PNCT "Post_A" is the same as GPenSIM "Ai"
transposed!
PNCT "Pre_A" is the same as GPenSIM "Ao"
transposed!
```

Bibliography

Davidrajuh, R. (2018). *Modeling Discrete-Event Systems with GPenSIM*, Springer International Publishing, Cham.

Chapter 13

PNML-GPenSIM

Petri Net Markup Language (PNML; standard: ISO/IEC-15909) is an XML-based transfer format for Petri nets. With PNML, Petri net models can be transferred from GPenSIM to another tool. This chapter presents two functions that can exchange Petri net models implemented with GPenSIM with other tools. The two functions presented (`pnml2gpensim` and `gpensim2pnml`) in this chapter enhance the usefulness of GPenSIM. For example, since GPenSIM lacks animation of Petri net execution, one can use a tool with graphic facilities to run the Petri net models and then continue with GPenSIM for further model building and analysis:

1. Preliminary model building and Simulation: The initial Petri net model can be created using a tool with a graphical Petri net editor (e.g., PIPE2 ([PIPE2 2017]).
2. Conversion of the preliminary model into GPenSIM format: using the function pnml2gpensim, creating the static Petri net graph (PDF) and a main simulation file (MSF) with the initial dynamics (initial markings and firing times).
3. Advanced modeling and simulation with GPenSIM: With PDF and MSF, a modeler can continue to develop the pre-processors and post-processors to complete the model building.

Table-13.1 presents a summary of these two functions.

Function	Description
`pnml2gpensim`	creates GPenSIM Petri net model from PNML file.
`gpensim2pnml`	creates a PNML file from a GPenSIM Petri net model.

Table 13.1: PNML - GPenSIM functions.

13.1   gpensim2pnml

Function name: `gpensim2pnml`
Full name: GPenSIM to PNML.
Purpose: Create a PNML document that represents a Petri Net structure
in the GPenSIM environment.
Input: Either static Petri net graph structure (output of the function `pnstruct`)
or Petri net structure with initial dynamics (output of the function `initialdynamics`).
Output: a PNML document that represents the Petri net.
This functions uses: (none)
This function is used by: [in Main Simulation File]
Further info: Chapter 8.2, "PNML-2-GPenSIM Converter," in Davidrajuh
(2018) and Davidrajuh (2013).
Sample use:

```matlab
% in MSF:
spng = pnstruct('example_pdf');
% or,  pni = initialdynamics(spng, dyn);
gpensim2pnml(pni); % a PNML document will be produced
```

Related function: `pnml2gpensim`.

13.2   pnml2gpensim

Function name: `pnml2gpensim`
Full name: PNML to GPenSIM.
Purpose: From a PNML document that represents a Petri net structure,
create a Petri net Definition file (PDF), a sample main simulation file (MSF),
and COMMON_PRE and COMMON_POST files that represent the Petri
net structure in the GPenSIM environment.
This function performs the following steps:

1. PNML file to MATLAB structure: Convert the PNML file to MATLAB
   structure and get the root of the **DOM tree**.
2. DOM tree: From the root tree of the DOM tree, recursively visit the
   child nodes to get the PNML structure.
3. PNML structure: From the PNML structure, get the net child and
   start extracting the Petri net structure (places, transitions, and arcs).
4. Petri net structure to GPenSIM files: Write the Petri net structure
   into the GPenSIM files MSF and PDF.

Input: a PNML document that represents a Petri net.
Output: a PDF file (will be given a random name, e.g., "pdf???_pdf.m")
a sample main simulation file (named "msf.m"), and templates for common
processors (COMMON_PRE and COMMON_POST files)).

This functions uses: MATLAB function 'xmlread'.
This function is used by: [in Main Simulation File]
Further info: Chapter 8.2, "PNML-2-GPenSIM Converter," in Davidrajuh (2018) and Davidrajuh (2013).
Sample use:

```
% in MSF:
pnml2gpensim('samplePNML1.xml'); % the following files will be ...
    created:
% a PDF, an MSF, and common processor files
```

Application example: A simple example ("Example-30: pnml2gpensim") is given below.
Related function: gpensim2pnml.

13.3 Example-30: pnml2gpensim

Note that this example is the same as Example-31 given in Davidrajuh (2018).

Fig.13.1 shows a PNML file named 'PNML-file001.xml.' In the main simulation file (Listing-13.1), we feed this file to the function pnml2gpensim, which will automatically produce four GPenSIM files such as a main simulation file (named "msf.m"), a PDF (named "pdf???_pdf.m") and templates (sample) for COMMON_PRE and COMMON_POST.

Listing 13.1: **Main Simulation File** (Example-30)

```
clear all; clc; close all;
% input: name of the PNML file (incl. ending ".xml")
PNMLfile = 'PNML-file001.xml';
pnml2gpensim(PNMLfile);
disp(' ');
disp(' ************** ');
disp('GPenSIM files are generated for the PNML file: ');
disp(['          "', PNMLfile,'"']);

% list all the files in this folder
dir
```

These four automatically generated files (by the function pnml2gpensim) are shown in Listings 13.2 to 13.5.

Listing 13.2: **Generated PDF "pdf???_pdf.m"** (Example-30)

```
% GPenSIM PDF file generated from PNML-file001.xml ...
% PDF: 'pdf_pdf.m'
function [png] = pdf_pdf()
```

```xml
<?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 13.1: Sample PNML file ('PNML-file001.xml') for testing the function `pnml2gpensim`.

```
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};
```

Listing 13.3: **Generated MSF "msf.m"** (Example-30)

```
% GPenSIM Main Simulation File
% this MSF is generated from PNML file "PNML-file001.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);
```

Listing 13.4: **Generated template for COMMON_PRE** (Example-30)

```
% COMMON_PRE file generated from PNML file "PNML-file001.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
```

Listing 13.5: **Generated template for COMMON_POST** (Example-30)

```
% COMMON_POST file generated from PNML file "PNML-file001.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
```

Bibliography

Davidrajuh, R. (2013). Adding pnml capability to gpensim, *2013 European Modelling Symposium*, IEEE, pp. 183–188.

Davidrajuh, R. (2018). *Modeling Discrete-Event Systems with GPenSIM*, Springer International Publishing, Cham.

Chapter 14

Print Colors

This chapter presents the functions for printing colors of tokens, such as `prncolormap`, `prnfinalcolors`, and `prnfinalcolorsSummary`. Table-14.1 presents a summary of these functions.

Function	Description
`prncolormap`	prints tokens' colors at different places.
`prnfinalcolors`	prints the colors of the final tokens, which were the ones left in different places when the simulation was stopped.
`prnfinalcolorsSummary`	prints the colors of the final tokens in a summarized format.

Table 14.1: Summary of functions for printing token colors from the simulation results.

14.1   print_colormap_for_place

Function name: `print_colormap_for_place`
Purpose: Prints tokens' colors in one specific place.
Input Parameters: 1) Simulation results; 2) place index.
Output to screen: Printout of tokens' colors in the specific place.
This function is used by: `prncolormap` and [in MSF, processor files]
Further info: Refer to Chapter 2, "Colored Petri Nets," of Davidrajuh (2023) for details and examples.
Sample use:

```
% in Processor files:
...
Sim_Results = gpensim(pni);
```

```
% place p2's index is 5
print_colormap_for_place(Sim_Results, 5);
```

Related functions: `prncolormap`

14.2   prncolormap

Function name: `prncolormap`
Full name: Print color map.
Purpose: After simulation by `gpensim`, this function will print the colors of all the tokens in different places at different times.
Input Parameters: 1) simulation results (output of function `gpensim`); 2) Optional - a set of places.
Output to screen: The colors of all the tokens in all places (in specific places, if given as the optional input) will be printed on the screen.
This function uses: **print_colormap_for_place**.
This function is used by: [in the main simulation file]
Further info: Refer to Chapter 2, "Colored Petri Nets," of Davidrajuh (2023) for details and examples.
Sample use:

```
% in main simulation file
...
results = gpensim(pni);
prncolormap(results, {'pNUM1','pADDED', 'pRESULT'});
```

Application example: A simple example ("Example-31: Print token colors" in Section 14.5) is given at the end of this chapter.
Related functions: `print_colormap_for_place`, `prnfinalcolors`

14.3   prnfinalcolors

Function name: `prnfinalcolors`
Full name: Print colors of final tokens.
Purpose: After simulation by the function `gpensim`, this function will print the colors of all the final tokens that reside in various places at the end of the simulation.
Input Parameters: 1) simulation results (output of function `gpensim`); 2) Optional - a set of places.
Output to screen: Printout of the colors of all the final tokens at the end of the simulation.
This function uses: **print_colormap_for_place**.
This function is used by: [in the main simulation file]
Further info: Refer to Chapter 2, "Colored Petri Nets," of Davidrajuh

(2023) for details and examples.
Sample use:

```
% in main simulation file
...
results = gpensim(pni);
prnfinalcolors(results);
```

Application example: A simple example ("Example-31: Print token colors" in Section 14.5) is given at the end of this chapter.
Related functions: `print_colormap_for_place`, `prncolormap`

14.4 prnfinalcolorsSummary

Function name: `prnfinalcolorsSummary`
Full name: Print a summary of colors of the final tokens.
Purpose: **This is a compact version of the function `prnfinalcolors`.** After simulation by `gpensim`, this function will print a summary of the colors of all the final tokens that reside in various places at the end of the simulation.
Input Parameters: 1) simulation results (output of `gpensim`). 2) (Optional) a set of places.
Output to screen: Printout of a summary of all the colors of all the final tokens at the end of the simulation.
This function uses: `print_colormap_for_place`.
This function is used by: [in the main simulation file]
Further info: Refer to Chapter 2, "Colored Petri Nets," of Davidrajuh (2023) for details and examples.
Sample use:

```
% in main simulation file
...
results = gpensim(pni);
prnfinalcolorsSummary(results);
```

Application example: A simple example ("Example-31: Print token colors") is given below.
Related functions: `print_colormap_for_place`, `prncolormap`, `prnfinalcolors`

14.5 Example-31: Print token colors

Fig.14.1 shows a simple Petri net for experimenting with the functions for printing token colors after the simulation. In COMMON_PRE (Listing-
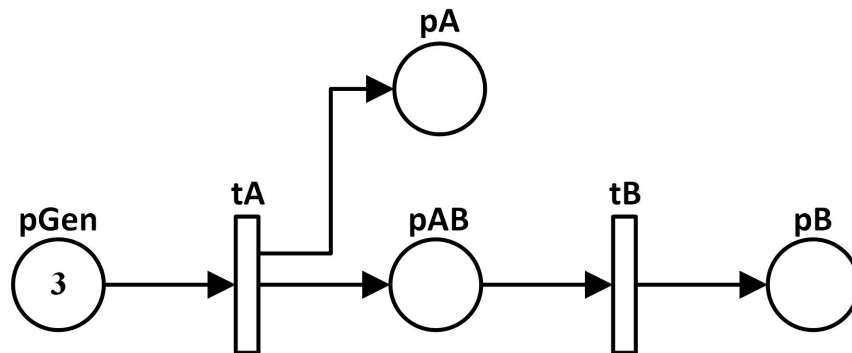
Figure 14.1: Petri net for testing the functions for printing token colors.

14.1), **tA** adds the color 'tA' to the tokens it deposits on **pAB**, whereas **tB** clears all colors and deposits colorless tokens on **pB**.

Listing 14.1: **COMMON_PRE** (Example-31)

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

switch transition.name
    case 'tA'
        % transition's name ("tA") is the new color
        transition.new_color = transition.name;
    case 'tB'
        % clear all inherited colors
        % also, add no new colors
        transition.override = 1;
end

% let the enabled transition fire
fire = 1;
```

Hence, when the simulation ends:

○ **pAB** should have, if there were any tokens on it, tokens with the color 'tA' only.
○ **pA** should also possess a number of tokens with the color 'tA'.
○ **pB** should have some colorless tokens.
○ **pGen**: If there is any token left in **pGen**, as these tokens are initial tokens, they must be colorless as well.

Listing 14.2: **Main Simulation File** (Example-31)

```matlab
clear all; clc; close all;
global global_info
```

```
global_info.STOP_AT = 10;

spng = pnstruct('ex_31_pdf');
dyn.m0 = {'pGen',3};
dyn.ft = {'allothers',1};
pni = initialdynamics(spng, dyn);
sim = gpensim(pni);

disp('prncolormap(sim) *****');
prncolormap(sim);

disp('prncolormap(sim, {"pA","pB"}) *****');
prncolormap(sim, {'pA','pB'});

disp('prnfinalcolors(sim) *****');
prnfinalcolors(sim);

disp('prnfinalcolors(sim, {"pA", "pB"}) *****');
prnfinalcolors(sim, {'pA', 'pB'});

disp('prnfinalcolorsSummary(sim) *****');
prnfinalcolorsSummary(sim);

disp('prnfinalcolorsSummary(sim, {"pGen", "pB"}) *****');
prnfinalcolorsSummary(sim, {'pGen', 'pB'});
```

The function `prncolormap`, without the second (optional) input of a set of places, prints all the colors of all the tokens that were located in all the places at any time during and at the end of the simulation (when the compiler was sampling the net with the default sampling frequency (if not changed by OPTION)).

```
prncolormap(sim) *****

**** **** Printing Colormap ...

Color Map for place:  pA
% print the extracted color_map
Time:  1 Colors:  "tA"
Time:  2 Colors:  "tA"
Time:  2 Colors:  "tA"
Time:  2 Colors:  "tA"
Time:  3 Colors:  "tA"
...
...


Color Map for place:  pAB
% print the extracted color_map
Time:  1 Colors:  "tA"
Time:  2 Colors:  "tA"
Time:  3 Colors:  "tA"

Color Map for place:  pB
(no colors)

Color Map for place:  pGen
(no colors)
```

When a set of places (e.g., {'pA', 'pB'}) are input to function `prncolormap` as the optional input, it will only print the colors of tokens that were available on these specific places at any time during and at the end of the simulation.

```
prncolormap(sim, "pA","pB") *****

**** **** Printing Colormap ...

Color Map for place:  pA
% print the extracted color_map
Time:  1 Colors:  "tA"
Time:  2 Colors:  "tA"
Time:  2 Colors:  "tA"
Time:  2 Colors:  "tA"
Time:  3 Colors:  "tA"
...
...


Color Map for place:  pB
 (no colors)
```

The function `prnfinalcolors`, without the second (optional) input of a set of places, will print only the colors of final tokens - tokens that were left in different places at the end of the simulation.

```
prnfinalcolors(sim) *****

**** **** Colors of Final Tokens ...
No.  of final tokens:  6

Place:  pA
Time:  1 Colors:  "tA"
Time:  2 Colors:  "tA"
Time:  3 Colors:  "tA"


Place:  pB
Time:  2 *** NO COLOR ***
Time:  3 *** NO COLOR ***
Time:  4 *** NO COLOR ***
```

Function `prnfinalcolors`, with the second (optional) input of a set of places (e.g., {'pA', 'pB'}), will print only the colors of final tokens that were left in the specific places at the end of the simulation. This printout will be the same as above, as the final tokens happen to end up in **pA** and **pB** only.

```
prnfinalcolors(sim, "pA", "pB") *****

**** **** Colors of Final Tokens ...
No.  of final tokens:  6

Place:  pA
Time:  1 Colors:   "tA"
Time:  2 Colors:   "tA"
Time:  3 Colors:   "tA"


Place:  pB
Time:  2 *** NO COLOR ***
Time:  3 *** NO COLOR ***
Time:  4 *** NO COLOR ***
```

Function `prnfinalcolorsSummary`, without the second (optional) input of a set of places, prints a summary of all the colors of tokens that were left in places at the end of the simulation run.

```
prnfinalcolorsSummary(sim) *****

Colors of Final Tokens (SUMMARY):

**** Place:  pA
Total number of tokens:  3
Colors:   "tA"
Total cost of tokens:  0


**** Place:  pAB
Total number of tokens:  0


**** Place:  pB
Total number of tokens:  3
*** NO COLOR ***
Total cost of tokens:  0


**** Place:  pGen
Total number of tokens:  0


Grand Totals:
Total number of tokens in all places:  6
Total cost of all the tokens:  0
Set of colors in all tokens:  "tA"
```

Function `prnfinalcolorsSummary`, with the second (optional) input of a set of places (e.g., {'pGen', 'pB'}), prints a summary of the colors of

final tokens that were left in the specific places at the end of the simulation run.

```
  prnfinalcolorsSummary(sim, "pGen", "pB") *****

Colors of Final Tokens (SUMMARY):

**** Place:  pGen
Total number of tokens:  0

**** Place:  pB
Total number of tokens:  3
*** NO COLOR ***
Total cost of tokens:  0

Grand Totals:
Total number of tokens in all places:  3
Total cost of all the tokens:  0
```

Bibliography

Davidrajuh, R. (2023). *Colored Petri Nets for Modeling of Discrete Systems: A Practical Approach With GPenSIM*, Springer Nature.

Chapter 15

Print State

This chapter presents the functions of printing states (markings). Table-15.1 presents a summary of these functions.

Function	Description
current_marking	returns the current marking (current state) of the simulation and prints it on the screen.
initial_marking	returns the initial marking (initial state) at the beginning of the simulation and prints it on the screen.
final_marking	returns the final marking (final state) at the end of the simulation and prints it on the screen.
marking_string	convert a marking (vector of natural numbers) into a printable text string.
prnstate	prints the current markings.
prnVirtualState	prints the current virtual state (tokens inside transitions).
prnTransStatus	prints details of current status (e.g., current time, firing transitions, and enabled transitions).

Table 15.1: Summary of functions for printing state info.

15.1   current_marking (new in version 11)

Function name: current_marking
Full name: current marking (current state).
Purpose: After the simulation starts (gpensim is running), we may want to know the current state in the processor files. This function prints the

current marking on screen in a compact form (e.g., "3p2 + 5p3") also returns it as a vector (length of the vector is the number of places (n_p)).

Note-1: current marking, as a vector, is the same as `PN.X`, where `PN` is the Petri Net run-time structure.

Note-2: This function is similar to `prnstate`.

Input Parameter: (none)

Output Parameter: A vector of natural numbers (length is n_p, number of places); in this vector, each column represents the number of tokens in a place identified by the column as an index. E.g., in a Petri net with five places, **p1** to **p5**, and if the initial marking is (3p2 + 5p3), then the vector returned will be [0 3 5 0 0].

Output to screen: The current state will be printed in an intelligible format (e.g., **3p2 + 5p3**).

This function uses: (none)

Sample use:

```
% in a pre-processor
disp(' ');
disp(['Before firing of "', trans.name, '" :']);
cMarking = current_marking();
...
```

Application example: A simple example ("Example-32: Print Markings" in Section 15.9) is given at the end of this chapter.

Related functions: `initial_marking`, `final_marking`, `prnstate`

15.2   initial_marking (new in version 11)

Function name: `initial_marking`

Full name: initial marking (initial state).

Purpose: After the simulation starts (`gpensim` is running), we may want to know the initial state in the processor files. This function prints the initial marking on screen in a compact form (e.g., '3p2 + 5p3') also returns it as a vector (length is number of places (n_p)).

Note: initial marking, as a vector, is the same as `pni.initial_marking` and `PN.initial_marking`, where `pni` is the Petri Net structure with initial marking and `PN` is the Petri Net run-time structure.

Input Parameter: (none)

Output Parameter: A vector of natural numbers (length is n_p, number of places); in this vector, each column represents the number of tokens in a place identified by the column as an index. E.g., in a Petri net with five places, **p1** to **p5**, and if the initial marking is (3p2 + 5p3), then the vector returned will be [0 3 5 0 0].

Output to screen: The initial state will be printed in an intelligible format

(e.g., **3p2 + 5p3**).
This function uses: (none)
Sample use:

```
% in the main simulation file or a processor file
disp(' ');
iMarking = initial_marking();
...
```

Application example: A simple example ("Example-32: Print Markings" in Section 15.9) is given at the end of this chapter.
Related functions: current_marking, final_marking

15.3   final_marking (new in version 11)

Function name: final_marking
Full name: final marking (final state).
Purpose: After the simulation ends (gpensim stopped running), we may want to know the final state in the main simulation file. This function prints the final marking on screen in a compact form (e.g., '3p2 + 5p3') also returns it as a vector (length is number of places).
Note: final marking, as a vector, is the same as PN.X in the simulation results output from gpensim.
Input Parameter: (none)
Output Parameter: A vector of natural numbers (length is n_p, number of places); in this vector, each column represents the number of tokens in a place identified by the column as an index. E.g., in a Petri net with five places, **p1** to **p5**, and if the final marking is (3p2 + 5p3), then the vector returned will be [0 3 5 0 0].
This function uses: (none)
Sample use:

```
% in the main simulation file
disp(' ');
fMarking = final_marking();
...
```

Application example: A simple example ("Example-32: Print Markings" in Section 15.9) is given at the end of this chapter.
Related functions: initial_marking, current_marking

15.4   markings_string

Function name: `markings_string`
Full name: Text string representing a marking (state).
Purpose: returns a text string representing the given input marking.
Input Parameter: a vector of length n_p (number of global places) representing a marking (e.g., the output of function *current_marking*).
Input Parameter - Optional: selected places: a vector of places indices.
Output Parameter : a text string representing the given marking.
Sample use:

```
% in a processor file:
...

% get current marking
cMarking = current_marking();

% expected state: 5 more tokens in p3
expected_state = cMarking + [0 0 5 0 0];

% make a text string for expected_state
mStr = markings_string(expected_state);

% print the text string on screen
disp(['Expected marking: ', mStr]);
```

15.5   print_real_time_state_info

Function name: `print_real_time_state_info`
Full name: Print status of all transitions (enabled and firing).
Purpose: same as function `prnTransStatus`; see Section 15.7.

15.6   prnstate

Function name: `prnstate`
Full name: Print current state (marking).
Purpose: To print the current state (marking) that shows the number of tokens in different places.
Input Parameter - Optional: Any text that should precede the current state display (e.g., 'Current state is: ').
Output Parameter: (none). **Output to screen**: Display of current state (marking); e.g., p2 + 3p5 + 7p7.
This function uses: **markings_string**.
Sample use:

```
% in a processor file (pre- or post-)

% display the current state
prnstate('The current state is: ');
```

The potential output will be something like:

```
The current state is:  5pRobot1 + 7pRobot2
```

Application example: A simple example ("Example-33: Print State" in Section 15.10) is given at the end of this chapter.
Related functions: prnTransStatus, prnVirtualState, markings_string

15.7   prnTransStatus (new in version 11)

Function name: prnTransStatus
Full name: Print status of all transitions (enabled and or firing).
Purpose: (During simulation by the function gpensim) To print the status of all transitions (enabled and or firing), along with the current time. Thus, this function is useful in the pre-processors and post-processors.
Note: this function is the same as print_real_time_state_info in the earlier versions.
Input Parameters: (none).
Output Parameters: (none).
Output to screen: The current time and the status (firing and or enabled) of each transition.
This function uses: (none)
This function is used by: [Pre- and post-processor files]
Sample use:

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

% every time any transition is enabled,
% print the status of all the transitions
prnTransStatus();
...
```

Application example: A simple example ("Example-33: Print State" in Section 15.10) is given at the end of this chapter.
Related functions: prnstate, markings_string, print_real_time_state_info

15.8   prnVirtualState

Function name: `prnVirtualState`

Full name: Print current **virtual** state (tokens inside transitions).

Purpose: To print the current **virtual** state that shows the number of tokens **still residing inside different transitions**.

Input Parameter- Optional: Any text that should precede the current virtual state display (e.g., 'Current virtual state is: ').

Output to screen: Display the current virtual state (tokens inside transitions).

This function uses: `markings_string`.

Further info: chapter 1.6, "Atomicity and Virtual Tokens," in Davidrajuh (2018).

Sample use:

```
% inside any pre- or post-processor files
prnVirtualState('Tokens in side transitions right now: ');
...
```

Application example: A simple example ("Example-33: Print State" in Section 15.10) is given at the end of this chapter.

Related functions: `prnstate`, `markings_string`

15.9   Example-32: Print Markings

This example is for experimenting with the functions `initial_marking`, `current_marking`, `final_marking`, and `marking_string`.
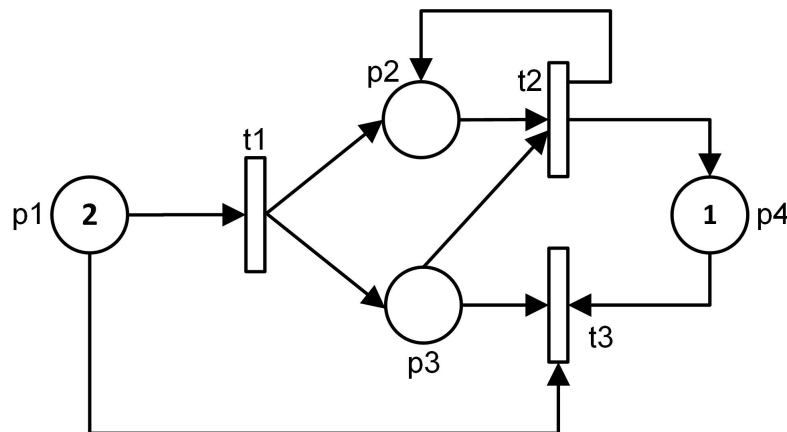

Figure 15.1: Petri net for testing the functions for extracting markings.

Listing-15.1 is the main simulation file for simulating the Petri net, which is shown in Fig.15.1. In the MSF, first, we echo (print to screen) the initial

marking (initial state) and then start gpensim. Once the simulation by
gpensim is completed, we will also echo the final marking (final state) on the
screen. Meanwhile, while gpensim is running, in the COMMON_PRE and
in COMMON_POST, we will echo the enabled transition and the current
state (in the COMMON_PRE) and the fired transition and the current state
(in the COMMON_POST).

Listing 15.1: **Main Simulation File** (Example-32)

```
clear all; clc; close all;

spng = pnstruct('ex_32_pdf');

dyn.m0 = {'p1',2, 'p4',1};
dyn.ft = {'allothers',1};
pni = initialdynamics(spng, dyn);

iniMarking = initial_marking();
disp(['Initial marking in vector form: ', ...
            int2str(iniMarking )]);

disp('starting "gpensim" .......');
sim = gpensim(pni);
disp('exiting "gpensim" .......');

finMarking = final_marking();
disp(['Final marking in vector form: ', ...
            int2str(finMarking )]);
```

Listing 15.2: **COMMON_PRE** (Example-32)

```
function [fire, trans] = COMMON_PRE(trans)
disp(' ');
disp(['Before firing of "', trans.name, '" :']);
current_marking();

fire = 1;
```

Listing 15.3: **COMMON_POST** (Example-32)

```
function [] = COMMON_POST(trans)

disp(['After firing of "', trans.name, '" :']);
current_marking();
disp(' ');
```

The printout on the screen (shown below) resembles animation; all we
need to do is to put "pause" statements in COMMON_PRE and COM-
MON_POST so that we can follow each firing and the resulting state change.

```
Initial marking:  2p1 + p4
Initial marking in vector form:  2 0 0 1


starting "gpensim" .......

Before firing of "t1" :
Current marking:  2p1 + p4
After firing of "t1" :
Current marking:  p1 + p2 + p3 + p4


Before firing of "t3" :
Current marking:  p1 + p2 + p3 + p4
After firing of "t3" :
Current marking:  p2

exiting "gpensim" .......


Final marking :  p2
Final marking in vector form:  0 1 0 0
```

15.10   Example-33: Print State

This example is the same as the previous example, Example-32 (Section 15.9).
However, in this example, we use the following functions instead: prnstate,
prnTransStatus, and prnVirtualState.

Listing-15.4 is the main simulation file for simulating the Petri net, which
is shown in Fig.15.1. In the MSF, first, we echo (print to screen) the initial
marking (initial state) and then start gpensim. Once the simulation by
gpensim is completed, we will also echo the final marking (final state) on the
screen. Meanwhile, while gpensim is running, in the COMMON_PRE and
COMMON_POST, we will echo the enabled and firing transitions, virtual
state, along with the current time and current state.

Listing 15.4: **Main Simulation File** (Example-33)

```
clear all; clc; close all;

spng = pnstruct('ex_33_pdf');
dyn.m0 = {'p1',2, 'p4',1};
dyn.ft = {'allothers',1};
pni = initialdynamics(spng, dyn);
```

```
disp('At start: ');
prnstate();

disp('starting "gpensim" .......');
sim = gpensim(pni);
disp('exiting "gpensim" .......');

disp('At last: ');
prnstate();
```

Listing 15.5: **COMMON_PRE** (Example-33)

```
function [fire, trans] = COMMON_PRE(trans)
disp(' ');
disp(['Before firing of "', trans.name, '" :']);
prnVirtualState('Virtual state: ');
prnstate('Current state: ');
prnTransStatus();

fire = 1;
```

Listing 15.6: **COMMON_POST** (Example-33)

```
function [] = COMMON_POST(trans)

disp(['After firing of "', trans.name, '" :']);
prnVirtualState('Virtual state: ');
prnstate('Current state: ');
prnTransStatus();
disp(' ');
```

The printout on the screen (shown below) resembles the printout of the previous example (Example-32) but includes additional information on enabled and firing transitions and virtual state.

```
At start:
2p1 + p4


starting "gpensim" .......

Before firing of "t1" :
Virtual state:  (no tokens)
Current state:  2p1 + p4

Time:  0
Enabled Transitions:  t1

After firing of "t1" :
Virtual state:  (no tokens)
Current state:  p1 + p2 + p3 + p4

Time:  1
Firing Transitions:  t1
Enabled Transitions:  t1
...
...
...
Time:  3
Firing Transitions:  t2


exiting "gpensim" .......

At last:
2p2 + 3p4
```

Bibliography

Davidrajuh, R. (2018). *Modeling Discrete-Event Systems with GPenSIM*, Springer International Publishing, Cham.

Chapter 16

Print State Space

This chapter presents only one function, namely `prnss`. This function prints the simulation results (output of the function `gpensim`) on the screen.

16.1   prnss

Function name: `prnss`
Full name: Print state space.
Purpose: This function takes the simulation results from `gpensim` and prints all the states and the causing transitions and timing on screen (no graphical display).
Input Parameter: Simulation Results (a structure output by function `gpensim`).
Output on screen: Text display on the screen showing the states, enabled transitions on each state and the fired transitions.
This function uses: (some sub-functions)
This function is used by: [in the main simulation file]
Further info: Chapter 2, "Modeling with GPenSIM: Basic Concepts," of Davidrajuh (2018).
Sample use:

```
% in Main Simulation File:
...
Sim_Results = gpensim(pni); % perform simulation runs
prnss(Sim_Results);
```

Application example: A simple example ("Example-34: prnss") is given below.
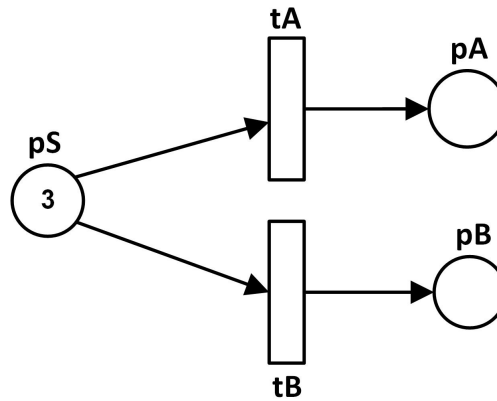**Related functions**: `cotree`

16.2   Example-34: prnss

Figure 16.1: Petri net for testing the functions `prnss`.

Petri net for testing the function `prnss` is shown in Fig.16.1. The reachability tree for this Petri net would possess several paths, as shown in Fig.16.2. However, when we simulate the Petri net using `gpensim`, only one of the paths is achieved. The function `prnss`, taking the simulation results from `gpensim`, will print the path with all the states, from the initial state to the final state.

Listing-16.1 shows the main simulation file, and the printout on screen is shown after the MSF.

Listing 16.1: **Main Simulation File** (Example-34)

```
clear all; clc; close all;

global global_info
global_info.STOP_AT = 100;

spng = pnstruct('ex_34_pdf');
dyn.m0 = {'pS',3};
dyn.ft = {'tA',1, 'tB',2};
pni = initialdynamics(spng, dyn);

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

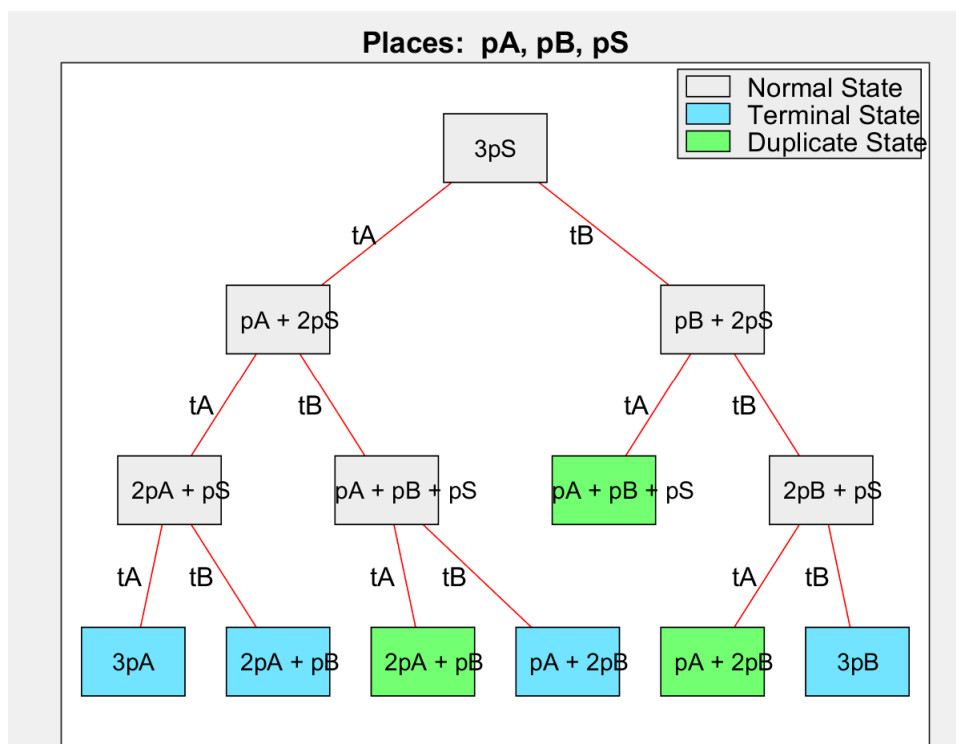The printout on the screen by the function `prnss` is shown below:

Figure 16.2: Reachability tree for the Petri net shown in Fig.16.1.

```
Simulation of "Example-34:  testing "prnss"":
======= State Diagram =======
** Time:  0 **
State:0 (Initial State):  3pS
At start ....
At time:  0, Enabled transitions are:  tA tB
At time:  0, Firing transitions are:  tA tB


** Time:  1 **
State:  1
Fired Transition:  tA
Current State:  pA + pS
Virtual tokens:  pS


Right after new state-1 ....
At time:  1, Enabled transitions are:  tA tB
At time:  1, Firing transitions are:  tA tB



** Time:  2 **
State:  2
Fired Transition:  tA
Current State:  2pA
Virtual tokens:  pS


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



** Time:  2 **
State:  3
Fired Transition:  tB
Current State:  2pA + pB
Virtual tokens:   (no tokens)


Right after new state-3 ....
At time:  2, Enabled transitions are:
At time:  2, Firing transitions are:
```

Bibliography

Davidrajuh, R. (2018). *Modeling Discrete-Event Systems with GPenSIM*, Springer International Publishing, Cham.

Chapter 17

Priority

This chapter presents four simple functions for the manipulation of transition priorities. These functions can be used in pre- and post-processor files. In GPenSIM, all the transitions have zero priority by default. At the start, in the main simulation file, transitions can be assigned different priorities (integer values; the higher the value, the higher the priority). During runtime, the priorities can be changed in pre-processors and post-processors using the functions mentioned in this chapter.

Table-17.1 presents a summary of these functions.

Function	Description
`priorcomp`	compares the priorities of two transitions.
`priordec`	decreases the priority of a transition by the given value (if a value is not provided, then by 1).
`priorinc`	increases the priority of a transition by the given value (if a value is not provided, then by 1).
`priority_enabled_ trans`	sorts out enabled transitions in descending order of priority.
`priorset`	set (assign) a value as priority to a transition.

Table 17.1: Summary of functions for manipulating transition priority.

17.1   priorcomp

Function name: `priorcomp`
Full name: Priority comparison.
Purpose: This function compares the priorities of two transitions. A positive (zero or negative, resp.) value will be returned if the first transition has

a higher (equal or lower, resp.) value than the second transition.

Input Parameter: Names or indices of two transitions.

Output Parameter:

1. '1': if the first transition has higher priority than the second transition.
2. '0': if the priorities are equal.
3. '-1': if the first transition has lower priority than the second transition.

This function uses: (none).

Further info: See Chapter 3, "Pre-processor and Post-processor Files," of Davidrajuh (2018).

Sample use:

```
% in the specific pre-processor of "t1", t1_pre.m:
...
HEL = priorcomp('t1', 't2');
fire = HEL; % fire, if priority of "t1" is higher than "t2"
```

Related functions: `priordec, priorinc, priorset`

17.2   priordec

Function name: `priordec`

Full name: Priority decrement.

Purpose: To decrease the priority of a transition. If the value (second input parameter, a positive integer) is given, then the priority will be reduced by that amount. Otherwise, priority will be decreased by 1.

Input Parameter - Compulsory: Name or index of the transition.

Input Parameter - Optional: Value (a positive integer) for decrement.

Output: (none).

This function uses: (none)

Further info: See Chapter 3, "Pre-processor and Post-processor Files," of Davidrajuh (2018).

Sample use:

```
% in the specific pre-processor of "t1", t1_pre.m:
...
% if "t2" is firing now, reduce its priority so that
%   "t2" will not fire in the next round
if is_firing('t2')
    priordec('t2');
end
```

Application example: A simple example ("Example-36: priordec" in Section 17.6) is given at the end of this chapter.

Related functions: `priorcomp, priorinc, priorset`

17.3   priorinc

Function name: `priorinc`
Full name: Priority increment.
Purpose: To increase the priority of a transition. If the value (second input parameter, a positive integer) is given, then the priority will be increased by that amount. Otherwise, priority will be increased by 1.
Input Parameter - Compulsory: Name or index of the transition.
Input Parameter - Optional: Value (a positive integer) for increment.
Output: (none).
This function uses: (none).
Further info: See Chapter 3, "Pre-processor and Post-processor Files," of Davidrajuh (2018).
Sample use:

```
% in the specific pre-processor of "t1", t1_pre.m:
...
% if "t2" is not firing now, increase its priority so that
%   "t2" will fire in the next round
if not(is_firing('t2'))
    priorinc('t2');
end
```

Application example: A simple example ("Example-35: priorset and priorinc" in Section 17.5) is given at the end of this chapter.
Related functions: `priorcomp`, `priordec`, `priorset`

17.4   priorset

Function name: `priorset`
Full name: Set new priority to a transition.
Purpose: To set a new value to the priority of a transition. The priority values are positive integers; the higher the value, the better the priority. The default priority values of transitions are zero.
Input Parameters: 1) Name or index of the transition; 2) The value for priority (a positive integer).
Output Parameter: (none).
This function uses: (none).
Further info: See Chapter 3, "Pre-processor and Post-processor Files," of Davidrajuh (2018).
Sample use:

```
% in the specific pre-processor of "t1", t1_pre.m:
...
% if "t2" is not firing now, set its priority to a
```

```
% high value so that "t2" will fire in the next round
if not(is_firing('t2'))
    priorset('t2', 10);
end
```

Application example: A simple example ("Example-35: priorset and priorinc") is given below.

Related functions: `priorcomp, priordec, priorinc`
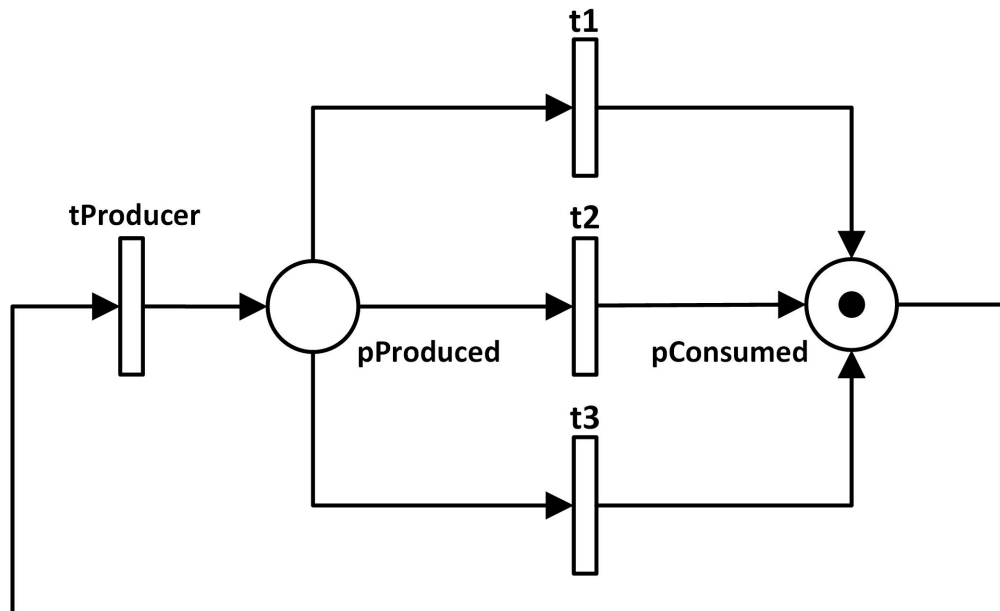
17.5   Example-35: priorset and priorinc

Figure 17.1: Petri net model for the Producer-Consumers Problem.

The producer-consumer problem (PCP) is a classical problem in Computer Science. PCP is an example of a synchronization problem when multiple processes are involved. Fig.17.1 shows a crude Petri net model to simulate the PCP. The model is composed of a producer (**tProducer**) and three consumers (**t1**, **t2**, and **t3**). Also, there are two buffers: **pProduced** and **pConsumed**.

In this model, the producer **tProducer** is active, and the consumers are passive. **tProducer** decides who will consume the product in each cycle by randomly choosing the consumer. At the beginning of each cycle:

- ○ The priorities of all consumers are reset to 0.
- ○ **tProducer** randomly chooses a consumer.
- ○ The chosen consumer's priority will be increased by 1.

○ Since all three consumers compete for the product, the one with the higher priority (the chosen one for this round) will win.

Listings 17.1 to 17.3 show the main simulation file, COMMON_PRE, and COMMON_POST. After the listings, a typical simulation output is displayed.

Listing 17.1: **Main Simulation File** (Example-35)

```matlab
clear all; clc; close all;
global global_info
global_info.STOP_AT = 20;

spng = pnstruct('ex_35_pdf');

dyn.m0 = {'pConsumed',1};
dyn.ft = {'allothers',1};
pni = initialdynamics(spng, dyn);

sim = gpensim(pni);
```

Listing 17.2: **COMMON_PRE** (Example-35)

```matlab
function [fire, trans] = COMMON_PRE(trans)
trans_name = trans.name;

if ismember(trans_name, {'t1','t2','t3'})
    disp(['Enabled consumer: ', trans_name]);
    fire = 1; return
end

%%%%%% if we are here, it is the producer
%% Step-1: reset all priorities
priorset('t1',0); priorset('t2',0); priorset('t3',0);
%% step-2: randomly choose a consumer
choosen_consumer = ceil(3*rand);
nameChosen = ['t', int2str(choosen_consumer)];
disp(['Producer chooses: "', nameChosen, '"']);
%% step-3: set higher priority to choosen
priorset(nameChosen, 1);

fire = 1;
```

Listing 17.3: **COMMON_POST** (Example-35)

```matlab
function [] = COMMON_POST(trans)
trans_name = trans.name;

% print name of the consumer only
if ismember(trans_name, {'t1','t2','t3'})
```

```
    disp(['fired consumer: "', trans.name, '" :']);
    disp(' ');
end
```

The printout on the screen:

```
Producer chooses:   "t1"
Enabled consumer:   "t1"
fired consumer:   "t1"

Producer chooses:   "t3"
Enabled consumer:   "t3"
fired consumer:   "t3"

Producer chooses:   "t1"
Enabled consumer:   "t1"
fired consumer:   "t1"

Producer chooses:   "t2"
Enabled consumer:   "t2"
fired consumer:   "t2"

Producer chooses:   "t2"
...
...
```

17.6   Example-36: priordec

We shall use the same Petri net model (Fig.17.1) for the producer-consumer problem (PCP) that we used in the previous example (Example-35, Section 17.5). In the previous example, the producer (**tProducer**) was active as it chose the consumer for each round. However, in this example, the producer is passive, while the consumers are active. The consumers decide which consumer will consume in the current round.

The decision-making process is elegantly simple yet remarkably effective: a consumer is allowed to consume at random. After consumption, in COMMON_POST, the consumer decreases its priority, thereby giving the other two (starving) consumers a chance in the next round.

Listings 17.4 and 17.5 show the MSF and COMMON_POST; COMMON_PRE is not needed. After the listings, a typical simulation output is displayed.

Listing 17.4: **Main Simulation File** (Example-36)

```
clear all; clc; close all;

global global_info
global_info.STOP_AT = 200;

spng = pnstruct('ex_36_pdf');

dyn.m0 = {'pConsumed',1};
dyn.ft = {'allothers',1};
pni = initialdynamics(spng, dyn);

sim = gpensim(pni);
disp("SUMMARY ********");
disp(['t1 fired: ', int2str(timesfired('t1'))]);
disp(['t2 fired: ', int2str(timesfired('t2'))]);
disp(['t3 fired: ', int2str(timesfired('t3'))]);
```

Listing 17.5: **COMMON_POST** (Example-36)

```
function [] = COMMON_POST(trans)

trans_name = trans.name;
% print name of the consumer only
if ismember(trans_name, {'t1','t2','t3'})
    priordec(trans_name);
    disp(['fired consumer: "', trans.name, '"']);
    disp(' ');
end
```

The printout on the screen:

```
...
...
...
fired consumer:  "t2"
fired consumer:  "t3"
fired consumer:  "t1"
fired consumer:  "t2"
fired consumer:  "t3"


SUMMARY ********
t1 fired:  33
t2 fired:  33
t3 fired:  33
```

Bibliography

Davidrajuh, R. (2018). *Modeling Discrete-Event Systems with GPenSIM*, Springer International Publishing, Cham.

Chapter 18

Reachability Tree with Time and Cost

This chapter presents only one function: 'cotreeTC.' The reachability (coverability) tree generated by the function cotree possesses four shortcomings:

1. The function cotree does not include time, even if the transitions have firing times.
2. The function cotree does not include costs, even if the transitions are assigned with firing costs.
3. The function cotree does not consider the pre-processors (the enabling conditions).
4. The function cotree does not allow parallel firings of transitions; only one transition is allowed to fire at a time.

The function cotreeTC, presented in this chapter, fixes the first two shortcomings of the function cotree:

1. The function cotreeTC considers the firing times of transitions (if transitions are timed)
2. The function cotreeTC considers the firing costs of transitions (if transitions are assigned with firing costs)

However, the function cotreeTC still has the other two issues: It does not consider pre-processors and allows only one transition to fire at a time.

18.1   cotreeCT (new in version 11)

Function name: cotreeCT
Full name: Print coverability (reachability) tree with time and cost.
Purpose: This function is similar to cotree, as it prints the reachability

tree. However, each state will be printed with the time and cost of the state if the transitions are timed and assigned with firing costs. Also, optionally, we can limit the number of states printed by giving an optional input number.

Input Parameters: 1) Petri net structure with initial dynamics (`pni`, output of `initialdynamics`). 2) Optional: maximum number of states to be printed.

Output on screen: a printout of state space (similar to the printout of `cotree`); however, the time and cost of the states are also shown.

Output Parameter: `COTREE_TC` matrix in which each row corresponds to a state. In a row, the first part is the marking, followed by fired transition, parent state, time of the state, and cost of the state.

This function uses: `print_retree`

Further info: Sections 2.1, "COTREE Matrix" of this reference manual. Also, Chapter 4, "Analysis of Petri Nets," in Davidrajuh (2018) for reachability tree; chapter 6, "Token selection based on cost," in Davidrajuh (2023) for cost calculation.

Sample use:

```
% in main simulation file
spng = pnstruct('cotree_example_def');
dyn.m0 = {'p1',2, 'p4', 1};
dyn.ft = {'t1',2, 't2', 1};
dyn.fc_fixed = {'t1',16, 't2', 32};
dyn.fc_variable = {'t1',5, 't2', 5};
pni = initialdynamics(spng, dyn);
COTREE_TC = cotreeTC(pni, 10); % print only the first ten states
```

Application example: A simple example ("Example-37: cotreeTC" in Section 18.3) is given at the end of this chapter.

Related functions: `cotree`

18.2   retree

Function name: `retree`
Full name: Print coverability (reachability) tree with time and cost.
Purpose: same as function `cotreeCT`; see Section 18.1.

18.3   Example-37: cotreeTC

Fig.18.1 shows a simple Petri net for which we are going to generate the reachability tree with time and costs. Listing-18.1 shows the main simulation file in which we add time and cost to the transitions.
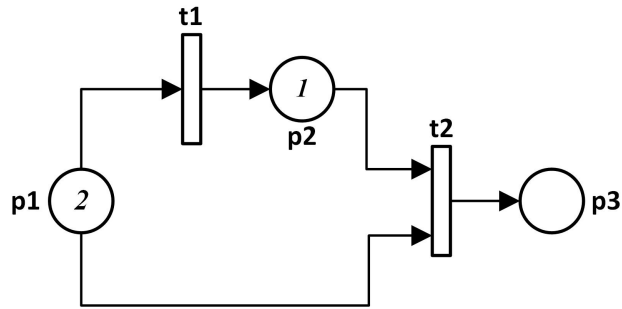
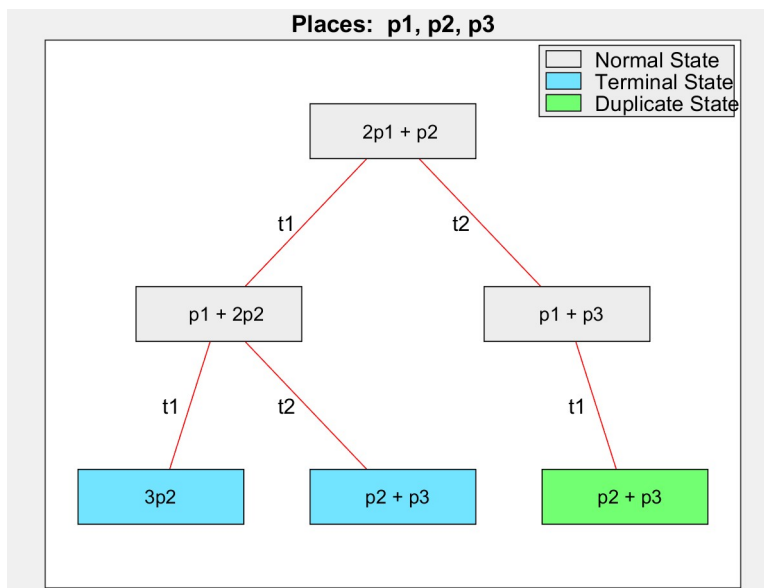Figure 18.1: Petri net for generating reachability tree with time and cost.

Figure 18.2: Reachability tree generated with the function `cotree`.

Listing 18.1: **Main Simulation File** (Example-37)

```
clear all; clc; close all;

spng = pnstruct('ex_37_pdf');
dyn.m0 = {'p1',2, 'p2',1};
dyn.ft = {'t1',1, 't2',2};
dyn.fc_fixed = {'t1',100, 't2',200};
dyn.fc_variable = {'t1',10, 't2',20};

pni = initialdynamics(spng, dyn);
COTREE_TC = cotreeTC(pni);
disp('');
disp('COTREE_TC matrix:'); disp(COTREE_TC);
cotree(pni, 1);
```

Fig.18.2 shows the coverability tree **obtained by the function `cotree`**, in which **time and cost details are missing**.

The function `cotreeTC` printout ASCII text output only, and it doesn't produce graphical out. The text output is given below. Comparing the text output with the graphical plot in Fig.18.2, we can see that the text output prints all the state info and adds time and costs as well.

```
======= Extended Reachability Tree =======
State no.:  1 ROOT node
2p1 + p2

State no.:  2 Firing event:  t1
Time of the state:  1
Cost of the state:  110
State:  p1 + 2p2
Node type:  ' '  Parent state:  1

State no.:  3 Firing event:  t2
Time of the state:  2
Cost of the state:  240
State:  p1 + p3
Node type:  ' '  Parent state:  1

State no.:  4 Firing event:  t1
Time of the state:  2
Cost of the state:  220
State:  3p2
Node type:  'T' Parent state:  2

State no.:  5 Firing event:  t2
Time of the state:  3
Cost of the state:  350
State:  p2 + p3
Node type:  'T' Parent state:  2

State no.:  6 Firing event:  t1
Time of the state:  3
Cost of the state:  350
State:  p2 + p3
Node type:  'D' Parent state:  3
```

Finally, the screen dump of the matrix COTREE_TC is given below. In this matrix:

1. As there are three places, the first three integers are the markings.
2. The fourth element is the fired transition (represented) that causes the state.
3. The fifth element is the parent state (row number).
4. The sixth element is the type of the state (Root = 82, normal = 0, duplicate = 68, or terminal = 84).
5. The seventh element is the time of the state.

6. Finally, the eighth and last element is the cost of the state.

```
COTREE_TC matrix:
2 1 0 0 0 82 0 0
1 2 0 1 1 0 1 110
1 0 1 2 1 0 2 240
0 3 0 1 2 84 2 220
0 1 1 2 2 84 3 350
0 1 1 1 3 68 3 350
```

Bibliography

Davidrajuh, R. (2018). *Modeling Discrete-Event Systems with GPenSIM*, Springer International Publishing, Cham.

Davidrajuh, R. (2023). *Colored Petri Nets for Modeling of Discrete Systems: A Practical Approach With GPenSIM*, Springer Nature.

Chapter 19

Resource Management

In Petri nets, resources are represented by places with some initial tokens equaling the number of available instances of the resource. If the modeler wants, resources can be handled differently in GPenSIM, as they can be kept away from the Petri net. Thus, the Petri net model becomes smaller, and GPenSIM takes care of the resource management in the background. This chapter presents functions for transitions acquiring resources, and then releasing them after use. There are also two functions, namely `prnschedule` and `plotGC`, which are useful to print and plot resource usage.

The acquisition of resources happens in the pre-processor files and the release in post-processor files. For complete details on resources, see chapter 8, "GPenSIM Resources: The Basics," in Davidrajuh (2023). Table-19.1 summarizes the functions for resource acquisition and release.

A short note on GPenSIM Resources: The functions presented in this chapter involve 'instances of a resource', 'generic resource', 'specific resource', and 'write access':

1. Instances of a resource: A resource may have many identical copies ('instances'); E.g., in a supermarket, there are three cashiers, and all of them are the same for a customer who wants to check out. In this case, we say that the resource 'cashier' possesses three instances.
2. Generic resource: Though the system may possess multiple resources, we just need any resource - it doesn't matter which one - for some applications.
3. Specific resource: For some applications, we need a specific (named) resource, as a generic resource may not suit well for the job.
4. Write access: If a resource contains many instances, in some cases (e.g., for maintenance), we may need to acquire all the instances of the resource.

Function	Description
`availableInst`	Checks whether any instances of a resource are available.
`availableRes`	Checks whether any resources are available.
`requestSR`	Requests some instances from specific ('named') resources.
`requestGR`	Requests some resource instances without naming any resource.
`requestAR`	Requests some resource instances among many alternative resources.
`requestWR`	Requests all the instances of a specific resource (exclusive access).
`release`	Releases all the resources and their resource instances held by a transition.
`prnschedule`	Prints useful information on resource usage.
`plotGC`	Plots Gantt Chart, showing how the resources were used (by which transition and how long).

Table 19.1: Summary of functions for resource usage.

19.1   availableInst

Function name: `availableInst`
Full name: Available instances of a given resource.
Purpose: Returns information about a given resource's available (free) instances.
Input Parameter: Name or index of the resource.
Output Parameter: A structure, "Resource Availability Info (`RAI`)" consisting of the three fields:

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

This function uses: (none).
Further info: See chapter 8, "GPenSIM Resources: The Basics," in Davidrajuh (2023).
Sample use:

```
% in a pre-processor file
...
% check available printers
avInst_PRN = availableInst('printers');
```

```
if not(avInst_PRN.n)
    % no available printers
    ...
```

Related functions: `availableRes`

19.2 availableRes

Function name: `availableRes`
Full name: Available resources.
Purpose: Returns info about all the available (free) resources.
Input Parameter: a set of resource names (or resource indices). If the input parameter is not given, then all the resources will be checked for available instances.
Output Parameter: An array of structures (called "Resource Availability Info '`RAI`'"), where each structure has the following elements:

- `avRES.r_index`: index of the resource.
- `avRES.n`: this resource's free (available) instances.
- `avRES.instance_indices`: the set of indices of the available instances.

This function uses: (none).
Further info: See chapter 8, "GPenSIM Resources: The Basics," in Davidrajuh (2023).
Sample use:

```
% in processor file
% for checking available instances of 'prn', 'cpu':
avRes = availableRes({'printer', 'CPU'});

% for checking all available resources
% avRes = availableRes();

for i = 1: len(avRes)
   if avRES(i).n
      % there some available instances of this resource
      ...
    end
end
...
```

Related functions: `availableInst`

19.3 release

Function name: `release`
Full name: Release resource(s) used after firing of a transition.

Purpose: Releases *all* the resources used by a transition. Note that releasing only some of the resources is not possible.

Input Parameter (optional): Name or index of the fired transition. If the name is not given, then the transition just fired is assumed to release the resources it used.

Output Parameter: (none). (the resources used by the transition are released)

This function uses: (none).

Further info: See chapter 8, "GPenSIM Resources: The Basics," in Davidrajuh (2023).

Sample use:

```
% in post-processor of tROBOT2 (tROBOT2_post)
...
release('tRobot1'); % release all resources used by tRobot1
release(); % also release all resources used by tROBOT2
```

Application example: A simple example ("Example-38: Resource Management" in Section 19.10) is given at the end of this chapter.

Related functions: `requestAR`, `requestGR`, `requestSR`, `requestWA`

19.4   requestAR

Function name: `requestAR`

Full name: Request **alternative** resources.

Purpose: To reserve some resource instances from a group of named resources.

Input Parameters: 1) a pool of resources; 2) the number of instances required.

Output Parameter: Boolean, '0' if the reservation is unsuccessful.

This function uses: (none).

Further info: See chapter 8, "GPenSIM Resources: The Basics," in Davidrajuh (2023).

Sample use:

```
% in a pre-processor file
...
% request *any of the* two German cars
r1 = requestAR({'Benz', 'BMW', 'Audi'}, 2);

% and *any of the* 3 Japanese cars
r2 = requestAR({'Toyota', 'Honda', 'Mazda'}, 3);

if and(r1, r2)
    % yes, we have reserved 2 German + 3 Japanese cars
```

```
...
```

Related functions: `requestGR, requestSR, requestWA`

19.5 requestGR

Function name: `requestGR`
Full name: Request **generic** (not specifically named) resources.
Purpose: To reserve some instances from any of the (unspecified) resources.
Input Parameter: the number of instances required.
Output Parameter: Boolean, '0' if the reservation is unsuccessful.
This function uses: (none)
Further info: See chapter 8, "GPenSIM Resources: The Basics," in Davidrajuh (2023).
Sample use:

```
% in a pre-processor file
...
% request *any* two cars (German, Japan, whatever...)
status = requestGR(2);
...
```

Related functions: `requestAR, requestSR, requestWA`

19.6 requestSR

Function name: `requestSR`
Full name: Request **specific** resources.
Purpose: To reserve some instances of the specific (named) resources.
Input Parameter: a list of resources and the number of instances required.
Output Parameter: Boolean, '0' if the reservation is unsuccessful.
This function uses: (none).
Further info: See chapter 8, "GPenSIM Resources: The Basics," in Davidrajuh (2023).
Sample use:

```
% in a pre-processor file
...
% request two BMWs and one Totoya
status = requestSR({'BMW',2, 'Toyota',1});
...
```

Application example: A simple example ("Example-38: Resource Management" in Section 19.10) is given at the end of this chapter.
Related functions: `requestAR, requestGR, requestWA`

19.7 requestWA

Function name: `requestWA`
Full name: Request Write Access to a resource.
Purpose: Request **all the instances** of a specific resource.
Input Parameter: Name of the resource.
Output Parameter: Boolean, '0' if the reservation is unsuccessful.
This functions uses: (none).
Further info: See chapter 8, "GPenSIM Resources: The Basics," in Davidrajuh (2023).
Sample use:

```matlab
% in a pre-processor file
...
% request *all the instances* of Cashiers
status = requestWA('Cashier');
if not (status)
    disp(['some of the Cashier instances, '...
        ' or all of them are busy']);
    ...
end
```

Related functions: `requestAR`, `requestGR`, `requestSR`

19.8 plotGC

Function name: `plotGC`
Full name: Plot Gantt Chart.
Purpose: After simulation involving resources, this function plots the resource usage as a Gantt Chart.
Input Parameters:

1. **Simulation results** (output of `gpensim`).
2. Optional input - Process groups: we can group transitions into groups. As the optional second input parameter, we can group transitions into processes (e.g., 't3', 't5', 't2', 't4', 't6', 't1') so that transitions belonging to the same process will be plotted in the same color.
3. Optional input - Time Limit: we can limit the time axis on the plot.
4. Optional input - Title: Text for the title of the plot.

Output to screen: A Gantt Chart is plotted.
This function uses: (none)
Further info: See chapter 8, "GPenSIM Resources: The Basics," in Davidrajuh (2023).
Sample use:

```
% main simulation file
...
dyn.m0 = {'pSTART',10};
dyn.ft = {'allothers',1};
dyn.re = {'Resource-X',3, inf}; % resource as semafor
pni = initialdynamics(spng, dyn);
sim = gpensim(pni);

% plot Gantt Chart
processGroups = {{'t3', 't5'}, {'t2', 't4', 't6'}, {'t1'}};
plotGC(sim, processGroups);
```

Application example: A simple example ("Example-38: Resource Management" in Section 19.10) is given at the end of this chapter.
Related functions: `prnschedule`

19.9   prnschedule

Function name: `prnschedule`
Full name: Print Schedule.
Purpose: After simulation involving resources, this function prints the resource usage. For every resource utilized, this function prints the transition that used the resource and the start and end times of utilization.
Input Parameter: Simulation results (output of `gpensim`).
Output on screen: A detailed report on resource usage is printed.
Output Parameters:

1. LE (Line efficiency, in %).
2. RES_USAGE: [number_of_resources X 2] matrix, column-1: number of times each resource was used; column-2: total time of usage of each resource.
3. completion_time (simulation end time).
4. LT (Line time).
5. Total_time_at_Ks: total time of all resource usage.

This function uses: (some sub-functions)
Further info: See chapter 8, "GPenSIM Resources: The Basics," in Davidrajuh (2023).
Sample use:

```
% part of Examples-19: "Resources to Realize Critical"
%   in Davidrajuh (2023) "Colored Petri Nets for Modeling
%   of Discrete Systems"
...
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(spng, dyn);
sim = gpensim(pni);

% summary of resource usage
prnschedule(sim);
```

Application example: A simple example ("Example-38: Resource Management" in given below.

Related functions: plotGC

19.10   Example-38: Resource Management
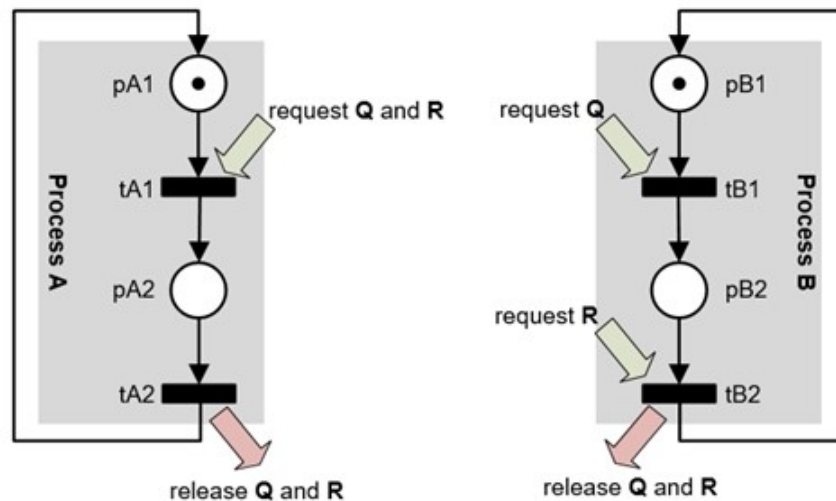
Figure 19.1: Processes **A** and **B** using two Resources **Q** and **R** (Adapted from Davidrajuh (2023).

This example is adapted from Davidrajuh (2023). Fig.19.1 shows a Petri net with two processes (**A** and **B**) that share two resources (**Q** and **R**):

○ Resources **Q** and **R** have one instance each.
○ Process **A** consists of two steps, **tA1** and **tA2**.
  – **tA1**: **tA1** needs both resources **Q** and **R** to start. When **tA1** is completed, it will not release the resources, as they are also required for step **tA2**.
  – **tA2**: When **tA2** is completed, both resources will be released.
○ Process **B** also consists of steps **tB1** and **tB2**.

    – **tB1**: **tB1** needs the resource **Q** to start. When **tB1** is completed, **Q** will not be released as it is necessary for **tB2** as well.

    – **tB2**: in addition to **Q**, **tB2** needs **R** too. When **tB2** is completed, resources **Q** and **R** will be released together.

○ Processes **A** and **B** run cyclically.

Listing-19.1 shows the main simulation file.

Listing 19.1: **Main Simulation File** (Example-38)

```
global global_info
global_info.STOP_AT = 20;

pns = pnstruct('ex_38_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);

% print resource usage
prnschedule(sim);
plotGC(sim, {{'tA1','tA2'},{'tB1','tB2'}});
```

The common processor files for the acquisition and release of resources are given in Listings 19.2. and 19.3.

Listing 19.2: **COMMON_PRE** (Example-38)

```
function [fire, transition] = COMMON_PRE(transition)
switch transition.name
    case 'tA1'
        % request: specific, both Q and R
        granted = requestSR({'Q',1, 'R',1});
    case 'tA2'
        % need nothing (do nothing)
        granted = 1;

    case 'tB1'
        % request: specific, Q only
        granted = requestSR({'Q',1});
    case 'tB2'
        % request: specific, R only
        granted = requestSR({'R', 1});
end

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

Listing 19.3: **COMMON_POST** (Example-38)

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

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

The printout generated by the function prnschedule is given below. This printout describes which transition is used for each resource and each instance of this resource. The printout ends with a summary of resource usage. The details of the printout are described in the following chapter on "Resource Usage".

```
RESOURCE USAGE:

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


RESOURCE INSTANCES OF ***** R *****
tB2 [1 :   2]
tA1 [2 :   4]
tA1 [4 :   6]
tA1 [6 :   8]
tA1 [8 :  10]
tB2 [11 :  12]
tB2 [13 :  14]
tB2 [15 :  16]
tA1 [16 :  18]
Resource Instance:  R::  Used 9 times.
Utilization time:  14
```

```
RESOURCE USAGE SUMMARY:
Q: Total occasions:  9 Total Time spent:  18
R: Total occasions:  9 Total Time spent:  14


***** 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:  32
LE = 80 %
**
Sum resource usage costs:  0 (NaN% of total)
Sum firing costs:  0 (NaN% of total)
Total costs:  0
**
```

Note that **tA2** is missing in the printout shown above and in the Gantt chart shown below (Fig.19.2). This is because **tA2** never acquired any resource; **tA2** was merely using the resource acquired by **tA1**. Hence, the time the resources (**Q** and **R**) used by **tA2** are credited to **tA1** in the printout and in the Gantt chart.

Fig.19.2 shows the Gantt Chart generated by the `plotGC` function. Note that this figure has been edited to be more intelligible.
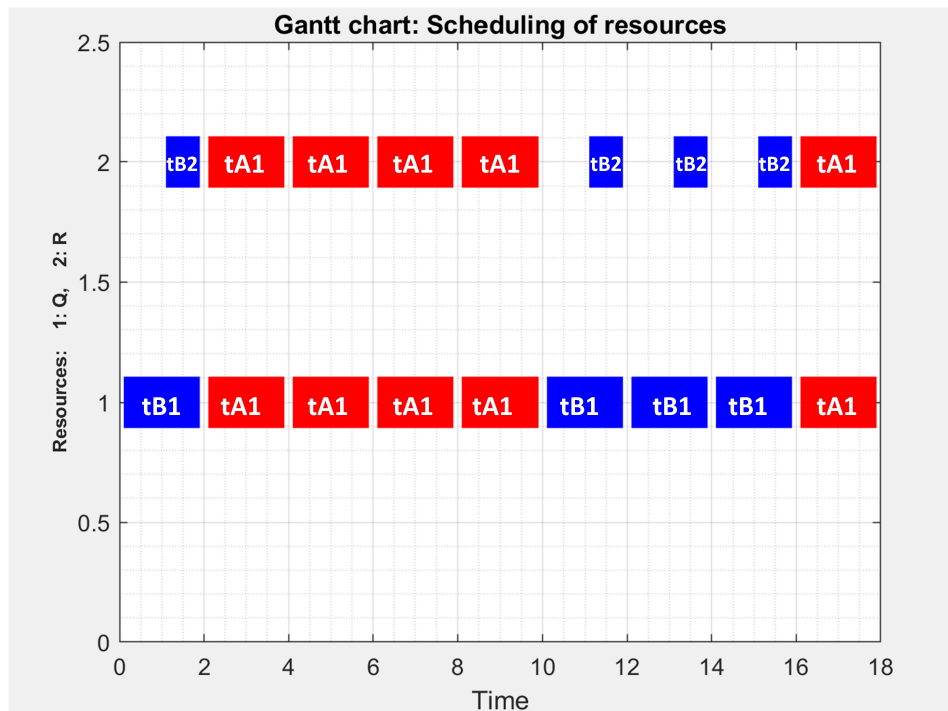
Figure 19.2: Gantt Chart showing the resource usage among the transitions.

Bibliography

Davidrajuh, R. (2023). *Colored Petri Nets for Modeling of Discrete Systems: A Practical Approach With GPenSIM*, Springer Nature.

Chapter 20

Structural-Invariants

This chapter presents the functions for finding structural invariants (e.g., siphons, traps, place invariants (P-invariants), and transition invariants (T-invariants). Structural invariants (*aka* net invariants) are the structural properties of a Petri net that depend only on the static (topological) structure, and they are independent of the Petri net dynamics (e.g., markings, firing times, etc.). For a detailed study, see Chapter 9, "Structural Invariants," in Davidrajuh (2018).

GPenSIM functions for structural properties use The Petri Net Control Toolbox (PNCT) developed at the University of Cagliari. GPenSIM uses the bridge `gpensim_2_PNCT()` (section 12.1) to access the functions in PNCT.

Table-20.1 summarizes the functions for structural invariants.

Function	Description
`pinvariant`	finds the place-invariant of a Petri net.
`siphons`	finds the siphons of a Petri net.
`siphons_minimal`	finds the minimal-siphons of a Petri net.
`tinvariant`	finds the transition-invariant of a Petri net.
`traps`	finds the traps of a Petri net.
`traps_minimal`	finds the minimal-traps of a Petri net.

Table 20.1: Summary of functions for finding structural invariant.

20.1 pinvariant

Function name: `pinvariant`
Full name: Place Invariant (P-Invariant).
Purpose: To find the place invariants of a Petri net.

Input Parameter: Static Petri net graph (`spng`, output of `pnstruct`); alternatively, Petri net with initial dynamics (`pni`, output of `initialdynamics`) can also be input.

Output Parameter: A matrix in which each row represents a P-invariant. In each row, non-zero columns are the indices of places involved in P-invariant. Also, the P-invariants are printed on the screen.

This function uses: (PNCT function `pinvar`)

Further info: See chapter 9, "Structural Invariants," in Davidrajuh (2018).

Sample use:

```
% Example-38 in Davidrajuh (2018)
spng = pnstruct('ptinvar_pdf');
PI = pinvariant(spng);
disp('P-Invariant (in matrix form): ');
disp(PI);
```

Application example: A simple example ("Example-40: pinvariant and tinvariant" in Section 20.8) is given at the end of this chapter.

Related functions: `tinvariant`

20.2   siphons

Function name: `siphons`

Full name: Siphons.

Purpose: To find the siphons of a Petri net. Siphons are a set of places which, if they become empty of tokens, will always remain empty for all reachable markings of the net.

Input Parameter: Static Petri net graph (`spng`, output of `pnstruct`); alternatively, Petri net with initial dynamics (`pni`, output of `initialdynamics`) can also be input.

Output Parameter: A matrix in which each row represents a siphon. In each row, non-zero columns are the indices of places involved in the siphon. Also, the siphons are printed on the screen. **This function uses**: (PNCT function `siphon`)

Further info: See chapter 9, "Structural Invariants," in Davidrajuh (2018).

Sample use:

```
% Example-36 in Davidrajuh (2018)
spng = pnstruct('siphons_pdf');
S = siphons(spng);
disp('Siphons (in matrix form): ');
disp(S);
```

Related functions: `siphon_minimal`

20.3 siphons_minimal

Function name: `siphons_minimal`
Full name: Minimal Siphons.
Purpose: To find the minimal siphons of a Petri net. Siphons are a set of places which, if they become empty of tokens, will always remain empty for all reachable markings of the net. A siphon is called a minimal siphon if it contains no other siphon.
Input Parameter: Static Petri net graph (`spng`, output of `pnstruct`); alternatively, Petri net with initial dynamics (`pni`, output of `initialdynamics`) can also be input.
Output Parameter: A matrix in which each row represents a minimal siphon. In each row, non-zero columns are the indices of places involved in the minimal siphon. Also, the minimal siphons are printed on the screen.
This function uses: (PNCT function `siphon`)
Further info: See chapter 9, "Structural Invariants," in Davidrajuh (2018).
Sample use:

```
% Example-36 in Davidrajuh (2018)
spng = pnstruct('siphons_pdf');
SM = siphons_minimal(spng);
disp('Minimal siphons (in matrix form): ');
disp(SM);
```

Application example: A simple example ("Example-39: siphons and traps" in Section 20.7) is given at the end of this chapter.
Related functions: `siphon`

20.4 tinvariant

Function name: `tinvariant`
Full name: Transition Invariant (T-Invariant).
Purpose: To find the transition invariants of a Petri net.
Input Parameter: Static Petri net graph (`spng`, output of `pnstruct`); alternatively, Petri net with initial dynamics (`pni`, output of `initialdynamics`) can also be input.
Output Parameter: A matrix in which each row represents a T-invariant. In each row, non-zero columns are the indices of transitions involved in the T-invariant. Also, the T-invariants are printed on the screen.
    This function uses: (PNCT function `tinvar`)
Further info: See chapter 9, "Structural Invariants," in Davidrajuh (2018).
Sample use:

```
% Example-38 in Davidrajuh (2018)
```

```
spng = pnstruct('ptinvar_pdf');
TI = tinvariant(spng);
disp('T-invariants (in matrix form): ');
disp(TI);
```

Application example: A simple example ("Example-40: pinvariant and tinvariant" in Section 20.8) is given at the end of this chapter.
Related functions: `pinvariant`

20.5   traps

Function name: `traps`
Full name: Traps.
Purpose: To find the traps of a Petri net. A trap is a group of places where they never lose all their tokens once marked.
Input Parameter: Static Petri net graph (`spng`, output of `pnstruct`); alternatively, Petri net with initial dynamics (`pni`, output of `initialdynamics`) can also be input.
Output Parameter: A matrix in which each row represents a trap. In each row, non-zero columns are the indices of places involved in the trap. Also, the traps are printed on the screen. **This function uses**: (PNCT function `traps`)
Further info: See chapter 9, "Structural Invariants," in Davidrajuh (2018).
Sample use:

```
% Example-37 in Davidrajuh (2018)
spng = pnstruct('traps_pdf');
T = traps(spng);
disp('Traps (in matrix form): ');
disp(T);
```

Related functions: `traps_minimal`

20.6   traps_minimal

Function name: `traps_minimal`
Full name: Minimal Traps.
Purpose: To find the traps of a Petri net. A trap is a group of places which never lose all their tokens once marked. A trap is called a minimal trap if it contains no other traps.
Input Parameter: Static Petri net graph (`spng`, output of `pnstruct`); alternatively, Petri net with initial dynamics (`pni`, output of `initialdynamics`) can also be input.
Output Parameter: A matrix in which each row represents a minimal trap. In each row, non-zero columns are the indices of places involved in

the minimal trap. Also, the minimal traps are printed on the screen. **This function uses**: (PNCT function `traps`)

Further info: See chapter 9, "Structural Invariants," in Davidrajuh (2018).

Sample use:

```
% Example-37 in Davidrajuh (2018)
spng = pnstruct('traps_pdf');
TM = traps_minimal(spng);
disp('Minimal traps (in matrix form): ');
disp(TM);
```

Application example: A simple example ("Example-39: siphons and traps") is given below.

Related functions: `traps`

20.7 Example-39: siphons and traps
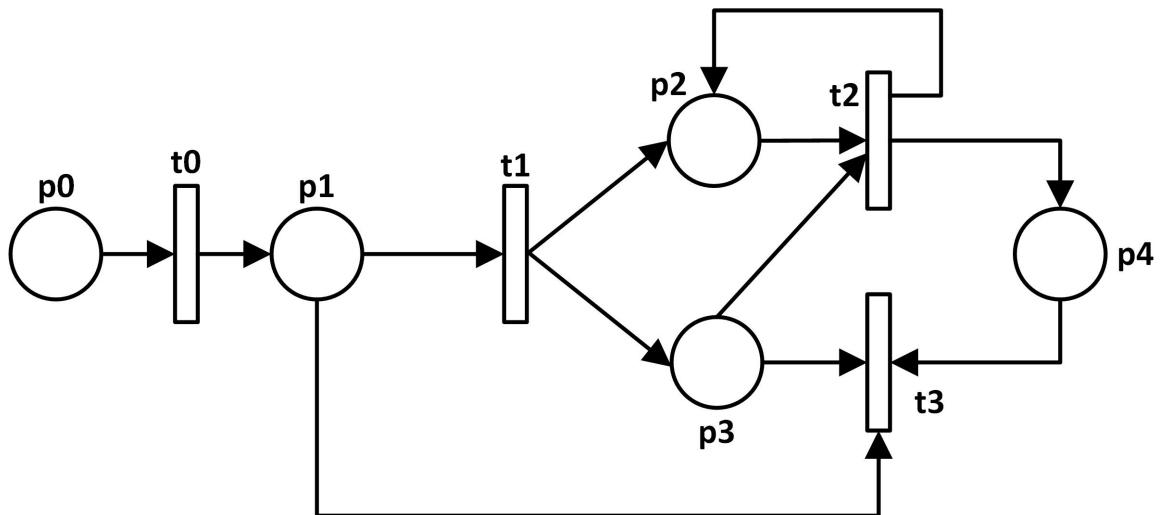
Figure 20.1: Finding siphons and traps.

We shall find the siphons and traps in the Petri net shown in Fig.20.1. Listing-20.1 shows the main simulation file.

Listing 20.1: **Main Simulation File** (Example-39)

```
clear all; clc; close all;

pns = pnstruct('ex_39_pdf');
```

```
SM = siphons_minimal(pns);
disp('Min. Siphons (in matrix form): '); disp(SM);

TM = traps_minimal(pns);
disp('Min. Traps (in matrix form): '); disp(TM);
```

The printout from running the main simulation file is shown below. It identifies **p0** as the sole siphon. If **p0** loses all of its tokens due to the repeated firing of **t0**, of course, **p0** has no chance of getting any tokens from anywhere. Hence, **p0** is a siphon. The printout also identifies **p2** as the sole trap. This is because once **p2** has a token, **p2** will lose this token due to the firing of **t2**. However, **t2** will deposit a new token into **p2**. Hence, **p2** will never become empty (a trap).

```
Minimal siphons in this net:
{p0}
Min.  Siphons (in matrix form):
1 0 0 0 0

Minimal traps in this net:
{p2}
Min.  Traps (in matrix form):
0 0 1 0 0
```

20.8   Example-40: pinvariant and tinvariant
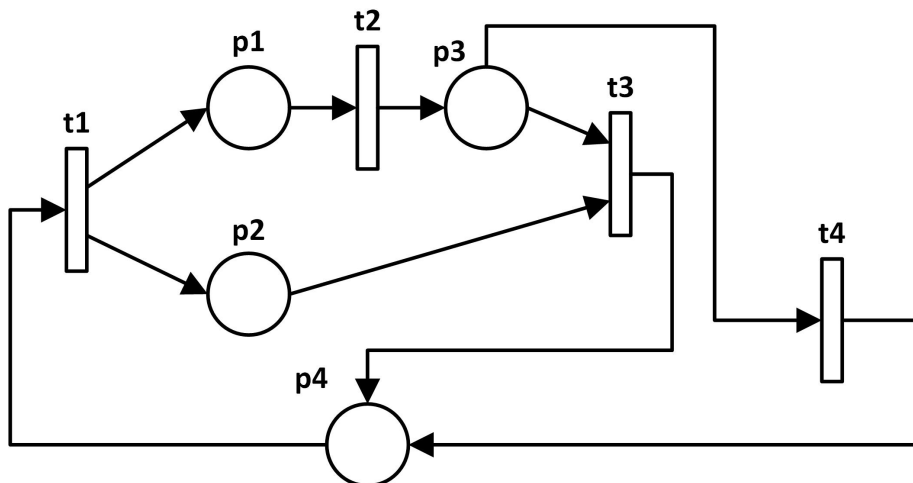
Figure 20.2: Finding P-invariants and T-invariants.

We shall find the P-invariants and T-invariants in the Petri net shown in Fig.20.2. Listing-20.2 shows the main simulation file.

Listing 20.2: **Main Simulation File** (Example-40)

```
pns = pnstruct('ex_40_pdf');

PI = pinvariant(pns); disp(' ');
disp('P-invariants in matrix form: '); disp(PI);

TI = tinvariant(pns); disp(' ');
disp('T-invariants in matrix form: '); disp(TI);
```

The printout from running the main simulation file is shown below. It identifies **p1**, **p3**, and **p4** as the sole P-invariant. This means the weighted sum of tokens in this group remains constant throughout the simulation, immaterial of initial tokens and which transitions fired:

○ **p1** loses a token if **t2** fires. However, **t2** will deposit a token into **p3**.

○ **p3** loses a token if **t3** or **t4** fires. However, these two transitions will deposit a token into **p4**.

○ **p4** loses a token if **t1** fires. However, **t1** will deposit a token into **p1** (and **p2**).

Hence, the sum of tokens **p1**, **p3**, and **p4** always remains the same.

The printout also identifies **t1**, **t2**, and **t3** as the sole T-invariant. This means if these three transitions fire one after the other, then we reach the same state we started with:

○ **t1** consumes a token from **p4** and deposits a token each into **p1** and **p2**.

○ **t2** transfers a token from **p1** to **p3**.

○ **t3** consumes a token each from **p2** and **p3** and deposits a token into **p4**.

Hence, the initial state is reached after the firing of **t1**, **t2**, and **t3**.

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

P-invariants in matrix form:
1 0 1 1

T-invariants:
{t1,t2,t3}

T-invariants in matrix form:
1 1 1 0
```

Bibliography

Davidrajuh, R. (2018). *Modeling Discrete-Event Systems with GPenSIM*, Springer International Publishing, Cham.

Chapter 21

Timer

This chapter presents four simple utility functions for working with time. Table-21.1 presents a summary of these functions.

Function	Description
`compare_time`	compares two time strings.
`current_clock`	returns the computer's real-time clock (the current time), either in seconds or in [hour min sec] format.
`current_time`	returns GPenSIM's global clock (the current time) (same as `PN.current_time`).
`rt_clock_string`	returns current time as a text string in [HH:MM:SS] format.

Table 21.1: Summary of timer functions.

21.1   compare_time

Function name: `compare_time`
Full name: Compare two input time.
Purpose: compares two input times and returns whether the first one is the earliest, the same, or the latest.
Input Parameters: 1) Time-1; 2) Time-2. Accepted input formats are either as a string (e.g. 'unifrdn(10, 12)') or as a vector [hour min sec].
Output Parameter: an integer: -1, if Time-1 is the latest; 0, if bother are equal; +1, if Timer-1 is the earliest.
This function uses: (none).
Sample use:

```
ct = compare_time ([hour1 min1 sec1], [hour2 min2 sec2]);
```

```
if not(ct) % both are equal
...
```

21.2   current_clock

Function name: `current_clock`
Full name: current clock.
Purpose: Returns the real-time clock (Computer's real-time) in seconds or as a vector in [hour min sec] format.
Input Parameter: '1' : output clock in seconds. '3' : output clock as a vector [hour min sec].
Output Parameter: Clock value is returned either as in seconds or a vector [hour min sec].
This function uses: (none).
Further info: Chapter 8, "Interfacing with External Hardware," in Davidrajuh (2018) for real-time simulation.
Sample use:

```
current_clock_HMS = current_clock(3);
disp(current_clock_HMS); % display: hourX minX secX
```

Related functions: `rt_clock_string`

21.3   current_time

Function name: `current_time`
Full name: current time.
Purpose: Returns the current value of the simulated time
(same as `PN.curren_time`).
Input Parameters: (none).
Output Parameter: the value of `PN.curren_time`.
This function uses: (none).
Sample use:

```
current_time = current_time(); % same as PN.curren_time
```

Related functions: `current_clock`

21.4   rt_clock_string

Function name: `rt_clock_string`
Full name: time as a text string.

Purpose: Returns the real-time clock [hour min sec] as a text string for display in HH:MM:SS format (e.g., '10:05:30').
Input Parameters: (none).
Output Parameter: current time as a text string in HH:MM:SS format.
This function uses: (none)
Further info: Chapter 5.5, "Using Hourly Clock," in Davidrajuh (2018) for real-time simulation.
Sample use:

```
disp_str = rt_clock_string();
disp(disp_str); % display '10:43:20'
```

Related functions: current_clock

Bibliography

Davidrajuh, R. (2018). *Modeling Discrete-Event Systems with GPenSIM*, Springer International Publishing, Cham.

Chapter 22

Token-Selection

Colored Petri nets are inevitable for modeling real-world discrete systems, and tokens' colors are the basis of Colored Petri nets. This chapter presents functions for selecting tokens based on color, time, and cost. Tables 22.1, 22.2, and 22.3 present a summary of these functions based on color, time, and cost, respectively.

For a thorough study of Colored Petri nets, see Chapter 2, "Colored Petri Nets: The Basics," in Davidrajuh (2023).

Function	Description
tokenAllColor	Selects only the tokens that have all of the specified colors.
tokenAny	Selects any tokens (without any preference on color).
tokenAnyColor	Selects tokens with any of the specified colors; selected tokens must have at least one of the specified colors.
tokenColorless	Selects only the colorless tokens (tokens with NO color).
tokenEXColor	Selects tokens with **exactly** the same colors as the specified colors.
tokenWOAllColor	Excludes a token with all of the specified colors.
tokenWOAnyColor	Excludes a token with ANY of the specified colors.
tokenWOEXColor	Excludes a token with *exactly* the same colors as the specified colors.
tokIDs	Returns a set of tokIDs of tokens in a place.

Table 22.1: Functions for color-based token selection.

Function	Description
`tokenArrivedBetween`	Selects tokens that were deposited into a place within the given time interval.
`tokenArrivedEarly`	Selects tokens that were deposited earliest into a place.
`tokenArrivedLate`	Selects tokens that were deposited latest into a place.

Table 22.2: Functions for time-based token selection.

Function	Description
`tokenCheap`	Selects tokens that are the cheapest in a specific place.
`tokenCostBetween`	Selects tokens that cost between two limits.
`tokenExpensive`	Selects tokens that are the most expensive in a specific place.

Table 22.3: Functions for cost-based token selection.

22.1   tokenAllColor

Function name: `tokenAllColor`
Full name: tokens with all the specified colors.
Purpose: returns a set of tokens from a specified place, where each token color consists of all the specified colors in the input parameter.
Input Parameters: 1) Place name or index. 2) Number of tokens wanted. 3) set of colors.
Output Parameters: 1) a set of tokIDs ('set_of_tokID'). 2) Number of valid tokIDs in 'set_of_tokID'. Note that the set of tokIDs returned will be the same length as the number of tokens wanted (second input argument); only the first few of 'set_of_tokID' will be valid tokIDs, followed by trailing zeros.
This function uses: `check_valid_place`.
Further info: See chapter 2, "Colored Petri Nets: The Basics," in Davidrajuh (2023).
Sample use:

```
[set_of_tokID, nr_token_av] = tokenAllColor('pBUFFER1', 3, ...
    {'Red','Green','Blue'}})
```

Application example: A simple example ("Example-41: Color-based Token Selection" in Section 22.16) is given at the end of this chapter.
Related functions: `tokenAnyColor, tokenEXColor`

22.2   tokenAny

Function name: `tokenAny`
Full name: any token.
Purpose: returns a set of tokens from a specified place without preference for token colors. In other words, pick up arbitrary tokens from the specified place.
Input Parameters: 1) Place name or index. 2) Number of tokens wanted.
Output Parameters: 1) a set of tokIDs ('set_of_tokID'). 2) Number of valid tokIDs in 'set_of_tokID'. Note that the set of tokIDs returned will be the same length as the number of tokens wanted (second input argument); only the first few of 'set_of_tokID' will be valid tokIDs, followed by trailing zeros.
Note that this function is similar to the function `tokenIDs`, in which the second input argument (Number of tokens wanted) is optional; However, in `tokenAny`, the two input arguments are compulsory.
This function uses: `check_valid_place`.
Further info: See chapter 2, "Colored Petri Nets: The Basics," in Davidrajuh (2023).
Sample use:

```
[set_of_tokID, nr_token_av] = tokenAny('pBUFFER1', 3)
```

Related functions: `tokenIDs`

22.3   tokenAnyColor

Function name: `tokenAnyColor`
Full name: tokens with any of the specified colors.
Purpose: returns a set of tokens from a specified place, where each token color consists of any of the specified colors (one or more of the specified colors) in the input parameter.
Input Parameters: 1) Place name or index. 2) Number of tokens wanted. 3) set of colors.
Output Parameters: 1) a set of tokIDs ('set_of_tokID'). 2) Number of valid tokIDs in 'set_of_tokID'. Note that the set of tokIDs returned will be the same length as the number of tokens wanted (second input argument); only the first few of 'set_of_tokID' will be valid tokIDs, followed by trailing zeros.
This function uses: `check_valid_place`.
Further info: See chapter 2, "Colored Petri Nets: The Basics," in Davidrajuh (2023).
Sample use:

```
[set_of_tokID, nr_token_av] = tokenAnyColor('pBUFFER1', 3, ...
    {'Red','Green','Blue'}})
```

Related functions: `tokenAllColor`, `tokenEXColor`

22.4   tokenColorless

Function name: `tokenColorless`
Full name: tokens without any color (colorless).
Purpose: returns a set of tokens from a specified place, where each token does not possess any color (colorless).
Input Parameters: 1) Place name or index. 2) Number of tokens wanted.
Output Parameters: 1) a set of tokIDs ('set_of_tokID'). 2) Number of valid tokIDs in 'set_of_tokID'. Note that the set of tokIDs returned will be the same length as the number of tokens wanted (second input argument); only the first few of 'set_of_tokID' will be valid tokIDs, followed by trailing zeros.
This function uses: `check_valid_place`.
Further info: See chapter 2, "Colored Petri Nets: The Basics," in Davidrajuh (2023).
Sample use:

```
[set_of_tokID, nr_token_av] = tokenColorless('pBUFFER1', 3);
```

22.5   tokenEXColor

Function name: `tokenEXColor`
Full name: tokens with the exact colors.
Purpose: (EX stands for 'exact') This function returns a set of tokens where each token has the same color set as specified.
Input Parameters: 1) Place name or index. 2) Number of tokens wanted. 3) set of colors.
Output Parameters: 1) a set of tokIDs ('set_of_tokID'). 2) Number of valid tokIDs in 'set_of_tokID'. Note that the set of tokIDs returned will be the same length as the number of tokens wanted (second input argument); only the first few of 'set_of_tokID' will be valid tokIDs, followed by trailing zeros.
This function uses: `check_valid_place`.
Further info: See chapter 2, "Colored Petri Nets: The Basics," in Davidrajuh (2023).
Sample use:

```
[set_of_tokID, nr_token_av] = tokenEXColor('pBUFFER1', 3, ...
    {'Red','Green','Blue'}})
```

Application example: A simple example ("Example-41: Color-based Token Selection" in Section 22.16) is given at the end of this chapter.
Related functions: `tokenAnyColor`, `tokenAllColor`

22.6   tokenWOAllColor

Function name: `tokenWOAllColor`
Full name: tokens without all the specified colors.
Purpose: (WO stands for 'without') returns a set of tokens from a specified place, each token color consisting of none of the specified colors in the input parameter.
Input Parameters: 1) Place name or index. 2) Number of tokens wanted. 3) set of colors.
Output Parameters: 1) a set of tokIDs ('set_of_tokID'). 2) Number of valid tokIDs in 'set_of_tokID'. Note that the set of tokIDs returned will be the same length as the number of tokens wanted (second input argument); only the first few of 'set_of_tokID' will be valid tokIDs, followed by trailing zeros.
This function uses: `check_valid_place`.
Further info: See chapter 2, "Colored Petri Nets: The Basics," in Davidrajuh (2023).
Sample use:

```
[set_of_tokID, nr_token_av] = tokenWOAllColor('pBUFFER1', 3, ...
    {'Red','Green','Blue'}})
```

Related functions: `tokenAllColor`

22.7   tokenWOAnyColor

Function name: `tokenWOAnyColor`
Full name: tokens without any of the specified colors.
Purpose: (WO stands for 'without') returns a set of tokens from a specified place, where each token color does not consist of any of the specified colors (one or more of the specified colors) in the input parameter.
Input Parameters: 1) Place name or index. 2) Number of tokens wanted. 3) set of colors.
Output Parameters: 1) a set of tokIDs ('set_of_tokID'). 2) Number of valid tokIDs in 'set_of_tokID'. Note that the set of tokIDs returned will be the same length as the number of tokens wanted (second input argument);

only the first few of 'set_of_tokID' will be valid tokIDs, followed by trailing zeros.

This function uses: `check_valid_place`.

Further info: See chapter 2, "Colored Petri Nets: The Basics," in Davidrajuh (2023).

Sample use:

```
[set_of_tokID, nr_token_av] = tokenWOAnyColor('pBUFFER1', 3, ...
    {'Red','Green','Blue'}})
```

Application example: A simple example ("Example-41: Color-based Token Selection" in Section 22.16) is given at the end of this chapter.

Related functions: `tokenAnyColor`

22.8   tokenWOEXColor

Function name: `tokenWOEXColor`

Full name: tokens without the exact colors.

Purpose: (EX stands for 'exact' and WO stands for 'without') returns a set of tokens (tokIDs) excluding the ones that have the same colors as specified.

Input Parameters: 1) Place name or index. 2) Number of tokens wanted. 3) set of colors.

Output Parameters: 1) a set of tokIDs ('set_of_tokID'). 2) Number of valid tokIDs in 'set_of_tokID'. Note that the set of tokIDs returned will be the same length as the number of tokens wanted (second input argument); only the first few of 'set_of_tokID' will be valid tokIDs, followed by trailing zeros.

This function uses: `check_valid_place`.

Further info: See chapter 2, "Colored Petri Nets: The Basics," in Davidrajuh (2023).

Sample use:

```
[set_of_tokID, nr_token_av] = tokenWOEXColor('pBUFFER1', 3, ...
    {'Red','Green','Blue'}})
```

Related functions: `tokenEXColor`

22.9   tokIDs

Function name: `tokIDs`

Full name: tokenID numbers.

Purpose: Returns a set of tokIDs of tokens in a place; if the second argument, the number of tokIDs wanted, is not given, tokIDs of all the tokens in

the place is returned.

Input Parameter - Compulsory: Place name or index.

Input Parameter - Optional: Number of tokIDs wanted ('nr_tokIDs_wanted').

Output Parameter: a set of tokIDs ('set_of_tokID'). Note that the set of tokIDs returned will be the same length as the number of tokens wanted (second input argument ´nr_tokIDs_wanted'); only the first few of 'set_of_tokID' will be valid tokIDs, followed by trailing zeros.

This function uses: `check_valid_place`.

Further info: See chapter 2, "Colored Petri Nets: The Basics," in Davidrajuh (2023).

Sample use:

```
set_of_tokID = tokIDs('pBUFFER2', 3);
```

Time-based Token Selection

22.10   tokenArrivedBetween

Function name: `tokenArrivedBetween`

Full name: tokens deposited in the specified place within the specified time interval.

Purpose: returns a set of tokens from a specified place, where each token was deposited within the specified time interval.

Input Parameters: 1) Place name or index. 2) Number of tokens wanted. 3) time interval as a vector [interval_start interval_end]).

Output Parameters: 1) a set of tokIDs ('set_of_tokID'). 2) Number of valid tokIDs in 'set_of_tokID'. Note that the set of tokIDs returned will be the same length as the number of tokens wanted (second input argument); only the first few of 'set_of_tokID' will be valid tokIDs, followed by trailing zeros.

This function uses: `check_valid_place`.

Further info: See chapter 5, "Time-based Token Selection," in Davidrajuh (2023).

Sample use:

```
% Example 13 in Davidrajuh (2023)
% pick three tokens from "pQueue" deposited between 10 and 20 TU
tokIDs = tokenArrivedBetween('pQueue',3, [10, 20]);
```

Related functions: `tokenArrivedEarly, tokenArrivedLate`

22.11   tokenArrivedEarly

Function name: `tokenArrivedEarly`
Full name: tokens deposited earliest.
Purpose: returns a set of tokens deposited earliest to a specified place.
Input Parameters: 1) Place name or index. 2) Number of tokens wanted.
Output Parameters: 1) a set of tokIDs ('set_of_tokID'). 2) Number of valid tokIDs in 'set_of_tokID'. Note that the set of tokIDs returned will be the same length as the number of tokens wanted (second input argument); only the first few of 'set_of_tokID' will be valid tokIDs, followed by trailing zeros.
This function uses: `check_valid_place`.
Further info: See chapter 5, "Time-based Token Selection," in Davidrajuh (2023).
Sample use:

```
% Example 13 in Davidrajuh (2023)
% pick oldest three tokens from "pQueue"
tokIDs = tokenArrivedEarly('pQueue',3);
```

Application example: A simple example ("Example-42: Time-based Token Selection" in Section 22.17) is given at the end of this chapter.
Related functions: `tokenArrivedBetween`, `tokenArrivedLate`

22.12   tokenArrivedLate

Function name: `tokenArrivedLate`
Full name: tokens deposited latest.
Purpose: returns a set of tokens deposited at the latest to a specified place.
Input Parameters: 1) Place name or index. 2) Number of tokens wanted.
Output Parameters: 1) a set of tokIDs ('set_of_tokID'). 2) Number of valid tokIDs in 'set_of_tokID'. Note that the set of tokIDs returned will be the same length as the number of tokens wanted (second input argument); only the first few of 'set_of_tokID' will be valid tokIDs, followed by trailing zeros.
This function uses: `check_valid_place`.
Further info: See chapter 5, "Time-based Token Selection," in Davidrajuh (2023).
Sample use:

```
% Example 13 in Davidrajuh (2023)
% pick newest three tokens from "pQueue"
tokIDs = tokenArrivedLate('pQueue',3);
```

Application example: A simple example ("Example-42: Time-based Token Selection" in Section 22.17) is given at the end of this chapter.
Related functions: `tokenArrivedBetween, tokenArrivedEarly`

Cost-based Token Selection

22.13 tokenCheap

Function name: `tokenCheap`
Full name: cheapest tokens in a specific place.
Purpose: returns a set of tokens that are the cheapest in a specific place.
Input Parameters: 1) Place name or index. 2) Number of tokens wanted.
Output Parameters: 1) a set of tokIDs ('set_of_tokID'). 2) Number of valid tokIDs in 'set_of_tokID'. Note that the set of tokIDs returned will be the same length as the number of tokens wanted (second input argument); only the first few of 'set_of_tokID' will be valid tokIDs, followed by trailing zeros.
This function uses: `check_valid_place`.
Further info: See chapter 6, "Token Selection Based on Cost," in Davidrajuh (2023).
Sample use:

```
% Example 17 in Davidrajuh (2023)
% pick cheapest three tokens from "pBuffer"
tokIDs = tokenCheap('pBuffer',3);
```

Application example: A simple example ("Example-43: Cost-based Token Selection" in Section 22.18) is given at the end of this chapter.
Related functions: `tokenCostBetween, tokenExpensive`

22.14 tokenCostBetween

Function name: `tokenCostBetween`
Full name: tokens that cost between two limits.
Purpose: Returns tokens in a specific place that cost between two limits, the lower cost (lc) and the upper cost (uc).
Input Parameters: 1) Place name or index. 2) Number of tokens wanted. 3) Lower cost limit. 4) Upper cost limit.
Output Parameters: 1) a set of tokIDs ('set_of_tokID'). 2) Number of valid tokIDs in 'set_of_tokID'. Note that the set of tokIDs returned will be the same length as the number of tokens wanted (second input argument); only the first few of 'set_of_tokID' will be valid tokIDs, followed by trailing zeros.

This function uses: `check_valid_place`.
Further info: See chapter 6, "Token Selection Based on Cost," in Davidrajuh (2023).
Sample use:

```
% Example 17 in Davidrajuh (2023)
% three tokens from "pBuffer" that cost between 270 CU and 290 CU
tokID = tokenCostBetween('pBuffer', 3, 270, 290);
```

Related functions: `tokenCheap`, `tokenExpensive`

22.15   tokenExpensive

Function name: `tokenExpensive`
Full name: Most expensive tokens in a specific place.
Purpose: returns a set of most expensive tokens in a specific place.
Input Parameters: 1) Place name or index. 2) Number of tokens wanted.
Output Parameters: 1) a set of tokIDs ('set_of_tokID'). 2) Number of valid tokIDs in 'set_of_tokID'. Note that the set of tokIDs returned will be the same length as the number of tokens wanted (second input argument); only the first few of 'set_of_tokID' will be valid tokIDs, followed by trailing zeros.
This function uses: `check_valid_place`.
Further info: See chapter 6, "Token Selection Based on Cost," in Davidrajuh (2023).
Sample use:

```
% Example 17 in Davidrajuh (2023)
% pick most expensive three tokens from "pBuffer"
tokIDs = tokenExpensive('pBuffer',3);
```

Application example: A simple example ("Example-43: Cost-based Token Selection" in Section 22.18) is given at the end of this chapter.
Related functions: `tokenCostBetween`, `tokenCheap`

22.16   Example-41: Color-based Token Selection

Fig.22.1 shows the **Colored** Petri net we shall use to test some functions for selecting tokens based on color. This Petri net possesses two parts:

○ The upper part adds colors to tokens (**tPainter** adds colors to tokens and deposits them into **pPainted-1** and **pPainted-2**).
○ The lower part is the delay mechanism, involving **pDelay**, **tDelay**, and **pDelayed**; **tDelay** has a large firing time. The lower part is to make
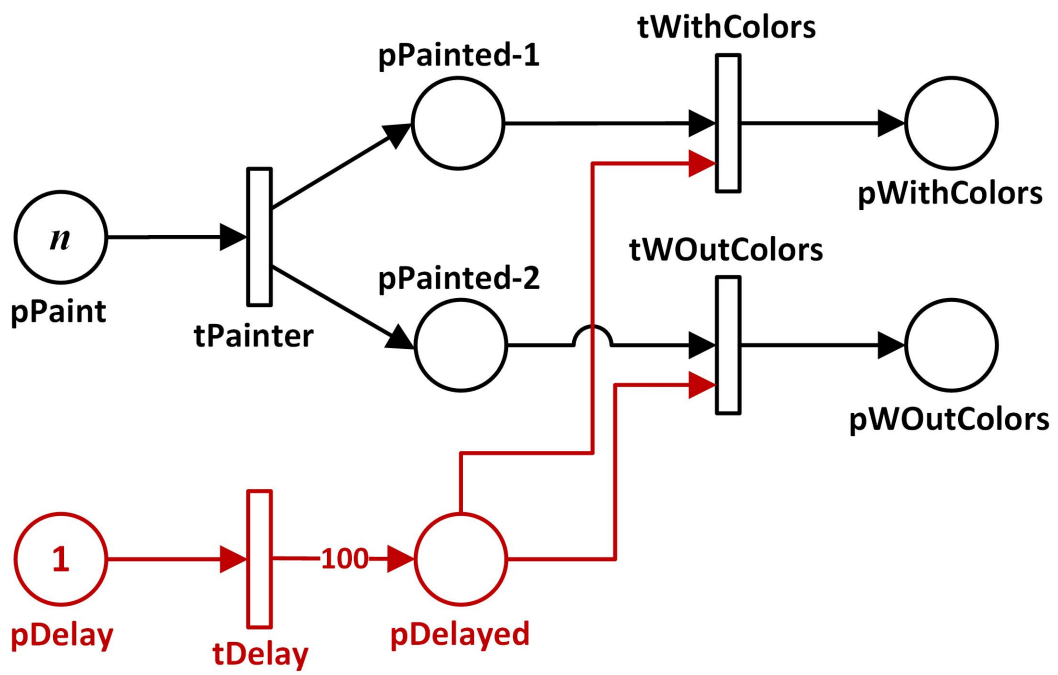
Figure 22.1: Testing functions for color-based token selection.

sure that the selecting transitions (**tWithColors** and **tWOutColors**)
are forced to wait until **tPainter** has completed the coloring of tokens.

tPainter only adds a few colors to the deposited tokens into **pPainted-1**
and **pPainted-2**. The first three colors are single strings: {'Red'}, {'Green'},
and {'Blue'}, followed by a color that has two strings: {'Red', 'Green'}.
The fourth and the fifth colors have three and four strings, respectively:
{'Red','Green','Blue'} and {'Red','Green','Blue','Grey'}.   The final token
will be colorless (**tPainter** adds empty color, {}).

Listing-22.1 shows the main simulation file.

Listing 22.1: **Main Simulation File** (Example-41)

```
global global_info
global_info.STOP_AT = 200;

% colors to add
global_info.cr = {{'Red'}, {'Green'}, {'Blue'},... % one string
    {'Red','Green'}, ... % two strings
    {'Red','Green','Blue'}, ... % three strings
    {'Red','Green','Blue','Grey'},{}}; % four and zero strings

% initial color rotation index
global_info.cr_index = 0;


png = pnstruct('ex_41_pdf');

%%%% initial dynamics %%%%
dyn.m0 = {'pPaint',10, 'pDelay',1};
dyn.ft = {'tDelay',100, 'allothers',1};
pni = initialdynamics(png, dyn);

sim = gpensim(pni);
prnfinalcolors(sim, {'pWithColors','pWOutColors'});
```

In this example, we are only using specific pre-processors to avoid a large
common pre-processor. The pre-processor for **tPainter** (Listing-22.2) adds
colors given in the global variable, 'global_info.cr', to the tokens deposited
in **pPainted-1** and **pPainted-2**.

Listing 22.2: **tPainter_pre** (Example-41)

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

% tPaint paints only seven colors (seven times)
ntP = timesfired('tPainter');
% colors are finished, no more
if eq(ntP, length(global_info.cr))
    fire = 0;
```

```
      return
end

global_info.cr_index = global_info.cr_index + 1;
transition.new_color = global_info.cr{global_info.cr_index};
fire = 1;
```

Transition **tWithColors** fires only two times. The first time it fires, it will ask for a token with *exactly* three colors (function `tokenEXColor`): {'Red', 'Green', 'Blue'}. There is only one token in **pPainted-1** with three specified colors only - color number four. Hence, this token will be selected and deposited into **pWithColors**. The second time **tWithColors** fires, it asks for a token with *all* three colors (function `tokenAllColor`); if a token has more colors than these three, then it is also OK. In this case, there is only one token that can satisfy this demand, and it has four colors - color number five; this token will be selected and deposited into **pWithColors**. The pre-processor tWithColors_pre is shown in Listing-22.3,

Listing 22.3: **tWithColors_pre** (Example-41)

```
function [fire, transition] = tWithColors_pre(transition)
nt = timesfired(transition.name);
switch nt
    case 0
        tokID1 = tokenEXColor('pPainted-1',1,...
            {'Red','Green','Blue'});
    case 1
        tokID1 = tokenAllColor('pPainted-1',1,...
            {'Red','Green','Blue'});

    otherwise
        tokID1 = 0;
end

transition.selected_tokens = tokID1;
fire = tokID1;  % must have the specified colors
```

Transition **tWOutColors** fires only once. In the pre-processor, **tWOutColors** requests a token that does not possess any of the three colors {'Red', 'Green', 'Blue'}. There is only one token that can satisfy this demand, and it is the colorless token. Hence, this token will be selected and deposited into **pWOutColors**. The pre-processor tWOutColors_pre is shown in Listing-22.4.

Listing 22.4: **tWOutColors_pre** (Example-41)

```
function [fire, transition] = tWOutColors_pre(transition)
nt = timesfired(transition.name);
switch nt
```

```matlab
    case 0
        tokID1 = tokenWOAnyColor('pPainted-2', 1,...
            {'Red','Green','Blue'});
    otherwise
        tokID1 = 0;
end

transition.selected_tokens = tokID1;
fire = tokID1;  % must have the specified colors
```

The printout of the main simulation file confirms the correct selection of the tokens.

```
**** **** Colors of Final Tokens ...
No.  of final tokens:  34

Place:  pWithColors
Time:  101 Colors:  "Blue" "Green" "Red"
Time:  102 Colors:  "Blue" "Green" "Grey" "Red"


Place:  pWOutColors
Time:  101 *** NO COLOR ***
```

22.17   Example-42: Time-based Token Selection

Fig.22.2 shows the Colored Petri net we shall use to test some functions for selecting tokens based on time. As in the previous example, this Petri net also possesses two parts:

- ○ The upper part adds colors to tokens; **tTimer** adds the current time as a color to tokens and deposits them into **pTimed**.
- ○ The lower part is the delay mechanism, involving **pDelay**, **tDelay**, and **pDelayed**; **tDelay** has a large firing time. The lower part is to make sure that the selecting transitions (**tEarliest** and **tLatest**) are forced to wait until **tTimer** has completed adding time as the color to tokens.

Listing-22.5 shows the main simulation file.

Listing 22.5: **Main Simulation File** (Example-42)

```matlab
global global_info
global_info.STOP_AT = 200;

spng = pnstruct('ex_42_pdf');
dyn.m0 = {'pStart',10, 'pDelay',1};
dyn.ft = {'tDelay',100, 'allothers',1};
```
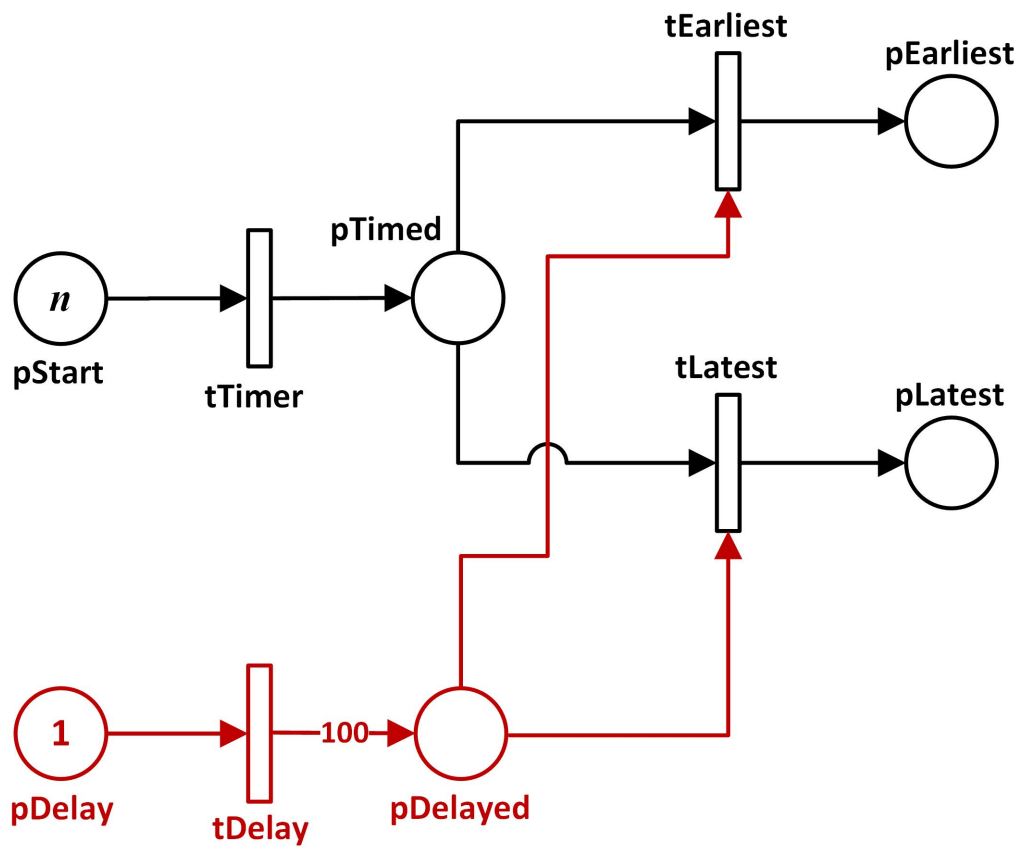
Figure 22.2: Testing functions for time-based token selection.

```
pni = initialdynamics(spng, dyn);
sim = gpensim(pni);
prnfinalcolors(sim, {'pEarliest','pLatest'});
```

All the enabled transitions are handled in the COMMON_PRE (Listing-22.6). In COMMON_PRE:

- **tTimer** is allowed to fire as many times as there are tokens in its input place **pStart**. However, **tTimer** adds the current time (function `current_time`) as the token's color.
- The two selection transitions (**tEarliest** and **tLatest**) are allowed to fire only three times.
- When **tEarliest** is enabled, it asks for the earliest token that arrived in **pTimed** (FIFO).
- When **tLatest** is enabled, it asks for the latest token that arrived in **pTimed** (LIFO).

Listing 22.6: **COMMON_PRE** (Example-42)

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

% Selecting trans (tEariest and tLatest)
% only three times
ntP = timesfired(transition.name);
if ismember(transition.name, {'tEarliest','tLatest'})
    if eq(ntP, 3)
        fire = 0;
        return
    end
end

switch transition.name
    case 'tTimer'
        transition.new_color = num2str(current_time());
        tokID1 = 1; % always let tTimer fire

    case 'tEarliest'
        tokID1 = tokenArrivedEarly('pTimed', 1);

    case 'tLatest'
        tokID1 = tokenArrivedLate('pTimed', 1);

    case 'tDelay'
        tokID1 = 1; % let tDelay fire
end

transition.selected_tokens = tokID1;
fire = tokID1;  % must have the specified colors
```

The printout of the main simulation file confirms the correct selection of the tokens.

```
**** **** Colors of Final Tokens ...
No.  of final tokens:  24


Place:  pEarliest
Time:  101 Colors:  "0"
Time:  102 Colors:  "1"
Time:  103 Colors:  "2"


Place:  pLatest
Time:  101 Colors:  "9"
Time:  102 Colors:  "8"
Time:  103 Colors:  "7"
```

22.18 Example-43: Cost-based Token Selection

Fig.22.3 shows a production facility where some production machines are employed:

○ The machines are arranged in assembly lines (Lines 1 to 3), and the products that end up in the output cartridge (**pOUT**) are the same.
○ However, the machines **M11** to **M33** work at different speeds, and these were purchased at different times in the factory's history.
○ Since the machines were purchased at different times, they also incur different costs (both fixed and variable costs).

The Petri net is to find the cheapest and most expensive production paths. Note that, like the previous two examples in this chapter, Petri net also employs a delay mechanism to freeze the selection transitions **tCheap** and **tExpen** until all three sample products (tokens) arrive in **pOUT**.

Listing-22.7 shows the main simulation file, and Listing-22.8 shows the COMMON_PRE.

Listing 22.7: **Main Simulation File** (Example-43)

```
global global_info
global_info.STOP_AT = 200;

spng = pnstruct('ex_43_pdf');
dyn.m0 = {'p1In',1,'p2In',1,'p3In',1, 'pDelay',1};
dyn.ft = {'tDelay',100, 'allothers', 1};
dyn.fc_fixed = {'M11',76, 'M12',62, 'M13',48, ...
    'M21',59, 'M22',29, 'M23',50, ...
    'M31',74, 'M32',12, 'M33',23};
```
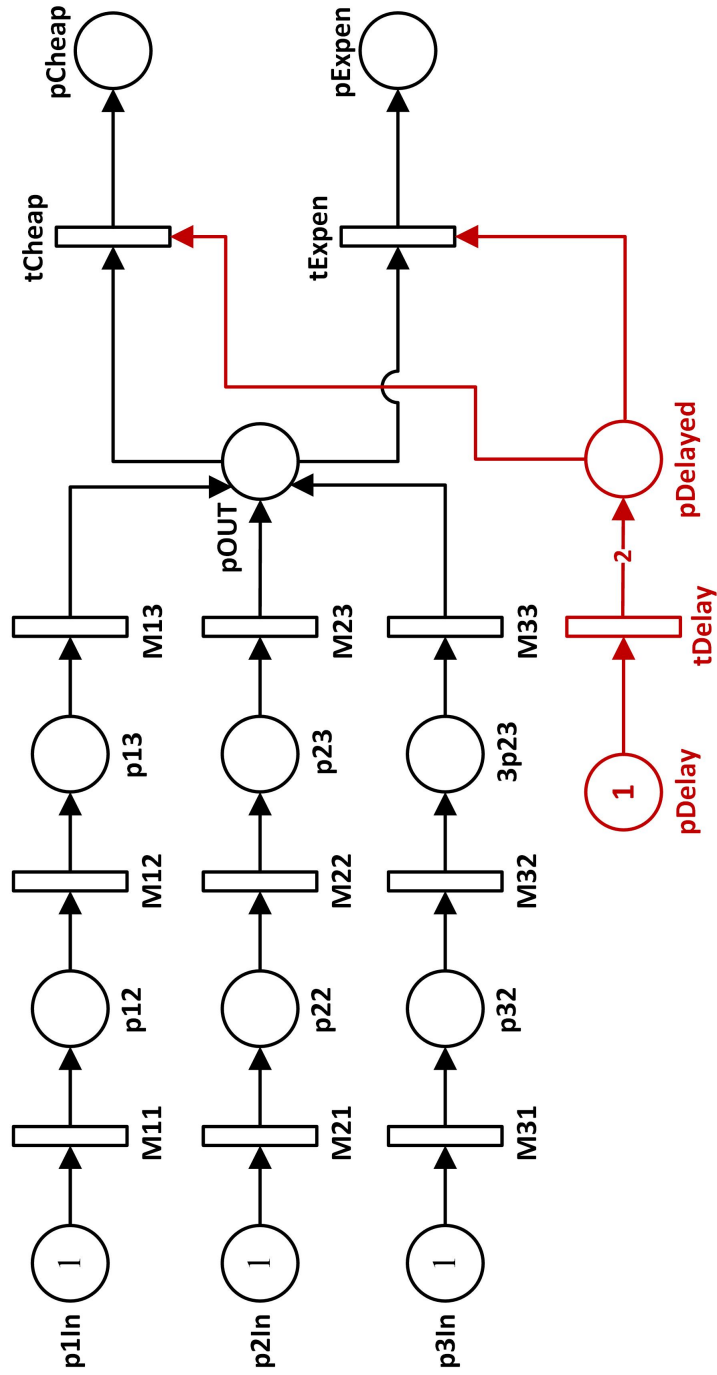
Figure 22.3: Testing functions for time-based token selection.

```matlab
pni = initialdynamics(spng, dyn);

sim = gpensim(pni);
prnfinalcolors(sim, {'pCheap','pExpen'});
```

Listing 22.8: **COMMON_PRE** (Example-43)

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

switch transition.name
    case 'tCheap'
        % chose the cheapest token form pOUT
        tokID1 = tokenCheap('pOUT',1);

    case 'tExpen'
        % chose the expensive token form pOUT
        tokID1 = tokenExpensive('pOUT', 1);

    case 'tDelay'
        tokID1 = 1; % let tDelay fire

    otherwise
        % all others – machine M11 to M33 add their
        % name to the path
        transition.new_color = transition.name;
        tokID1 = 1; % let them fire
end
transition.selected_tokens = tokID1;
fire = tokID1;  % must have the specified colors
```

The simulation result (shown below) indicates that assembly line 3 is the cheapest (a product costs 109 CU), and assembly line 1 is the most expensive (a product costs 186 CU).

```
**** **** Colors of Final Tokens ...
No.  of final tokens:  3


Place:  pCheap
Time:  101 Colors:  "M31" "M32" "M33" Cost:  109


Place:  pExpen
Time:  101 Colors:  "M11" "M12" "M13" Cost:  186
```

Bibliography

Davidrajuh, R. (2023). *Colored Petri Nets for Modeling of Discrete Systems: A Practical Approach With GPenSIM*, Springer Nature.

Chapter 23

Utility-Functions

This chapter - the final one - presents some more utility functions. Readers may find that some of the functions are useful and some are not. However, as the author of this book (and developer of GPenSIM), I want to list all these functions for the sake of completion. Table-23.1 presents a summary of these functions.

Function	Description
`arcweight,` `arcweightPT` and `arcweightTP`	Returns the weight of an arc from an element (place or transition) to another element (place or transition).
`combinatorics`	All combinations of the input vector; each combination becomes a row of a matrix.
`dispMultipleCR`	This functions displays multiple carriage-returns (given as the input parameter) on the screen.
`pnclass`	Prints the class of Petri net (e.g., Binary, State Machine, Marked (Event) Graph, Timed Petri Net, Strongly Connected component, etc.).
`randomgen`	Randomly rearranges the input set (e.g., [1 2 3] becomes [3 1 2]).
`wakeup` and `util_wakup`	Makes a wake-up sound to draw the users' attention.

Table 23.1: Summary of utility functions.

23.1   arcweight (new in version 11)

Function name: `arcweight`
Full name: weight of arc between two elements.

Purpose: the function that returns the weight of an arc from an element (place or transition) to another element (transition or place).
Input Parameters: 1) Place name or transition name (text string, index NOT accepted). 2) Transition name or Place name (text string, index NOT accepted).
Output Parameter: Weight of the arc between the first and second element.
This function uses: (none).
Sample use:

```
disp(['Arc weight (t1, p2) : ', ...
    int2str(arcweight('t1','p2'))]);
```

Application example: A simple example ("Example-44: arcweight" in Section 23.16) is given at the end of this chapter.
Related functions: `arcweightPT, arcweightTP`

23.2   arcweightPT

Function name: `arcweightPT`
Full name: weight of arc between a place and a transition.
Purpose: This is a low-level function that returns the weight of an arc from a given place to a given transition.
Input Parameters: 1) Place index (name as text string NOT accepted). 2) Transition index (name as text string NOT accepted).
Output Parameter: Weight of the arc between the given place and the transition.
This function uses: (none).
Sample use:

```
disp(['Arc weight between "pBuff1" (index 33) and ...
      "tMachine1" (index 5) is ', arcweightPT(33, 5)]);
```

Application example: A simple example ("Example-44: arcweight" in Section 23.16) is given at the end of this chapter.
Related functions: `arcweight, arcweightTP`

23.3   arcweightTP

Function name: `arcweightTP`
Full name: weight of arc between a transition and a place.
Purpose: This is a low-level function that returns the weight of an arc from a given transition to a given place.

Input Parameters: 1) Transition index (name as text string NOT accepted). 2) Place index (name as text string NOT accepted).
Output Parameter: Weight of the arc between the given transition and the place.
This function uses: (none).
Sample use:

```
disp(['Arc weight between "tMachine1" (index 5) and ...
      "pBuff1" (index 33) is ', arcweightTP(5, 33)]);
```

Application example: A simple example ("Example-44: arcweight" in Section 23.16) is given at the end of this chapter.
Related functions: `arcweight`, `arcweightPT`

23.4   combinatorics

Function name: `combinatorics`
Full name: all combinations of the input vector.
Purpose: Using dynamic programming, this function will compute all the combinations of the input vector.
For example, if the input is [3 1 2], the output will be a matrix with each row a new combination as shown below:
2 1 3
1 2 3
1 3 2
2 3 1
3 2 1
3 1 2
Input Parameter: A vector. E.g., [-3 5.2 3.1]
Output Parameter: A matrix in which each row represents a new combination of the elements in the input vector.
This functions uses: (none)
Sample use:

```
transition_indices = [7 9 11];
% randomize
all_random_combinations = combinatorics(transition_indices);
```

23.5   dispMultipleCR

Function name: `dispMultipleCR`
Full name: print multiple carriage returns.

Purpose: Sometimes, we need to separate two displays on screen with multiple carriage returns. This function does that. **Input Parameter**: Number of carriage returns needed to be printed.

Output on screen: Multiple carriage returns (blank lines) will be printed on the screen.

This function uses: (none).

Sample use:

```
disp('this is the first message');
dispMultipleCR(5);
disp('this second message comes after 5 blank lines ');
```

23.6   dispSetOfPlaces

Function name: `dispSetOfPlaces`

Full name: display a set of places.

Purpose: This function will print the names of places identified by their indices.

Input Parameters: 1) Header line. 2) Set of places identified by their indices.

Output to screen: a set of places will be printed on the screen. For example, let the inputs be 'The following places are source places: ' and [3 1 2]. Assume the names of places identified by their indices are 'p3', 'p1', and 'p2'. Then, the following message will be printed on the screen.

The following places are source places:
p3
p1
p2

This function uses: (none)

Sample use:

```
dispSetOfPlaces('The three sink places are: ', [2, 8, 11]);
```

Related functions: `dispSetOfTrans`

23.7   dispSetOfTrans

Function name: `dispSetOfTrans`

Full name: display a set of transitions.

Purpose: This function will print the names of transitions identified by their indices.

Input Parameters: 1) Header line. 2) Set of transitions identified by their indices.

Output to screen: a set of transition names will be printed on the screen. For example, let the inputs be 'The following transitions are cold (source) transitions: ' and [3 1 2]. Assume the names of transition identified by their indices are 'tC', 'tA', and 'tB'. Then, the following message will be printed on the screen.

The following transitions are cold (source) transitions:
tC
tA
tB

This function uses: (none).
Sample use:

```
dispSetOfTrans('The three sink transitions are: ', [5, 18, 32]);
```

Related functions: `dispSetOfPlaces`

23.8   goodname

Function name: `goodname`
Full name: make a better string of the given input text string.
Purpose: To make the length of a text string to a predefined size. If the input length exceeds the predefined size, the string will be chopped. If the length is less, the string will be padded with blanks.
For example, let us assume the input string is 'Reggie.' `goodname('Reggie', 20)` will return a string 'Reggie ' (containing 14 trailing spaces), whereas `goodname('Reggie', 2)` will return 'Re'.
Also, alignment can be fed as the third and optional input argument. For example,
`goodname('Reggie',2, 'l')` (left alignment) will return 'Reggie ';
Whereas, `goodname('Reggie',2, 'r')` (right alignment) will return ' Reggie'.
Input Parameters: 1) text string. 2) Length of string to be returned. 3) Optional: Alignment: 'l' for left and 'r' for right.
Output Parameter: A modified string.
This function uses: (none)
Sample use:

```
disp(['Name in 10 charater length, aligned to left:', ...
    goodname('Reggie', 10, 'l')]);
```

23.9   pnclass

Function name: `pnclass`
Full name: Petri Net Class.
Purpose: checks the class of Petri net. This function returns a vector of flags representing the following information (output variable):
flag-1: Binary (Ordinary) or Generalized Petri Net
flag-2: State Machine
flag-3: Marked (Event) Graph
flag-4: Timed Petri Net
flag-5: Number of Strongly Connected components

   Input Parameter: Petri net with initial dynamics (`pni`, output of `initialdynamics`); alternatively, Static Petri net graph (`spng`, output of `pnstruct`) can also be input.
Output Parameter: A vector with a set of flags, as described above.
Output on screen: The class of Petri Net will be printed.
This function uses: `stronglyconn`.
Further info: Chapter 6, "Petri net Extension," in Davidrajuh (2018), Chapter 11, "Discrete Systems as Petri Modules," Davidrajuh (2021), and Davidrajuh (2023).
Sample use:

```
% in a processor file
global PN
[classtype] = pnclass(PN);
disp(['PN Class: ', int2str(classtype)]);
```

Application example: A simple example ("Example-45: pnclass" in Section 23.17) is given at the end of this chapter.

23.10   randomgen

Function name: `randomgen`
Full name: Randomly rearrange.
Purpose: This function randomly rearranges the input set. For example, if the vector [1 2 3] is input, then the output will be one of the six possibilities.
Input Parameter: A vector of numbers.
Output Parameter: A vector containing the elements of the input vector rearranged.
This function uses: (none).
Sample use:

```
% input a vector of numbers
input_vector = [1:5];
[output_vector] = randomgen(input_vector);
disp(['Input vector ', int2str(input_vector), ...
        ' rearranged is : ', int2str(output_vector)]);
```

23.11   prnerrormsg

Function name: `prnerrormsg`
Full name: Print error message.
Purpose: During the simulation or before or after, we may want to catch
an error and display it clearly using a standard format. This simple function
will do that.
Input Parameter: Error message (a text string).
Output to screen: The error message will be displayed on the screen.
This function uses: (none).
Sample use:

```
...
% catch an error
if not_happened
    error_msg = 'Did not happen';
    prnerrormsg(error_msg);
end
...
```

23.12   search_names

Function name: `search_names`
Full name: Search a set of names.
Purpose: finds the indices of an element (name) within a set of names. If
an element is not found within the set of names, 0 is returned.
Input Parameters: 1) Name; 2) Set of names to be searched.
Output Parameter: Index of name within the set, if found. Otherwise, 0
will be returned.
This function uses: (none).
Sample use:

```
% Assume "Set_of_Names" be {'alfa', 'beta', 'Δ', 'alfa'}
% and a variable "var1" be 'gamma'
[element_nr] = search_names(var1, Set_of_Names);
if any(element_nr)
    disp([var1, ' is in the set!']);
```

```
else
    disp([var1, ' is NOT in the set!']);
end
```

23.13   string_HH_MM_SS

Function name: `string_HH_MM_SS`
Full name: Convert seconds to a string of "Hour:Minute:Sec" format.
Purpose: converts seconds to string '[HH MM SS]'.
Input Parameter: Seconds.
Output Parameter: A text string of '[HH MM SS]' format.
This function uses: (none).
Sample use:

```
% Assume Time1 is in seconds.
% convert Time1 to a text string
display_string = string_HH_MM_SS(Time1);
disp([' Time1 is :', display_string]);
```

23.14   util_wakeup

Same as the function `wakeup` described below.

23.15   wakeup

Function name: `wakeup`
Full name: Wake-up.
Purpose: Makes a wake-up sound. This function draws the users' attention.
It uses MATLAB's built-in function `chirp` to make sound. This function is
useful if the simulation (`gpensim`) takes a long time. In the min simulation
file, if we put this `wakeup` after the function `gpensim`, it will notify the
user that the simulation is complete.
NOTE: This function was named 'util_wakeup' before.
Input Parameter: (none)
Output: An audio warning sound.
This function uses: (MATLAB's built-in function `chirp`).
Sample use:

```
% In main simulation file:
sim_results = gpensim(pni);
wakeup(); % let the user know that the simulation  is complete
```

Related functions: `util_wakeup`

23.16   Example-44: arcweight
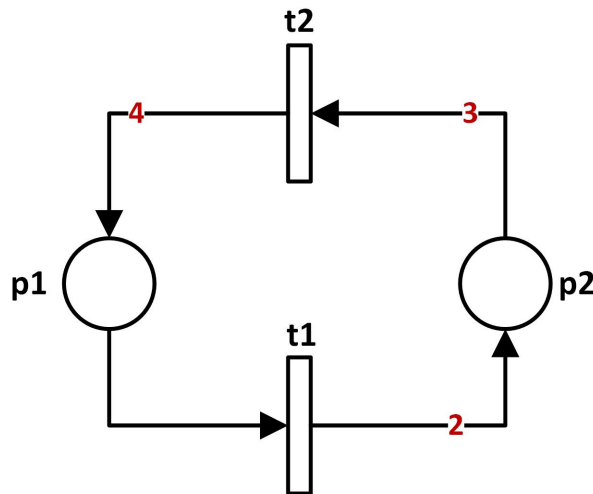
Figure 23.1: Petri net for testing functions for finding arc weights.

Fig.23.1 shows a Petri net for testing the newer utility functions. Listing-23.1 shows the main simulation file.

Listing 23.1: **Main Simulation File** (Example-44)

```
spng = pnstruct('ex_44_pdf');

%%%% function "arcweightPT"
p1_index = is_place('p1'); % get place index
t1_index = is_trans('t1'); % get trans index
aw_p1t1 = arcweightPT(p1_index, t1_index);
disp(['arc weight (p1, t1) = ', int2str(aw_p1t1)]);

%%%% function "arcweightTP"
p2_index = is_place('p2'); % get place index
aw_t1p2 = arcweightTP(t1_index, p2_index);
disp(['arc weight (t1, p2) = ', int2str(aw_t1p2)]);


%%%% function "arcweight"
aw_p2t2 = arcweight('p2', 't2');
disp(['arc weight (p2, t2) = ', int2str(aw_p2t2)]);
aw_t2p1 = arcweight('t2', 'p1');
disp(['arc weight (t2, p1) = ', int2str(aw_t2p1)]);
```

The arcweightPT and arcweightTP functions are low-level functions that accept only place and transition **indices**. For example, if we know the index of **t1** (function is_trans returns the index of a transition) and the

index of **p2** (function `is_place` returns the index of a place), then using
`arcweightTP`, we can get the weight of the arc from **t1** to **p2**. Similarly,
`arcweightPT` will return the weight of the arc from **p2** to **t1**.

However, if we want to use the names (text strings) rather than indices,
the `arcweight` is the function to use.

The simulation run result is shown below.

```
arc weight (p1, t1) = 1
arc weight (t1, p2) = 2
arc weight (p2, t2) = 3
arc weight (t2, p1) = 4
```

23.17   Example-45: pnclass

We shall use the same Petri net that is used in Example-44 (Section 23.16).
The main simulation file is given below:

Listing 23.2: **Main Simulation File** (Example-45)

```
spng = pnstruct('ex_45_pdf');

dyn.m0 = {'p1', 1};
dyn.ft = {'allothers', 1};
pni = initialdynamics(spng, dyn);

Flags = pnclass(pni);
disp(' ');
disp('Returned Flags:'); disp(Flags);
```

The simulation run result is shown below.

```
 **** Example-45:  testing function "pnclass"
****
This is a Generalized Petri Net.
This is a Petri Net State Machine.
This is an Event Graph (Marked Graph).
This is a Timed Petri net.
This is a Strongly Connected Petri net.

Returned Flags:
0 1 1 1 1 0 0 0
```

Bibliography

Davidrajuh, R. (2018). *Modeling Discrete-Event Systems with GPenSIM*, Springer International Publishing, Cham.

Davidrajuh, R. (2021). *Petri Nets for Modeling of Large Discrete Systems*, Springer.

Davidrajuh, R. (2023). A classifier of petri nets for gpensim environment, *2023 7th International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT)*, IEEE, pp. 1–6.

Appendices

Appendix A

GPenSIM Compiler OPTIONS

Table-A.1 presents some of the compiler OPTIONS in GPenSIM. OPTIONS are added on 'global_info' in the main simulation file. For details, see chapter 5, "Optimizing Simulations with GPenSIM," Davidrajuh (2018))

OPTION	Meaning
DELTA_TIME	the sampling frequency.
FIRING_SEQ, FS_REPEAT, FS_ALLOW_PARALLEL	options for firing sequence.
HOURLY_CLOCK	to activate the hourly (business) clock that runs in HH:MM:SS format.
MAX_LOOP	the maximum number of simulation loops.
PRINT_LOOP_NUMBER	print loop numbers during simulation.
REAL_TIME	to use the computer's real-time clock.
STARTING_AT	simulation start time.
STOP_AT	simulation stop time.
STOP_SIMULATION	to (abruptly) stop simulation.

Table A.1: Compiler OPTIONS in GPenSIM.

Bibliography

Davidrajuh, R. (2018). *Modeling Discrete-Event Systems with GPenSIM*, Springer International Publishing, Cham.

Appendix B

Reserved Words in GPenSIM

GPenSIM consists of a large number of functions. To fully utilize its facilities, it is important to avoid using these functions' names as variable names in files created by the user.

There are some reserved words in GPenSIM (GPenSIM uses these words in its systems files). Using these reserved words as variables' names is prohibited; otherwise, simulations may not start at all. Table-B.1 presents the reserved words.

Function	Description
PN	PN is a structure representing a run-time Petri net. PN is a global variable visible in all system files.
global_info	As the name depicts, global_info is also a global variable; global_info is used to set the global OPTIONS (see in Appendix-A); also, it can be used to pass user-defined variables between different files.
function names such as **'pnstruct'**, **'initialdynamics'**, **'gpensim'**	some of the main function names are reserved words as without these functions, there is no way to start the simulations.
COMMON_PRE, COMMON_POST	GPenSIM will automatically call these two files if the common processors are available.
'allothers'	In the main simulation file, 'allothers' is a keyword for assigning the firing times for the unspecified transitions.

Table B.1: GPenSIM Reserved Words.

For details, see Section 2.9, "GPenSIM Reserved Words," in Davidrajuh (2018).

Bibliography

Davidrajuh, R. (2018). *Modeling Discrete-Event Systems with GPenSIM*, Springer International Publishing, Cham.

Appendix C

GPenSIM Webpage
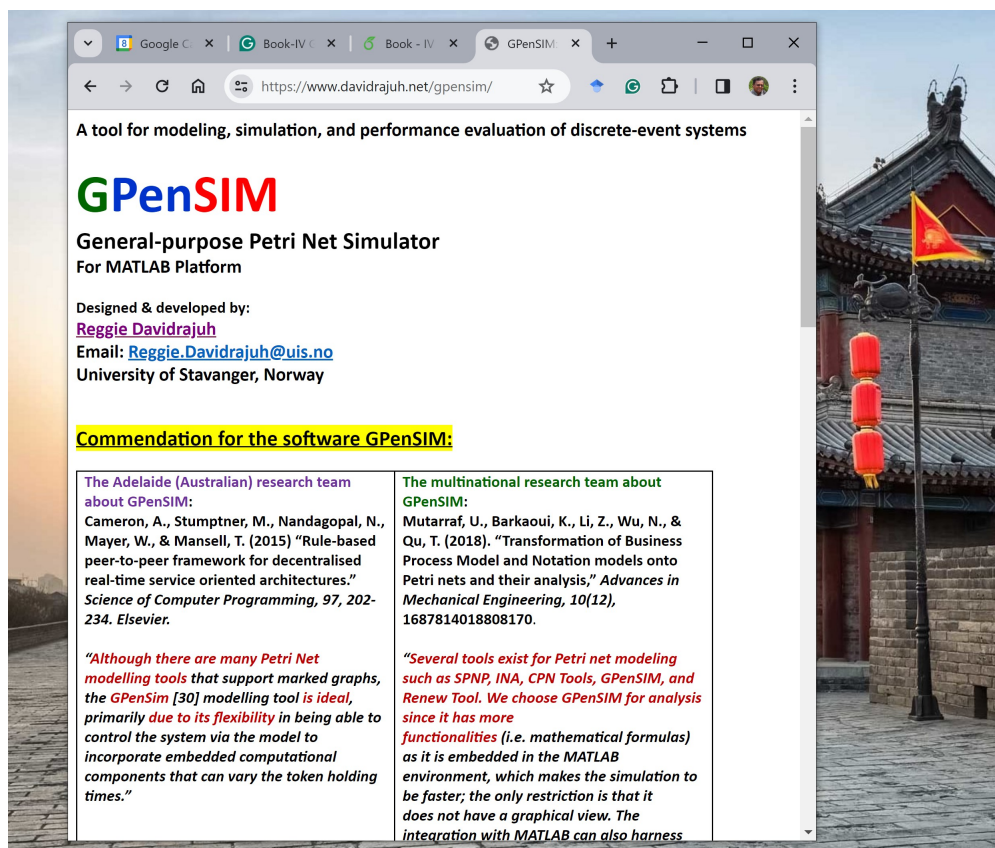
Figure C.1: GPenSIM Website

GPenSIM's website is `http://www.davidrajuh.net/gpensim/`. This website (Fig.C.1) presents some material on GPenSIM; for example, GPenSIM systems files (version 10) can be freely downloaded from this web-

site. Also, this website functions as a companion website for the three books published on GPenSIM.

Figure C.2 shows the basic book on GPenSIM ("**Modeling Discrete-Event Systems with GPenSIM**," Davidrajuh (2018)), which can be considered as a simple user manual. The examples in this book (source code) can be downloaded from the GPenSIM website.
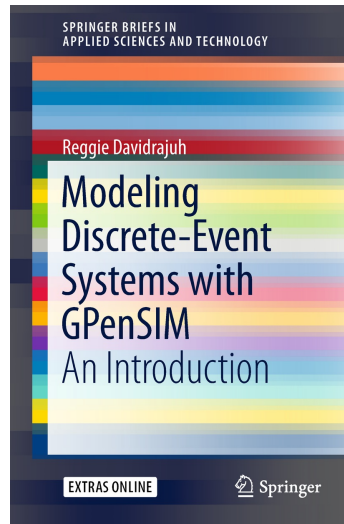
Figure C.2: Basic Book on GPenSIM.

Figure C.3 shows the second book on GPenSIM ("**Petri Nets for Modeling Large Discrete Systems**," Davidrajuh (2021)). This book discusses modular model building with a new Modular Petri Net theory, focusing on modeling large discrete systems. The examples in this book (source code) can also be downloaded from the GPenSIM website.

Figure C.4 shows the third book on GPenSIM ("**Colored Petri Nets for Modeling of Discrete Systems**," Davidrajuh (2023)). This book is for modeling real-world discrete systems for which Colored Petri Nets are needed. This book focuses on coloring of tokens, resources, and cost calculation.

Bibliography

Davidrajuh, R. (2018). *Modeling Discrete-Event Systems with GPenSIM*, Springer International Publishing, Cham.

Davidrajuh, R. (2021). *Petri Nets for Modeling of Large Discrete Systems*, Springer.
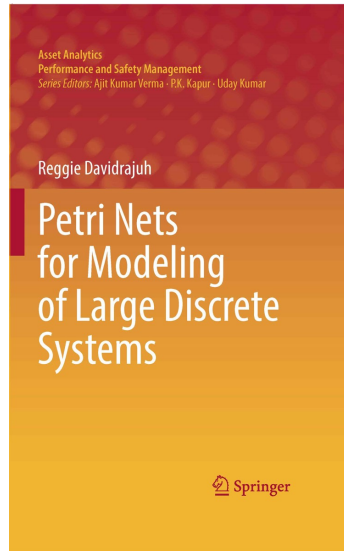
Figure C.3: Book on Modeling Large Discrete Systems with GPenSIM.
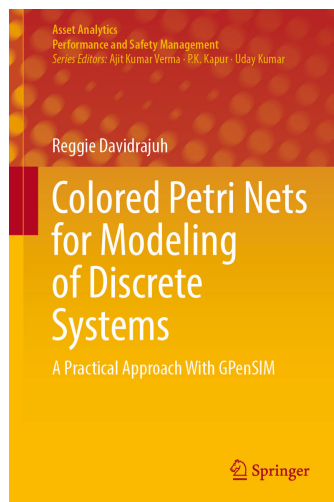
Figure C.4: Book on Colored Petri Nets with GPenSIM.

Davidrajuh, R. (2023). *Colored Petri Nets for Modeling of Discrete Systems: A Practical Approach With GPenSIM*, Springer Nature.

List of Figures

193

List of Tables

Index