Addendum-1 to the book **Davidrajuh, R. "Modeling Discrete-Event Systems with GPenSIM: An Introduction", Springer, 2018**

Colored Petri Nets in GPenSIM

By Reggie Davidrajuh, reggie.davidrajuh@uis.no, © 12 June 2018.

In P/T Petri Nets, tokens inside a place are indistinguishable: all the tokens inside a place are homogenous (the same). It does not matter which token arrived at the place first or last. It does not matter either whether a token is deposited in the place by one transition or another. However, this document introduces coloring of tokens in GPenSIM by which each token can become a unique one, identifiable with a unique token identification number (tokID). Also, we can add some tags ('colors') to each token. This document describes how Colored Petri Net can be realized with GPenSIM.

1. Coloring Tokens

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

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

1.1 Structure of a Token

A token has a structure that consists of three elements:

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

E.g.:

```
tokID: 101
```

```
creation_time: 30.25
      t_color: {'Tamil', 'Norwegian', 'English', 'German'}
```

1.2 Functions for selection of tokens based on their colors

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

Table-1-1: GPenSIM functions for manipulation of token color

Function	*Description*
tokenAllColor	Select only the tokens that have all of the specified colors.
tokenAny	Select any tokens (without any preference on color).
tokenAnyColor	Select tokens with any of the specified colors; selected tokens must have at least one of the specified color.
tokenArrivedBetween	Select tokens that were deposited into a place between the stated time intervals.
tokenArrivedEarly	Select tokens that were deposited earliest into a place.
tokenArrivedLate	Select tokens that were deposited latest into a place.
tokenColorless	Select only the colorless tokens (tokens with NO color).
tokenEXColor	Select tokens with **exact** colors (no more or no less).
tokenWOAllColor	Exclude a token ONLY if it has all of the specified colors.
tokenWOAnyColor	Exclude a token ONLY if it has ANY of the specified colors.
tokenWOEXColor	Exclude a token ONLY if it has the *exact* colors as specified.
tokIDs	Returns a set of tokIDs of tokens in a place; if the second argument 'nr_tokIDs_wanted' is not specified, then tokIDs of all the tokens in the place is returned.
prnfinalcolors	This function returns colors of the final tokens; final tokens are the tokens that are left in places when the simulation was stopped or completed. In addition to the first input argument (which is the simulation results), the optional second input argument limits the places we are interested. E.g.: **`prnfinalcolors(sim, {'p2', 'pNUM1'});`**
prncolormap	This function returns colors of all of the tokens (final tokens as well as previous ones) that were in different places during the simulations; the optional second input argument limits the places we are interested. E.g.: **`prncolormap(sim, {'p2', 'pNUM1'});`**

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

- The input arguments:
  - The first input argument is the identity of the place (place) from which we are going to select the tokens.
  - The second input argument is the number of tokens wanted (nr_tokens_wanted) with the specified color, and finally,
  - The specific colors are defined as the third input argument (t_color)
- The output arguments:

- The first output argument is the set of tokIDs (set_of_tokID) satisfying the color specifications. The length of the set is equal to the input argument of the nr_tokens_wanted.
- The second output argument (nr_token_av) is the number of valid tokIDs available in the set set_of_tokID; the set may have trailing zeros just to make its length equal to nr_tokens_wanted.

The using the functions:

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

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

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

- '**tokenAny**': this function just returns a set of tokens regardless of their colors (note: only 2 input arguments). Usage:
  [set_of_tokID, nr_token_av] = tokenAny(place, nr_tokens_wanted)

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

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

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

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

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

1.3 Color Inheritance

In GPenSIM, colored tokens can be utilized only by the transitions. Since transitions are the active elements, pre-processor files (specific and COMMON pre-processor) can be programmed with code for manipulating token colors:

1. **Token selection based on colors**: When a transition is enabled, it can choose (consume) input tokens with specified colors.
2. **Color Inheritance**: When a transition fires, it collects the colors of all of the input tokens. When the output tokens are deposited in the output places, the colors inherited from the input tokens are transferred to the output tokens. NOTE: inheritance of colors can be prohibited by 'overriding.'
3. **Adding new colors**: When the output tokens are deposited in the output places, new colors (in addition to the inherited colors) can also be added to the output tokens. This new color will be added on top of the inherited colors unless inheritance is overridden. In the case of overriding, the deposited tokens in the output places will only have the new colors added by the transition.

Let us start experimenting with coloring with the help of two simple examples, candidly called 'resolving conflict with coloring,' and 'simple adder.'

1.3.1 Example-39: Resolving conflict with coloring

Figure-1-1 shows that two transitions **t1** and **t2** are in conflict as they try to grab the same token from the input place **pS**. The cold start transition **tS** deposits token into **pS** at a slower rate (firing time = 10 TU) and this token is being sought by the two transitions **t1** and **t2**. To avoid conflict, let us say that **t1** is allowed to fire 90% of the time, and **t2** for the rest 10%. To realize this, **tS** will add color 't1' to the output token 90% of the time, and the color 't2' for the rest of the time. This means **t1** and **t2** can only take token that bears the respective color.

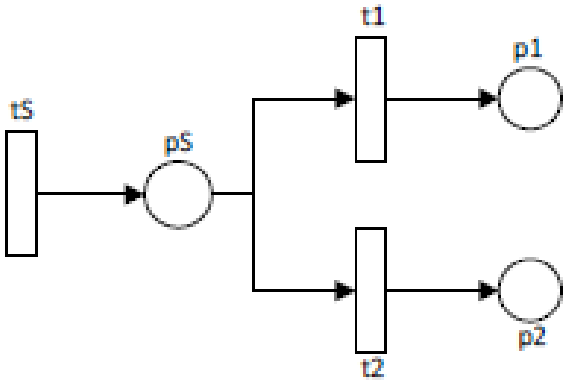

Figure 1-1: t1 and t2 are in conflict

PDF:

```
% Example-39: Resolving Conflict with color
function [png] = conflict_pdf()
png.PN_name = 'Resolving Conflict with color';
png.set_of_Ps = {'pS', 'p1', 'p2'};
png.set_of_Ts = {'tS','t1','t2'};
png.set_of_As = {'tS','pS',1, ...                % tS
        'pS','t1',1, 't1','p1',1, ... % t1
        'pS','t2',1, 't2','p2',1}; % t2
```

MSF:

```
% Example-39: Resolving conflict with color
% t1 and t2 are in conflict. t1 has 90% chance, whereas t2 has 10%
clear all; clc;
global global_info
global_info.STOP_AT = 1000; % stp after 1000 TU
png = pnstruct('conflict_pdf');

dyn.m0 = {'pS', 1}; % pS has one token initially
dyn.ft = {'tS', 10, 'allothers', 1};  % firing times of t1 & t2 is 1 TU
pni = initialdynamics(png, dyn);
sim = gpensim(pni);
plotp(sim, {'p1', 'p2'});
prnstate();
```

COMMON_PRE: In COMMON_PRE, **tS** adds color ‘t1’ to the output token 90% of the time (color ‘t2’ for the 10% of the time). **t1** selects token with color ‘t1’ only, and similarly, **t2** selects token with color ‘t2’ only.

```
function [fire, transition] = COMMON_PRE (transition)
tname = transition.name;
switch tname
    case 'tS'
        random_num = rand; % 0 to 1
        %%%%   t1 has 90% chance, whereas t2 has only 10%
        if and(ge(random_num,0), lt(random_num,0.9))
            color = 't1'; % t1 can fire
        else
            color = 't2'; % t2 can fire
        end
        transition.new_color = color;
        fire = 1; % always fire tE as it is not in conflict
```

```
    case 't1'
        % From pS, t1 takes only the token with color 't1'
       tokID = tokenAnyColor('pS',1, {'t1'});%select token with color 't1'
        fire = tokID; % fire only if the token has color 't1'

    case 't2'
        % From pS, t2 takes only the token with color 't2'
        tokID = tokenAnyColor('pS',1,{'t2'});% select token with color 't2'
        fire = tokID; % fire only if the token has color 't2'

    otherwise
        disp('Unknown method.')
end
```

The simulation results (print the final states with '**prnstate**') show that **t1** has fired about 90% of the time (as **p1** has 90% of the tokens).

```
90p1 + 9p2 + pS
```

1.3.2 Example-40: A Simple Adder

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

The Petri Net model of a simple adder shown in the figure-1-2. The Petri Net has six places and four transitions. Places **p1** and **p2** are just to keep the initial tokens so that the transitions **tGET_NUM1** and **tGET_NUM2** can fire only once. Transitions **tGET_NUM1** and **tGET_NUM2** get an input number each from the user; let say the numbers fed by the user are 21 and 45. Then these two transitions convert the numbers into text strings (**'**21**'** and **'**45**'**) and then add the strings as colors to the output tokens deposited into **pNUM1** and **pNUM2** respectively. Thus, the places **pNUM1** and **pNUM2** have tokens with input numbers as the colors.

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

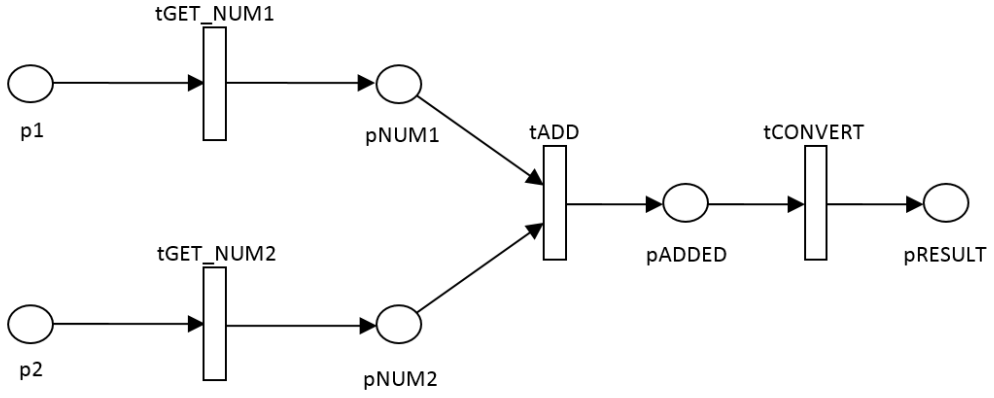

Figure 1-2: A Simple Adder.

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

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

MSF: 'simple_adder.m'

```
% Example-40: SIMPLE ADDER with colored tokens
pns = pnstruct('simple_adder_pdf');
dyn.m0 = {'p1',1, 'p2',1};  % initial tokens
dyn.ft = {'allothers',5};   % firing times of all transitions are 5 TU
pni = initialdynamics(pns, dyn);

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

prncolormap(sim); % print colors of all the tokens – present and past
prnfinalcolors(sim); % print only colors of the tokens that are present
```

PDF: 'simple_adder_pdf.m'

```
% Example-40: SIMPLE ADDER with colored tokens
function [pns] = simple_adder_pdf()
pns.PN_name = 'Color example: Simple Adder';
pns.set_of_Ps = {'p1', 'p2', 'pNUM1', 'pNUM2', 'pADDED','pRESULT'};
pns.set_of_Ts = {'tGET_NUM1','tGET_NUM2','tADD','tCONVERT'};
pns.set_of_As = {'p1','tGET_NUM1',1, 'tGET_NUM1','pNUM1',1,...
             'p2','tGET_NUM2',1, 'tGET_NUM2','pNUM2',1,...
             'pNUM1','tADD',1, 'pNUM2','tADD',1,...
             'tADD','pADDED',1, 'pADDED','tCONVERT',1, ...
             'tCONVERT','pRESULT',1};
```

Pre-processor: 'tGET_NUM1_pre.m'

The specific pre-processor will ask the user to input a number:

```
function [fire, transition] = tGET_NUM1_pre (transition)

num1 = input('input number-1: ');  % ask for a number from the user
transition.new_color = num2str(num1);% convert number to text string(color)
fire = 1;  % let it fire
```

Pre-processor: 'tGET_NUM2_pre.m'

This pre-processor will ask the user to input another number; otherwise same as **tGET_NUM1_pre**

Pre-processor: 'tADD_pre.m'

There is no need for tADD_pre. By default, it will inherit colors from input tokens and put the colors to the output token.

Pre-processor: 'tCONVERT_pre.m'

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

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

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

transition.new_color = num2str(num1+num2); % add the sum as new color
transition.override = 1; % only sum as color - NO inheritance
fire = 1;  % let it fire
```

Simulation Results:

The statement **`prncolormap(sim)`** prints the colors of all the tokens (tokens that are present as well as those existed before). As shown in the screen dump below,

- **p1** had tokens with no colors,
- **p2** had tokens with no colors,
- **pNUM1** had a token with color '21',
- **pNUM2** had a token with color '45',
- **pADDED** had tokens with colors '21' and '45', and
- **pRESULT** has a token with color '66'.

```
**** Enter two DIFFERENT Numbers ****
input number-2: 21
input number-1: 45

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

Color Map for place: p1
(no colors)

Color Map for place: p2
(no colors)

Color Map for place: pADDED
Time: 10     Colors:   "21"  "45"

Color Map for place: pNUM1
Time: 5     Colors:   "45"
Time: 5     Colors:   "45"

Color Map for place: pNUM2
Time: 5     Colors:   "21"

Color Map for place: pRESULT
Time: 15     Colors:   "66"
```

The statement **`prnfinalcolors(sim)`** prints only the colors of the final tokens (tokens that are present in various places, at the end of the simulation). As shown in the screen dump below, only **pRESULT** had one token at the end of the simulation.

```
**** **** Colors of Final Tokens ...
No. of final tokens: 1

Place: pRESULT
Time: 15    Colors:    "66"
```

1.4 Color Pollution

In GPenSIM, color inheritance means passing the colors of the input tokens to the output tokens. Careless use of color inheritance can cause some troubles. The following example explains why.

1.4.1 Example-41: Color Pollution

In the figure-1-3 shown below, **tColor** is supposed to deposit three tokens into **pEnd**. Each token must have a different color: the first token with "RED," the second token with "GREEN," and the final one with "BLUE."

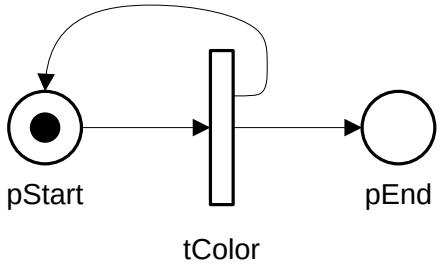

Figure 1-3: Color Pollution.

MSF and the pre-processor for **tColor** are given below.

MSF:

```
% Example-41: Color Pollution Example
global global_info
global_info.THREE_COLORS = {'RED', 'GREEN', 'BLUE'};
png = pnstruct('color_poll_pdf');

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

prnfinalcolors(sim); %%%% PRINT RESULTS %%%%%
```

Pre-processor for tColor ('tColor_pre.m'):

```
% Example-41: Color Pollution Example
function [fire, transition] = tColor_pre (transition)

global global_info
tColorFired = times_fired('tColor'); % how many times tColor has fired

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

% set a new color to next token
new_color = global_info.THREE_COLORS{tColorFired+1};
transition.new_color = new_color;
```

```
fire = 1;
```

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

```
**** **** Colors of Final Tokens ...
No. of final tokens: 4

Place: pEnd
Time: 10    Colors:    "RED"
Time: 20    Colors:    "GREEN"    "RED"
Time: 30    Colors:    "BLUE"    "GREEN"    "RED"

Place: pStart
Time: 30    Colors:    "BLUE"    "GREEN"    "RED"
```

The initial token in **pStart** is colorless. Each time the transition **tColor** fires, it is supposed to assign only one color to the token from **pStart**.

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

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

```
% Example-41: (version-2: overriding color inheritance)
function [fire, transition] = tColor_pre (transition)
global global_info

…
…

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

Simulation results after overriding color inheritance:

```
**** **** Colors of Final Tokens ...
No. of final tokens: 4

Place: pEnd
Time: 10    Colors:    "RED"
Time: 20    Colors:    "GREEN"
Time: 30    Colors:    "BLUE"

Place: pStart
Time: 30    Colors:    "BLUE"
```

1.5 Token Selection based on Color

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

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

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

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

1.5.1 Example-42: Selecting Input Tokens with Specific Colors

Figur-2-4 given below depicts a production process. Transition **tGEN** represents a machine that rotationally produces four types of products, 'A,' 'B,' 'C,' and 'X.' Though the buffer **pBUFF** contains all four types of products, **tA** is supposed to select 'A' products only. Similarly, **tB** selects 'B' products and **tC** selects 'C' products only.

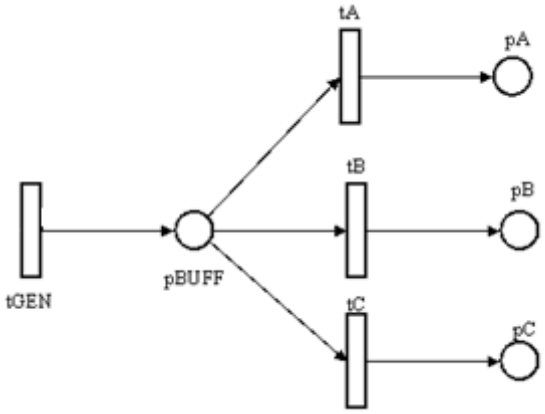

Figure 1-4: Selecting input tokens based on color

MSF:

```
% Example-42: Color Selection
global global_info
global_info.STOP_AT = 8.8; % limit the simulation time to 8.8 TUs

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

pns = pnstruct('select_color_pdf');

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

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

PDF:

```
% Example-42: Color Selection
function [pns] = select_color_pdf()
pns.PN_name = 'EX-28-1: SELECT COLOR Example';
pns.set_of_Ps = {'pBUFF', 'pA', 'pB', 'pC'};
pns.set_of_Ts = {'tGEN', 'tA', 'tB', 'tC'};
pns.set_of_As = {'tGEN','pBUFF',1, 'pBUFF','tA',1,'tA','pA',1,...
    'pBUFF','tB',1,'tB','pB',1, 'pBUFF','tC',1,'tC','pC',1};
```

COMMON_PRE:

tGEN is to produce tokens with a color: 'A,' 'B,' 'C,' or 'X.'; whereas, the other transitions **tA**, **tB**, and **tC** should select tokens with color 'A,' 'B,' or 'C,' respectively.

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

% tGEN generates tokens with color 'A', 'B', 'C'
if strcmp(transition.name, 'tGEN'),
    index = mod(global_info.cr_index, 4)+1;   % increase the CR index
    global_info.cr_index = global_info.cr_index + 1; % update the CR index
    transition.new_color = global_info.cr(index); % get the next color
    fire = 1;
    return
end

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

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

Simulation results

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

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

Place: pA
Time: 1.125    Colors:    "A"
Time: 5.175    Colors:    "A"

Place: pB
Time: 2.15    Colors:    "B"
Time: 6.175    Colors:    "B"

Place: pBUFF
```

```
Time: 4.075    Colors:    "X"
Time: 8.075    Colors:    "X"

Place: pC
Time: 3.175    Colors:    "C"
Time: 7.175    Colors:    "C"
```

1.5.2 Example-43: Required or Preferred Color?

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

In the previous example, we forced the transition **tA** to select a token with color 'A.' Function **`tokenAnyColor`** will return a tokID if a token is with 'A' color is available or else the value of the returned tokID will be zeros ('[0]'). And then we forced the transition to fire only if tokID is not zero, meaning there must be at least one token with the required color so that the transition can fire.

However, we may also allow a transition to ***prefer*** 'A' tokens. This means, if 'A' tokens are available, they will be consumed. If not, one of the other existing tokens of 'B,' 'C,' or 'X' will be consumed. In the newer pre-processor given below **tA** prefers (rather than insists) 'A' tokens, whereas **tB** and **tC** are very selective as before:

```
% Example-43: tA prefers color 'A', but others colors are acceptable
function [fire, transition] = COMMON_PRE (transition)

global global_info

% tGEN generates tokens with color 'A', 'B', 'C' or 'X'
if strcmp(transition.name, 'tGEN'),
    index = mod(global_info.cr_index, 4)+1;
    global_info.cr_index = global_info.cr_index + 1;
    transition.new_color = global_info.cr(index);
    fire = 1;
    return
end

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

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

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

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

Simulations show that now **pA** has tokens with many colors ('A,' 'B,' 'C,' and 'X'), whereas **pB** and **pC** have only specific colors:

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

Place: pA
Time: 1.125    Colors:    "A"
Time: 3.175    Colors:    "C"
Time: 4.175    Colors:    "X"
Time: 5.175    Colors:    "A"
Time: 6.175    Colors:    "B"
Time: 8.175    Colors:    "X"

Place: pB
Time: 2.15    Colors:    "B"

Place: pC
Time: 7.175    Colors:    "C"
```

1.5.3 Example-44: Using the color selection functions "tokenWO…"

In this example, we study the usage of the functions **tokenWOAnyColor**, **tokenWOAllColor**, and **tokenWOEXColor**. Figure-1-5 shows the Petri Net which stepwise filters tokens based on their colors. **tColor** produces tokens with color that consists of different combinations of 'A,' 'B,' 'C,' 'X,' 'Y,' and 'Z.' The Petri Net should filter these tokens to capture the tokens that possess either the three colors 'A,' 'B,' and 'C,' or the other three colors 'X,' 'Y,' and 'Z.'

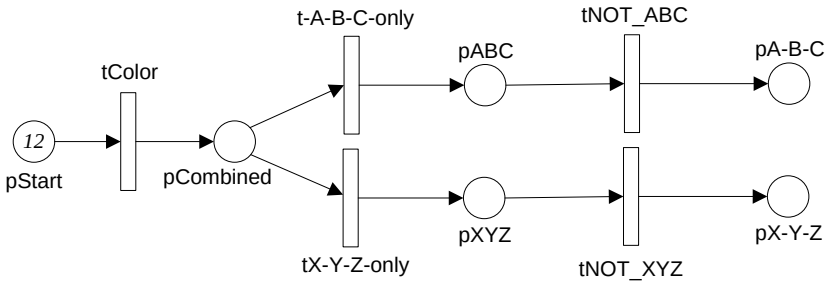

Figure 1-5: Using "tokenWO…" functions as filters

In the Petri Net:

- **tColor** produces tokens with any combination of A-B-C-X-Y-Z and dumps into **pCombined**.

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

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

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

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

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

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

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

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

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

The COMMON_PRE is given below:

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

% tColor generates tokens with many combination of colors
if strcmpi(transition.name, 'tColor'),
    global_info.counter = global_info.counter + 1;
    switch global_info.counter
        case {1}
            transition.new_color = {'A','B','C'};
        case {2}
            transition.new_color = {'X','Y','Z'};
        case {3}
            transition.new_color = {'A','B','C','X','Y'};
        case {4}
            transition.new_color = {'A','B','C','X'};
        case {5}
            transition.new_color = {'X','Y','Z','A'};
        case {6}
            transition.new_color = {'X','Y','Z','C','A'};
        case {7}
```

```
            transition.new_color = {'X'};
        case {8}
            transition.new_color = {'X','Y'};
        case {9}
            transition.new_color = {'A','X','Y'};
        case {10}
            transition.new_color = {'A','B','Y'};
        case {11}
            transition.new_color = {'A','B'};
        case {12}
            transition.new_color = {'C'};
        otherwise
            fire = 0; return
    end;
    fire = 1; return
end

if strcmpi(transition.name, 'tA-B-C-only'),
    tokID1 = tokenWOAnyColor('pCombined',1, {'X','Y','Z'});

elseif strcmpi(transition.name, 'tX-Y-Z-only'),
    tokID1 = tokenWOAnyColor('pCombined',1, {'A','B','C'});

elseif strcmpi(transition.name, 'tNOT_ABC'),
    tokID1 = tokenWOEXColor('pABC',1, {'A','B','C'});

elseif strcmpi(transition.name, 'tNOT_XYZ'),
    tokID1 = tokenWOEXColor('pXYZ',1, {'X','Y','Z'});
end
transition.selected_tokens = tokID1;
fire = tokID1;
```

1.6 Wrapping Up: Token Selection based on Color

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

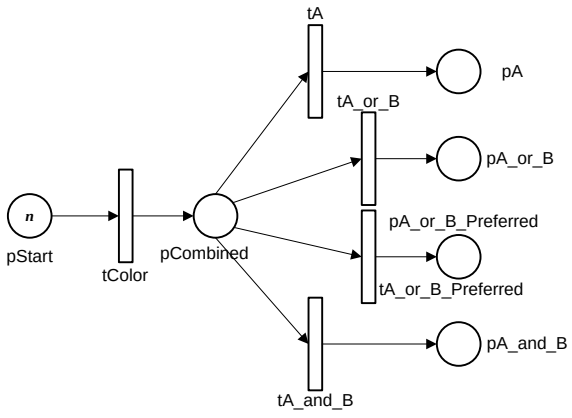

Figure 1-6: Simple Petri Net for color demo

- Let us say that **tA** selects a token with color 'A' from **pCombined** (meaning tokens with color {'A'} or {'A,' 'B'} or {'A,' 'X'} are relevant). The program code in the pre-processor:

```
selected_tokens = tokenAnyColor('pCombined', 1, {'A'})
fire = (selected_tokens);  % must
```

- Let us say that **tA_or_B** selects ‘A’ **or** ‘B’ from **pCombined**. The program code in the pre-processor:

```
tokID1 = tokenAnyColor('pCombined', 1, {'A','B'});
selected_tokens = tokID1;    % tokens to be removed
fire = tokID1; % must
```

- **tA_or_B_Preferred prefers** ‘A’ or ‘B’ from **pCombined**. The program code in the pre-processor:

```
tokID1 = tokenAnyColor('pCombined', 1, {'A','B'});
selected_tokens = tokID1; % preferred tokens to be consumed
fire = 1; % fire anyway
```

- **tA_and_B** selects a token with ‘A’ **and** ‘B’ from **pCombined**. The program code in the pre-processor:

```
tokID1 = tokenAllColor('pCombined', 1, {'A','B'});
selected_tokens = tokID1;    % tokens to be removed
fire = tokID1; % must
```

1.7 Token Selection from Multiple Input Places

The previous examples were about transitions consuming input tokens from a single input place. The selection of tokens from *multiple input places* is just a simple extension of the technique presented in the previous sections.

This section explains how we can let a transition consume tokens (based on colors) from different input places.

1.7.1 Example-45: Token selection from multiple input places

Let us say that **pAB** has tokens with colors {‘’, ‘A,’ ‘B,’ {‘A,’ ‘B’}} and **pXY** has tokens with colors {‘’, ‘X,’ ‘Y,’ {‘X,’ ‘Y’}} (‘’ means colorless).

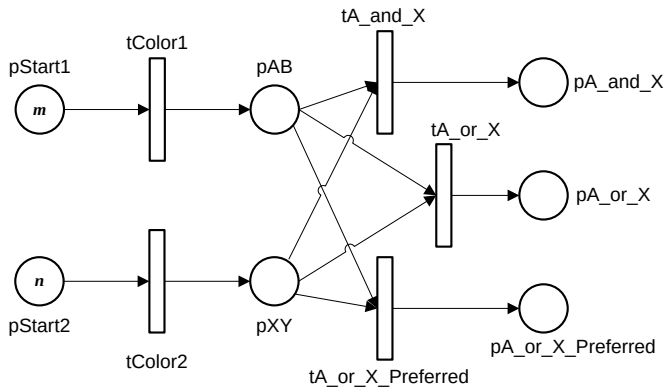

Figure 1-7: PN with two input places for the color demo.

- Let us say that **tA_and_X** selects a token with color ‘A’ from **pAB** **and** a token with ‘X’ from **pXY**. The program code in the pre-processor:

```
tokID1 = tokenAnyColor('pAB', 1, {'A'});
tokID2 = tokenAnyColor('pXY', 1, {'X'});
selected_tokens = [tokID1 tokID2];    % tokens to be removed
fire = all(selected_tokens); % must have both valid tokens
```

- **tA_or_X** selects a token with color ‘A’ from **pAB** **or** a token with color ‘X’ from **pXY** (at least one token be ‘A’ or ‘X’). The program code in the pre-processor:

```
tokID1 = tokenAnyColor ('pAB', 1, {'A'});
tokID2 = tokenAnyColor ('pXY', 1, {'X'});
selected_tokens = [tokID1 tokID2];    % tokens to be removed
fire = any(selected_tokens); % must be at least one valid token
```

- **tA_or_X_Preferred** **prefers** a token with color ‘A’ from **pAB** or a token with color ‘X’ from **pXY**. The program code in the pre-processor:

```
tokID1 = tokenAnyColor('pAB', 1, {'A'});
tokID2 = tokenAnyColor('pXY', 1, {'X'});
selected_tokens = [tokID1 tokID2];    % tokens to be removed
fire = 1; % fire even if the preferred colors not exist
```

1.8 Token Selection based on Time

So far, we looked into the functionality for selecting tokens with colors, where color refer to the text strings tagged on the tokens. We can also let a transition select tokens from input places based on the time; time here refers to which token is the oldest (the ones that were deposited earliest) in the place, which is the youngest, and which was deposited between a given time interval. Table-2-2 given below shows the select functions that deal with the deposited time of a token in a place.

Table-2-2: GPenSIM functions for selection of tokens based on time.

Function	*Description*
`tokenArrivedBetween`	Select tokens that were deposited in a place within the given time interval.
`tokenArrivedEarly`	Select tokens that were deposited earliest to a place.
`tokenArrivedLate`	Select tokens that were deposited latest to a place.

A transition may select input tokens based on the time these tokens were deposited into the places. Selection can be made by three ways:

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

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

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

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

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

```
function [fire, transition] = tLR_A_pre (transition)
[selected_tokens, nr_tokens_av] = tokenArrivedEarly('pBUFF',4);
fire = 1;
```

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

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

```
function [fire, transition] = tLR_A_pre (transition)
[selected_tokens, nr_tokens_av] = tokenArrivedLate('pBUFF',4);
fire = 1;
```

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

1.8.1 Example-46: Token selection based on time

Figure-1-8 shows the Petri Net for experimenting token selection based on time. **pStart** has 100 initial tokens (initial tokens are, of course, colorless). **tStart** moves one token at a time from **pStart** to **pQueue**. The branch "**pDelay – tDelay – pReady**" is a delay, just to keep the set of selection transitions (**tEarly**, **tIntv_E**, **tIntv_L**, and **tLate**) wait until all the 100 tokens from **pStart** are deposited into **pQueue**.

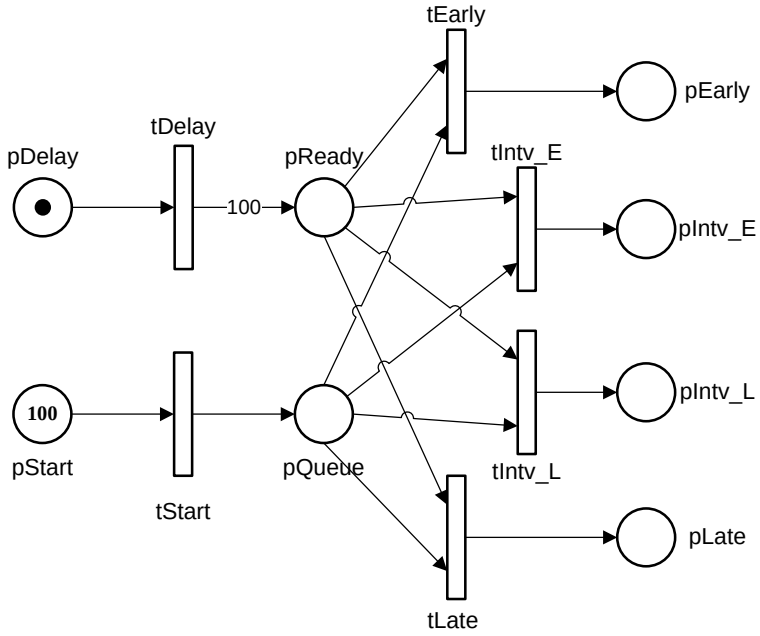

Figure 1-8: Token selection based on time.

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

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

Also, when the four transitions (**tEarly**, **tIntv_E**, **tIntv_L**, and **tLate**) pick their input tokens from **pQueue**, they also add color to the token followingly:

- Get the *creation time* of the token (the time the token was deposited in **pQueue** by **tStart**).
- Convert the creation time into a text string.
- Add the text string as color.

Thus, every token that is deposited into the respective output place (e.g., **pEarly** by **tEarly**) will have a color that represents the time the token was deposited into **pQueue**.

PDF:

```
% Example-46: Token selection based on time
function [png] = select_time_pdf()
png.PN_name = 'Token selection based on time';
png.set_of_Ps = {'pStart', 'pQueue', 'pDelay', 'pReady',...
```

```
                  'pEarly', 'pIntv_E', 'pIntv_L', 'pLate'};
png.set_of_Ts = {'tStart', 'tDelay', ...
                  'tEarly', 'tIntv_E', 'tIntv_L', 'tLate'};
png.set_of_As = {'pStart','tStart',1, 'tStart','pQueue',1,...  % tStart
    'pDelay','tDelay',1, 'tDelay','pReady',100,...      % tDelay 100
    'pQueue','tEarly',1, 'pReady', 'tEarly',1, ...      % tEarly
    'tEarly','pEarly',1, ...                            % tEarly
    'pQueue','tIntv_E',1, 'pReady', 'tIntv_E',1, ...    % tIntv_E
    'tIntv_E','pIntv_E',1, ...                          % tIntv_E
    'pQueue','tIntv_L',1, 'pReady', 'tIntv_L',1, ...    % tIntv_L
    'tIntv_L','pIntv_L',1, ...                          % tIntv_L
    'pQueue','tLate',1, 'pReady', 'tLate',1, ...        % tLate
    'tLate','pLate',1, ...                              % tLate
                };
```

MSF:

```
% Example-46: Token selection based on time
clear all; clc;
global global_info
global_info.STOP_AT = 300;

png = pnstruct('select_time_pdf');
dyn.m0 = {'pStart',100, 'pDelay',1};
dyn.ft={'tDelay',200,'allothers',1}; %ft of tDelay: any number > 100 is OK
pni = initialdynamics(png,dyn);
sim = gpensim(pni);
prnfinalcolors(sim, {'pEarly', 'pIntv_E', 'pIntv_L', 'pLate'});
```

COMMON_PRE:

```
% Example-46: Token selection based on time
function [fire, transition] = COMMON_PRE (transition)

tname = transition.name;

% if the enabled trans is tStart or tDelay, just let it fire:
if ismember(tname, {'tStart', 'tDelay'}),
    fire = 1; return
end

% the following transitions select only three tokens based on time
if ismember(tname, {'tEarly', 'tIntv_E', 'tIntv_L', 'tLate'}),
    tx = timesfired(tname); % how many times trans has already fired
    % trans can fire only three times
    if eq(tx, 3), fire = 0; return, end

    switch tname
        case 'tEarly'  % picks Earliest token
            tokID1 = tokenArrivedEarly('pQueue', 1);
        case 'tIntv_E' % picks token from early time period
            tokID1 = tokenArrivedBetween('pQueue', 1, 10, 20);
        case 'tIntv_L' % picks token from late time period
            tokID1 = tokenArrivedBetween('pQueue', 1, 80, 90);
        case 'tLate'   % picks latest token
            tokID1 = tokenArrivedLate('pQueue', 1);
        otherwise
            error('Not possible.')
    end % switch tname
```

```
    transition.selected_tokens = tokID1;
    token_creation_time = get_tokCT('pQueue', tokID1); % get creation time
    transition.new_color=num2str(token_creation_time); % add as color
    fire = tokID1;
end
```

Simulation results:

We break the results into three parts: The first part, given below, clearly indicates that **tEarly** selected the earliest arrived tokens from **pQueue**, as the colors attached to the tokens are '1'-'3', meaning these tokens were deposited into **pQueue** by **tColor** at the time 1-3.

```
Place: pEarly
Time: 201    Colors:    "1"
Time: 202    Colors:    "2"
Time: 203    Colors:    "3"
```

The second part is for **tIntv_E** which selects the tokens that were deposited into **pQueue** within the time interval [10 20].

```
Place: pIntv_E
Time: 201    Colors:    "10"
Time: 202    Colors:    "11"
Time: 203    Colors:    "12"
```

The third part is for **tIntv_L,** which selects the tokens that were deposited into **pQueue** within the time interval [80 90]:

```
Place: pIntv_L
Time: 201    Colors:    "80"
Time: 202    Colors:    "81"
Time: 203    Colors:    "82"
```

Finally, the fourth part, given below, shows that **tLate** selected the latest arrived tokens from **pQueue**, as the colors attached to the tokens are '98'-'100'.

```
Place: pLate
Time: 201    Colors:    "100"
Time: 202    Colors:    "99"
Time: 203    Colors:    "98"
```

2. Simple Applications with Colored Petri Nets

In this chapter, we solve three simple problems with Colored Petri Nets, using the coloring facilities in GPenSIM. These problems are classical problems that are often discussed in the computer science literature. The problems are: 1) Buffered Producers-Consumers with Shared Channel, 2) Cigarette Smokers' problem, and 3) Generation of Fibonacci number series.

GPenSIM allows only one type of 'color set,' which is the set of ASCII text strings. In this chapter, we shall see that this rudimentary facility for coloring tokens is usually sufficient to solve simple problems. However, for solving complex and large-scale industrial problems, we need to supplement this simple coloring mechanism with the enabling functions and global variables.

2.1 Producer-Consumer Problem

The "Buffered Producers-Consumers with Shared Channel (BPCwSC)" is a classical problem. The GPenSIM based solutions for the BPCwSC problem given in this section is based on the paper by Davidrajuh (2015).

The BPCwSC problem was put forward in the seventies as a mechanism to test conflict solutions in operating systems (Kosaraju 1973, Agerwala et al. 1973).

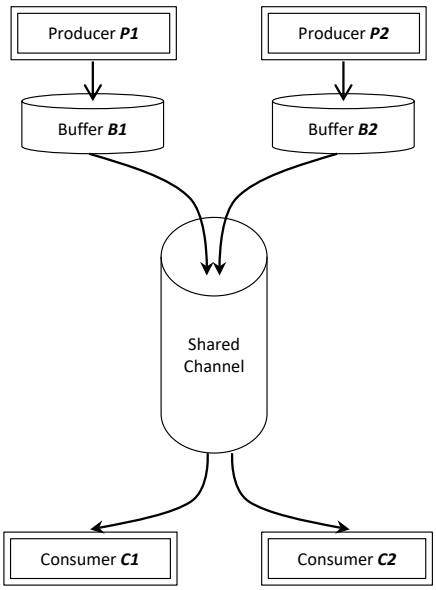

Figure 2-1: The Buffered Producers-Consumers with Shared Channel (BPCwSC) problem (Peterson, 1981).

Here are the details of the BPCwSC problem (figure-2-1):

- Four processes: there are four processes involved: two of them are producers, and the other two are consumers.
- Two Producer-Consumer pairs: The first producer (P1) produces items only for the first consumer (C1). Similarly, the second producer (P2) produces items only for the second consumer (C2).

- The buffers for products: Items which are produced by the producers are kept in the corresponding buffers (B1 and B2) until transported away to the consumers through the shared channel.
- The shared channel: the shared channel has a unit bandwidth, meaning only one item can be transported through it at a time.
- Pull production environment: The producers are passive as they merely produce products and put them into the respective buffers. The consumers are active, as they actively control the use of the channel to pull appropriate products.
- Differential treatment: the pair P1-C1 has priority over the P2-C2 pair; this means, transportation of items from B1 to C1 has priority over items from B2 to C2. This also means, as long as the B1 is nonempty, the transportation channel only transports items from B1 to C1.

In summary, the solutions for the BPCwSC problem must satisfy the following two requirements:

- Requirement-1: Only one product at a time can flow through the shared channel, and
- Requirement-2: P1-C1 pair has priority over the P2-C2 pair. This means, as long as there are products from P1 (to C1), the shared channel will be transporting only these P1-C1 products; products from P2 must wait.

The BPCwSC problem is solved many times, using a variety of Petri Net extensions (Peterson 1981, Kosaraju 1973, Agerwala and Flynn 1973, Keller 1974). In this section, we shall provide yet another solution using the coloring facilities in GPenSIM. It is also possible to use the other Petri Net extensions that are implemented in the GPenSIM (such as inhibitor arcs, priorities) to solve this problem.

2.1.1 P/T Petri Net model of BPCwSC problem

Figure-2-2 shows a P/T Petri Net model for the BPCwSC problem. The model satisfies the first requirement (“only one item at a time through the shared channel”) by utilizing a semaphore mechanism consisting of the place **pCh** and the transitions **tCh1** and **tCh2**. The transitions **tCh1** and **tCh2** that transport through the channel can be enabled at the same time; however, they can never fire simultaneously, as there is only one enabling token (‘semaphore’) in common in the place **pCh**.

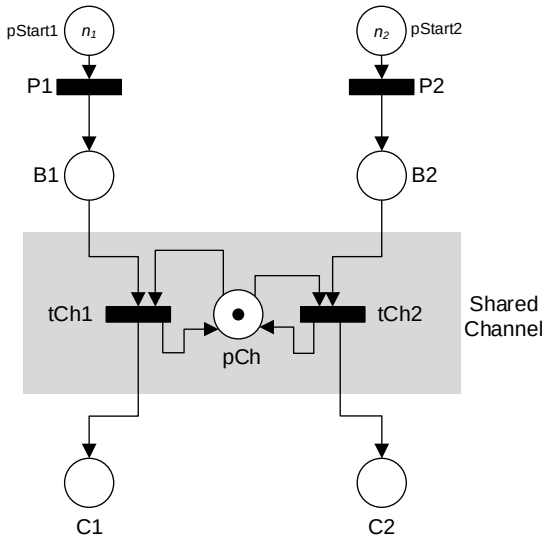

Figure 2-2: P/T Petri Net model of the BPCwSC problem.

Though the first requirement is satisfied, the model will not satisfy the second requirement ("P1-C1 pair has priority over P2-C2 pair"). This is because, if both **tCh1** and **tCh2** are enabled at the same time, the model does not prioritize **tCh1** over **tCh2**; when **tCh1** and **tCh2** are enabled at the same time, it is indeterminate whether **tCh1** will fire blocking **tCh2** from firing. As there is no mechanism in P/T Petri Net to give priority to **tCh1**, P/T Petri Net cannot be used solve the BPCwSC problem.

2.1.2 Example-47: Modeling BPCwSC problem with Colored Petri Net

The Colored Petri Net model of the BPCwSC problem is shown in figure-2-3.

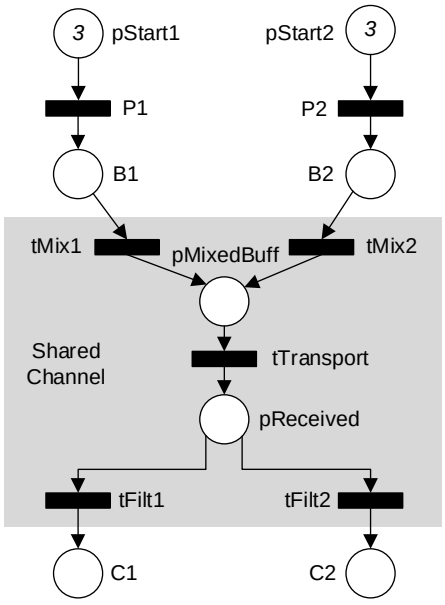

Figure 2-3: Colored Petri Net model.

To perform color manipulation, the model possesses some more transitions and places compared with the P/T Petri Net model. The color extension is mainly to impose the following actions and enabling conditions on transitions:

- When the transition **tMix1** fires, it adds a color "P1" on the tokens from **B1** and deposits them into the common buffer **pMixedBuff**. Similarly, transition **tMix2** adds color "P2" on the tokens from **B2** that are also deposited into the common buffer **pMixedBuff**.
- Transition **tTransport** is the only transition that transports products (tokens) through the channel. **tTransport** picks a product from **pMixedBuff** and deposits it into the buffer '**pReceived**.' However, when selecting tokens from **pMixedBuff**, **tTransport** *prefers* the token with color "P1".
- The buffer at the end of the channel is '**pReceived**.' **pReceived** may hold items with color "P1" and "P2". Thus, transition **tFilt1** (short for Filter-1) selects only the tokens with color "P1" and deposits them into the consumer **C1**; similarly, **tFilt2** selects only the tokens with color "P2" and deposits into **C2**.

Simulations show that the Colored Petri Net model satisfies both requirements. However, it was only possible with additional structures such as the pre-channel common buffer **pMixedBuff** and the post-channel common buffer **pReceived**, both of them are not defined in the BPCwSC problem.

Given below is the COMMON_PRE file for coding the color manipulations that makes for enabling the different transitions.

```
% Example-47: Solving Buffered Producer-Consumer with Shared Channel
%   Solution based on "Colored Petri Net"
%
function [fire, transition] = COMMON_PRE (transition)

switch transition.name
    case 'P1'  % P1 adds color "P1-to-C1" to tokens
        transition.new_color = 'P1-to-C1';
        fire = 1;

    case 'P2'  % P2 adds color "P2-to-C2" to tokens
        transition.new_color = 'P2-to-C2';
        fire = 1;

    case 'tTransport' % tTransport prefers P1. If no P1, then takes P2
        tokID1 = tokenAnyColor ('pMixedBuff', 1, {'P1-to-C1'});
        transition.selected_tokens = tokID1;    % tokens to be removed
        fire = 1; % ONLY in case of NO P1, take P2

    case 'tFilt1' % tFilt1 selects P2 tokens only from pReceived
        tokID1 = tokenAnyColor ('pReceived', 1, {'P1-to-C1'});
        transition.selected_tokens = tokID1;    % tokens to be removed
        fire = tokID1; % fire ONLY if P1 found

    case 'tFilt2' % tFilt2 selects P2 tokens only from pReceived
        tokID2 = tokenAnyColor ('pReceived', 1, {'P2-to-C2'});
        transition.selected_tokens = tokID2;    % tokens to be removed
        fire = tokID2; % fire ONLY if P2 found

    otherwise
        % just fire, in case of tMix1 and tMix2
        fire = 1;
end
```

The function **`prnfinalcolors`** shows that **C1** was the one that gets the products from **P1** (note the time). When there were no more products made by **P1**, **C2** starts getting products from **P2**.

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

Place: C1
Time: 13    Colors:    "P1-to-C1"
Time: 23    Colors:    "P1-to-C1"
Time: 33    Colors:    "P1-to-C1"

Place: C2
Time: 43    Colors:    "P2-to-C2"
Time: 53    Colors:    "P2-to-C2"
Time: 63    Colors:    "P2-to-C2"
```

2.2 Cigarette Smokers' Problem

In this section, the classical Cigarette Smokers' Problem (CSP) is solved using the coloring facility in GPenSIM. It is also possible to use the other Petri Net extensions that are implemented in the GPenSIM (such as inhibitor arcs, priorities) to solve this problem. The solution given in this section is based on the paper by Davidrajuh (2013). Literature study shows that the CSP is already solved many times, using a variety of Petri Net extensions (Parnas 1975, Kosaraju 1973, Agerwala and Flynn 1973, Keller 1974). Hence, this work providing the yet another solution to the classical CSP serves only as a benchmark to show the strengths and weakness of GPenSIM, in modeling and simulation of discrete-event systems. Suhas Patil originally described the cigarette smokers' problem (CSP) in a technical report at MIT in 1971 (Patil, 1971).

The four agents of the CSP: The CSP involves four agents (or processes); one of the agents is the supplier, and the rest three are smokers:

1. Supplier: supplier possesses an infinite amount of the three basic ingredients for making cigarettes: tobacco, wrapping paper, and matches.
2. Smoker with tobacco (TB): this agent possesses an infinite amount of tobacco and expects the other two ingredients (wrapping paper and matches) from the supplier.
3. Smoker with wrapping paper (WP): this agent possesses an infinite amount of wrapping paper and expects the other two ingredients (tobacco and matches) from the supplier, and
4. Smoker with matches (MA): this agent possesses an infinite amount of matches and expects the other two ingredients (tobacco and wrapping paper) from the supplier.

The procedure for smoking: Let us imagine that the supplier and the three smokers are sitting around a table. The Petri Net for simulating the problem is shown in figure-2-4. The procedure consists of the following steps:

1. The supplier has an infinite supply of all the three ingredients for making cigarettes; the three ingredients are represented by the three tokens in **pSupplier**. However, the supplier arbitrarily selects only two of the ingredients and places them on the table. Thus, **tSelect** deposits only two tokens into the output place **pReady** to represent the two ingredients selected.
2. The smoker who has the remaining ingredient can take the two ingredients, and then make and smoke a cigarette. Since transitions are primitive (zero timed) in P/T Petri Net, two transitions with a buffering place (e.g., **tTBStart**, **pTBSmoking**, and **tTBStop**) are used to indicate the start of smoking, ongoing smoking, and the completion of smoking.
3. Upon completion of smoking, the supplier will be signaled. This is done by the firing of stop transition (e.g., **tTBStop**) which is deposit two tokens back into **pSupplier**.

4. For the next cigarette, the supplier again arbitrarily selects two of the three ingredients and places them on the table. This cycle is repeated forever.

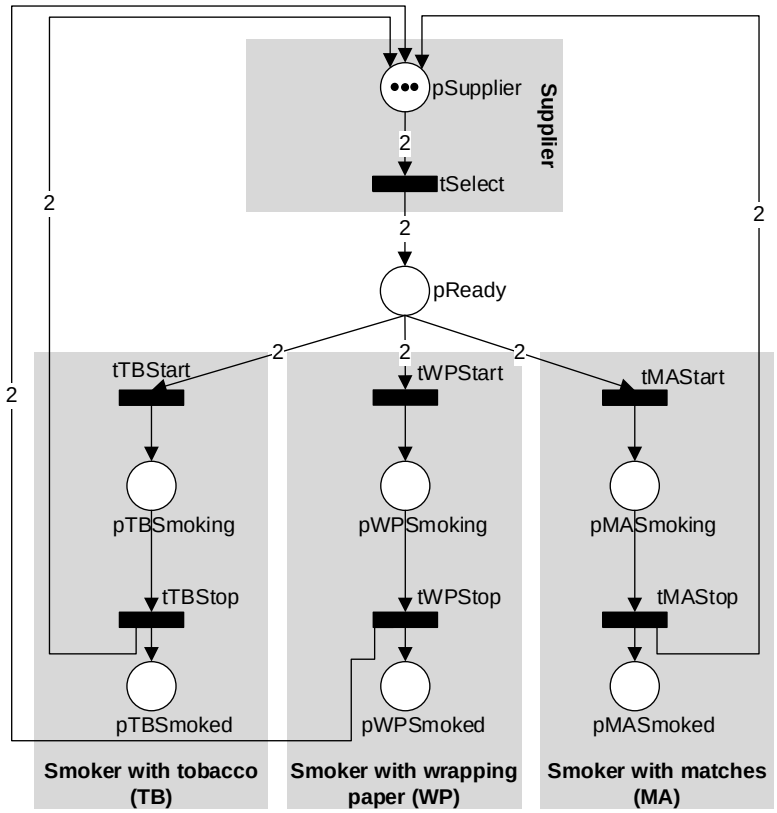

Figure 2-4: P/T Petri Net model for the SCP.

Figure-2-4 shows a P/T Petri Net model for the CSP, which will not work properly; this because, there is no way of preventing a smoker from grabbing the ingredient he already possesses thus preventing the other needy one from taking it. For example, if the supplier puts tobacco and wrapping paper on the table, the smoker with tobacco (or the smoker with wrapping paper) may grab the two items creating a deadlock situation, as he blocks himself from smoking as well as the smoker with matches. As there is no mechanism in P/T Petri Net to prevent the possibility of deadlock, P/T Petri Net cannot model the CSP.

2.2.1 Example-48: Solving the CSP with Color extension

The Colored Petri Net model is shown in the figure-2-5. In comparison with the P/T Petri Net, there are two changes in this model, on the supplier side and the smokers.
The supplier is slightly extended, by including the place **pSelected** and the transition **tTag**. The extension is mainly to impose the following actions and enabling conditions on transitions:

- Every time **tSelect** starts firing, it will randomly choose two ingredients and save them in global_info as `selected_materials`.
- For every firing of **tSelect**, the transition **tTag** will fire two times. Each time, **tTag** will add one of the two selected colors to the output token. Thus, the two tokens deposited in **pReady** would possess the two colors selected by **tSelect**.

The smokers are simplified in figure-2-5. This is because, as we are using the colored Petri Net, the transitions are no longer primitive. Hence, a single transition (e.g., **tTobacco**) can represent a smoker choosing the ingredients, and smoking the cigarette. For example, the smoker with tobacco:

- **tTobacco** can only fire if it finds two tokens in **pReady**, one with color 'Wrapping Paper' and the other with color 'Matches.' Once two tokens with these specified colors are found, **tTobacco** can start firing and will complete after the defined firing time. Similar enabling conditions are imposed on the transitions **tWrapping** (the smoker with wrapping paper) and **tMAStart** (the smoker with matches).

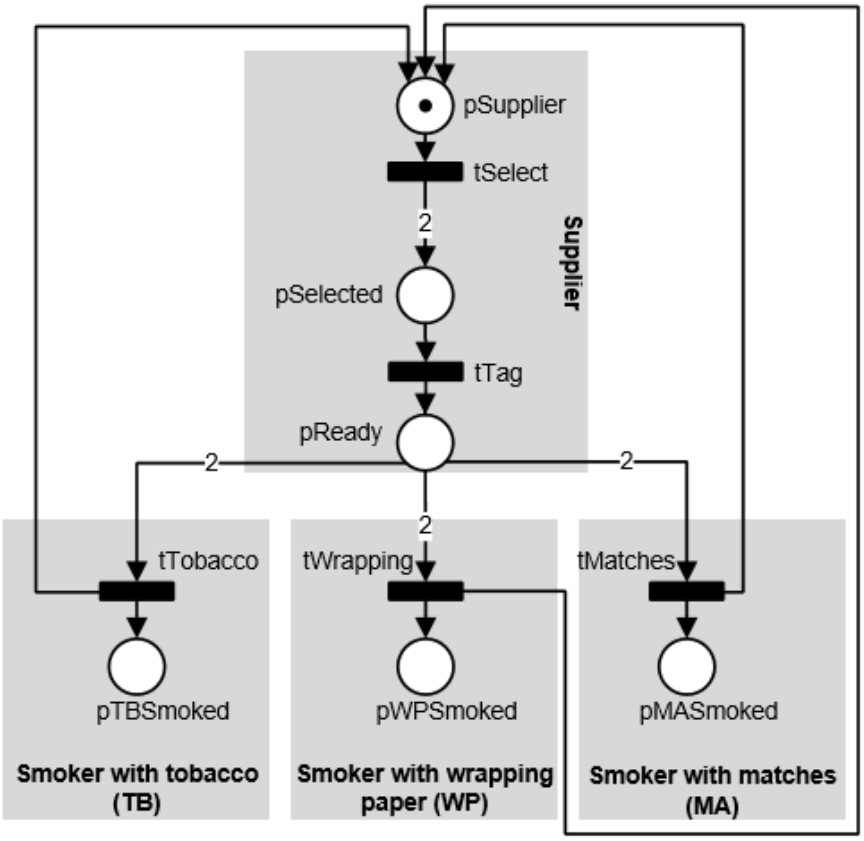

Figure 2-5: The Colored Petri Net for simulating Cigarette smokers' problem

In the program code given below, three smoking rounds are planned.

MSF:

```
% Example-48: Solving Cigerette Smokers' Problem
clear all; clc;

global global_info
global_info.STOP_AT = 50;  %
global_info.SUPPLIES={'Tobacco','Matches','Wrapping-Paper'};%
global_info.MAX_NUMBER_OF_CIGARETTES = 3; % cigarettes to be smoked
global_info.cigarette_counter = 0; % smoked cigarettes so far

pns = pnstruct('csp_col_def');
dyn.m0 = {'pSupplier',1}; % just to start
dyn.ft = {'allothers',1}; % all transitons take 1 TU
pni = initialdynamics(pns, dyn);

sim = gpensim(pni);
prnfinalcolors(sim, {'pMASmoked', 'pTBSmoked', 'pWPSmoked'});
```

Pre-processor for tSelect:
tSelect selects two ingredients for the smoker and save it in the global_info as selected_materials.

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

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% 1. Check whether more cigarettes needed
if eq(global_info.cigarette_counter,
global_info.MAX_NUMBER_OF_CIGARETTES)
    global_info.STOP_SIMULATION = 1;
    fire = 0; return
else
    global_info.cigarette_counter = ...
     global_info.cigarette_counter + 1;
    disp(['***** Cigarette-', ...
        int2str(global_info.cigarette_counter), ':']);
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% 2. Randomly choose two materials

% randomize array [1, 2, 3] using GPenSIM's function 'randomgen'
randomized_1_to_3 = randomgen(1:3);
the_selected = randomized_1_to_3(1:2);
disp('"tSlect" selected the following two: ');
disp(['   ', global_info.SUPPLIES{the_selected(1)}, ' & ', ...
            global_info.SUPPLIES{the_selected(2)}]);
global_info.selected_materials = the_selected;
disp(' ');
fire = 1;
```

Pre-processor for tTag:
For every one firing of **tSelect**, **tTag** fires two times, each time add one of the selected_materials as a color to the output token.

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

% pick one in the selected_materials
color_index = global_info.selected_materials(1);
color = global_info.SUPPLIES{color_index};
disp(['"tTag" tagged the following color: ', color]);

% them remove this pick from selected materials
if eq(numel(global_info.selected_materials), 2),
    global_info.selected_materials = global_info.selected_materials(2);
end

% add the pick as colors
transition.new_color = color;
transition.override = 1; % no color inheritence
fire = 1;
```

COMMON_PRE: This file that shows the conditions imposed upon **tTBStart**, **tWPStart**, and **tMAStart**, so that they select the appropriate tokens from **pReady**.

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

if ismember(transition.name, {'tSelect', 'tTag', ...
        'tTBStop', 'tWPStop', 'tMAStop'})
    % tSelect and tTag have their own pre-processors
    % just let other three to fire (tTBStop, tWPStop, and tMAStop)
    fire = 1; return
end

switch transition.name,
    case 'tTBStart'
        % tTBStart select tokens with color 'Wrapping-Paper' or 'Matches'
        tokID1 = tokenAllColor('pReady',1, {'Wrapping-Paper'});
        tokID2 = tokenAllColor('pReady',1, {'Matches'});
        tokIDs = [tokID1, tokID2];

    case 'tWPStart'
        % tWPStart select tokens with color 'Tobacco' or 'Matches'
        tokID1 = tokenAllColor('pReady',1, {'Tobacco'});
        tokID2 = tokenAllColor('pReady',1, {'Matches'});
        tokIDs = [tokID1, tokID2];

    case 'tMAStart'
        % tMAStart selects tokens with color 'Tobacco' or 'Wrapping-Paper'
        tokID1 = tokenAllColor('pReady',1, {'Tobacco'});
        tokID2 = tokenAllColor('pReady',1, {'Wrapping-Paper'});
        tokIDs = [tokID1, tokID2];
    otherwise
        errormsg([transition.name, 'what is this transition? ']);
end

transition.selected_tokens = tokIDs;
fire = all(tokIDs);
```

Finally, **COMMON_POST** is used to print the name of the smoker (the one out of the three) who was allowed to smoke.

COMMON_POST:

```
function [] = COMMON_POST(trans)

% print who finally smoked!
if ismember(trans.name, {'tTBStart', 'tWPStart', 'tMAStart'}),
    disp(' ');
    disp(['and the one who smoked is: "', trans.name, ...
          '" at time = ', num2str(current_time)]); % the fired trans
    dispMultipleCR(1); % print one empty line
end
```

Simulation results: The first part of the simulation result shows that the cycle time is 4 TU, as it involves 4 firings (one of **tSelect**, two of **tTag**, and one of **tTobacco**/**tWrapping**/**tMatches**).

```
***** Cigarette-1:  *****
"tSlect" selected the following two:
   Wrapping-Paper & Tobacco

"tTag" tagged the following color: Wrapping-Paper
"tTag" tagged the following color: Tobacco
```

```
and the one who smoked is: "tMatches" at time = 4

***** Cigarette-2:  *****
"tSlect" selected the following two:
   Tobacco & Matches

"tTag" tagged the following color: Tobacco
"tTag" tagged the following color: Matches

and the one who smoked is: "tWrapping" at time = 8

***** Cigarette-3:  *****
"tSlect" selected the following two:
   Wrapping-Paper & Tobacco

"tTag" tagged the following color: Wrapping-Paper
"tTag" tagged the following color: Tobacco

and the one who smoked is: "tMatches" at time = 12
```

The second part of the simulation result proves that the smokers correctly took the missing ingredients.

```
**** **** Colors of Final Tokens ...
No. of final tokens: 4

Place: pMASmoked
Time: 4    Colors:    "Tobacco"   "Wrapping-Paper"
Time: 12    Colors:    "Tobacco"   "Wrapping-Paper"

Place: pWPSmoked
Time: 8    Colors:    "Matches"   "Tobacco"
>>
```

2.3 Fibonacci number generation

Shown below in figure-2-6 is a simple Colored Petri Net model for generating Fibonacci series of numbers. In this figure:

- **tInit** fires only once as **pInit** has only one initial token. When tInit fires, it adds color '1' to the two output tokens that are deposited into **pFibo**.
- **tFibo** is a "cold start" transition as it is not controlled by any input tokens (**tFibo** has no input places). Thus, **tFibo** is always enabled. **tFibo** will check the colors of the two latest tokens found in **pFibo**. **tFibo** will take the colors of the two latest tokens and make a new color (1+1 -> 2, 1+2 -> 3, 2+3 -> 5, and so on) for the output token that will be deposited into **pFibo**.

At the end of the simulation, there will be some tokens in **pFibo**. The color of these tokens represents the Fibonacci number, the earliest tokens have the first two numbers 1 and 1, and the latest token has the largest number (the last number generated in the series).

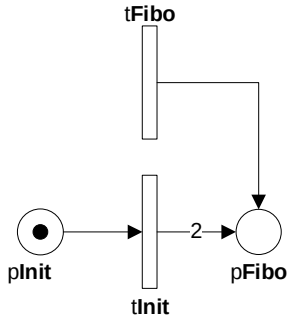

Figure 2-6: Fibonacci number generation

2.3.1 Example-49: Fibonacci number generation

PDF:

```
% Example-49: Fibonacci series generation
function [png] = fibonacci_pdf()
png.PN_name = 'Fibonacci series generation';
png.set_of_Ps = {'pInit', 'pFibo'};
png.set_of_Ts = {'tInit', 'tFibo'};
png.set_of_As = {'pInit','tInit',1, 'tInit','pFibo',2, ... % tInit
                 'tFibo','pFibo',1};     % tFibo
```

MSF:

```
% Example-49: Fibonacci number generation
% MSF: fibonacci.m
clear all; clc;
global global_info
global_info.COUNTER = 1; % for generating first five fibonacci numbers

png = pnstruct('fibonacci_pdf');

%%%% initial dynamics %%%%
dyn.m0 = {'pInit',1};
dyn.ft = {'allothers',1};
pni = initialdynamics(png, dyn);

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

Pre-processor for **tInit**: tInit_pre.m

```
% Pre-processor for tInit: tInit_pre.m
% tInit will fire only one time; it adds the 1st and 2nd Fibo numbers
% (1 and 1) as colors to the two tokens depositied into pFibo
function [fire, transition] = tInit_pre (transition)
transition.new_color = int2str(1);
fire = 1;
```

Pre-processor for **tFibo**: tFibo_pre.m

```
% tFibo a "cold start" transition. It always take the colors
%   of the two latest tokens in pFibo. From these two colors,
%   a new Fibonacci number is made, and put as the color
%   of the token depositied into pFibo
function [fire, transition] = tFibo_pre(transition)
global global_info
[set_of_tokIDs, nr_token_av] = tokenArrivedLate('pFibo', 2);
if ne(nr_token_av, 2), % pFibo doesn't have 2 tokens yet
    fire = 0; return
end

tokID1 = set_of_tokIDs(1);  % get tokID of the latest token in pFibo
tokID2 = set_of_tokIDs(2);  % tokID of the second latest token in pFibo
color_set1 = get_color('pFibo', tokID1); % colors of the leatest token
color_set2 = get_color('pFibo', tokID2); % colors of second leatest token
color1 = color_set1{1};  % extract textstring from color set
color2 = color_set2{1};  % extract textstring from color set

fibo_num1 = str2num(color1); % convert textstring into number
fibo_num2 = str2num(color2); % convert textstring into number
new_fibo = int2str(fibo_num1 + fibo_num2); % make the new number

transition.new_color = new_fibo;
transition.override = 1;
transition.selected_tokens = [tokID1 tokID2];
fire = 1;
% already produced 5 fibonacci number? Then, stop
global_info.COUNTER = global_info.COUNTER + 1;
if gt(global_info.COUNTER, 5), global_info.STOP_SIMULATION = 1; end
```

Simulation output:

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

Place: pFibo
Time: 1    Colors:    "1"
Time: 1    Colors:    "1"
Time: 2    Colors:    "2"
Time: 3    Colors:    "3"
Time: 4    Colors:    "5"
Time: 5    Colors:    "8"
>>
```

References

- T. Agerwala, and M. Flynn, “Comments on Capabilities, Limitations and ‘Correctness’ of Petri Nets,” Proceedings of the First Annual Symposium on Computer Architecture, New York: ACM, pp. 81–86, 1973.
- R. Davidrajuh (2013) “Revisiting Petri Net modeling of the Cigarette Smokers' Problem: A GPenSIM Approach.” European Modelling Symposium 2013, Manchester, UK, 20-22 November 2013. IEEE, DOI 10.1109/EMS.2013.34, pp. 185-190.
- R. Davidrajuh (2015) “Benchmarking GPenSIM”. Lecture Notes in Electrical Engineering, Springer, Volume 312 (1), s. 373-379, 2015.
- R. Keller, “Vector Replacement Systems: A Formalism for Modeling Asynchronous Systems,” Technical Report 117, Computer Science Laboratory, Princeton University, Princeton, New Jersey, January 1974.
- S. Kosaraju, “Limitations of Dijkstra’s Semaphore Primitives and Petri Nets,” Operating Systems Review, Vol. 7, No. 4, pp. 122–126, October 1973.
- D. Parnas, “On a Solution to the Cigarette Smokers’ Problem (Without Conditional Statements),” Communications of the ACM, Volume 18, Number 3, (March 1975), pp. 181-183.
- S. Patil, “Limitations and Capabilities of Dijkstra’s Semaphore Primitives for Coordination Among Processes,” Computation Structures Group Memo 57, Project MAC, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1971.
- J. L. Peterson, *Petri Net Theory and the Modeling of Systems*; Prentice-Hall: Englewood Cliffs, NJ, USA, 1981.