Addendum-4 to the book **Davidrajuh, R. "Modeling Discrete-Event Systems with GPenSIM: An Introduction", Springer, 2018**

Building Modular Petri Net models

By Reggie Davidrajuh, reggie.davidrajuh@uis.no, © 15 August 2018.

This chapter discusses modularization as a technique for modeling of large discrete-event systems. Though modularization is supported in many Petri Net tools, it was not possible before in GPenSIM. The newer version of GPenSIM (version 10) allows modularization so that flexibility (ability to add or change functionality) and comprehensibility (readability) of the models can be improved. Modularization reduces the development time of large Petri Net models too, as separate groups can develop individual modules at the same time. Also, modularization also increases the robustness (less prone to error) of the models (Davidrajuh, 2017).

The earlier versions of GPenSIM (before v.10) provided a crude facility for modularization known as 'basic grouping (segmenting) of related places and transitions.' We will study about this approach first. The latter sections of this chapter are for the modern approach.

1. Modular Model Building

1.1 Segments by grouping of related elements

When implementing a large Petri Net in GPenSIM, we may end up with a PDF file that consists of a large number of places and transitions. A large number of places and transitions causes a vast amount of arcs to be coded in the PDF. Coding a large number of arcs is tiresome and one of the main reason for errors. In this case, we can divide the system into smaller functional entities and create separate PDFs, one for each of these entities. Thus, the approach of basic grouping only offers some ease in coding the PDF files.

1.1.1 Example-61: Model for Cross-Talk

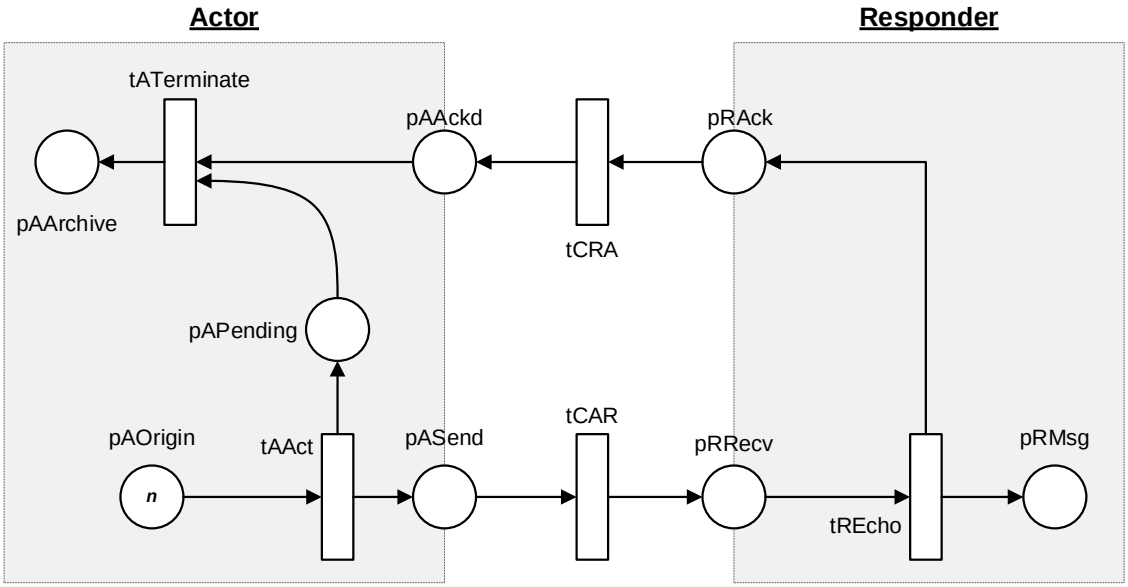

Figure 1-1: Actor and Responder

The Petri Net model that is shown in figure-1-1 represents an actor-responder model for simple communication between two agents (Reisig, 2013). The communication happens in cycles, starting with message creation and sending (by the actor), receiving and acknowledging (by the responder), and archiving (actor). The actor actively creates a message and send it to the responder. The actor then waits for the acknowledgment. The responder waits for any messages from the actor. When a message is received, the responder acknowledges the message and then waits for further messages. In this example, we assume that there is only one actor-responder pair. Thus there is no need for the actor or the responder to embed their IDs in the messages and the acknowledgments.

The Petri Net model shown in figure-1-1 can be partitioned into two segments, the actor, and the responder. Also, because we have two separate segments, we also need some transitions to connect these two segments. Thus, we have a connection segment (consisting of two transitions **tCAR** and **tCRA**) also. Thus, the three segments resulting from the model that is shown in figure-1-1 are:

- **Actor** consisting of the transitions **tATerm** and **tAAct** and the places **pAAckd**, **pASend**, **pAArchive**, and **pAPending**. The place **pOrigin** is to limit the number of messages generated (limited by the number of initial tokens on it). The place **pAArchive** is to archive the successfully transmitted messages.
- **Responder** consisting of the transitions **tREcho** and the places **pRRecv**, **pRAck**, and **pRMsg**. Place **pRMsg** is to store the received messages.
- **Connector** consisting of the transitions **tCRA** and **tCAR**. The arcs that connect **tCAR** and **tCRA** to the actor and responder modules also belong to the connector module.

We define one PDF file for each segment.

Actor (act_pdf.m):

```
% PDF: Actor (actor_pdf)
function [png] = actor_pdf()
png.PN_name = 'ACTOR';
png.set_of_Ps = {'pOrigin','pAAckd','pAArchive','pAPending',
'pASend'};
```

```
png.set_of_Ts = {'tAAct','tATerm'};
png.set_of_As = {'pOrigin','tAAct',1, ... % tAAct input
    'tAAct','pASend',1,'tAAct','pAPending',1, ...% tAAct outputs
    'pAPending','tATerm',1,'pAAckd','tATerm',1,...% tATerm inputs
    'tATerm','pAArchive',1};                    % tATerm outputs
```

Responder (responder_pdf.m):

```
% PDF: Responder (responder_pdf)
function [png] = responder_pdf()
png.PN_name = 'RESPONDER';
png.set_of_Ps = {'pRAck', 'pRRecv', 'pRMsg'};
png.set_of_Ts = {'tREcho'};
png.set_of_As = {'pRRecv','tREcho',1, ...    % tREcho inputs
    'tREcho','pRAck',1, 'tREcho','pRMsg',1}; % tREcho outputs
```

Connector (connect_pdf.m):

```
% PDF: Connector (connector_pdf.m)
function [png] = connector_pdf()
png.PN_name = 'CONNECTOR';
png.set_of_Ps = {};
png.set_of_Ts = {'tCAR', 'tCRA'};  % dummy connector trans
png.set_of_As = {'pRAck','tCRA',1,  'tCRA','pAAckd',1, ...  % tCRA
           'pASend','tCAR',1, 'tCAR','pRRecv',1};    % tCAR
```

Finally, in the **MSF**, we need to declare all the three PDF files.

```
% Example-61: Actor-Responder
global global_info
global_info.Message_Serial_Number = 0;  % MSNo starts with 0
global_info.STOP_AT = 100;

pns = pnstruct({'actor_pdf', 'responder_pdf', 'connector_pdf'});

dyn.m0 = {'pOrigin',10};
dyn.ft = {'tCAR',10, 'tCRA',10, 'allothers',1}; %
pni = initialdynamics(pns, dyn);
sim = gpensim(pni);

prnstate('Final markings: ');
prnVirtualState('Final virtual tokens: ');
prnfinalcolors(sim);
```

In the COMMON_PRE, we will code the following actions:

- **tAAct** will create output tokens (for **pASend** and **pAPending**) with one color, indicating the message number.
- Whenever a token arrives in **pAAckd**, **tATerm** will inspect the token, get the message number and then use this message number to remove a token from **pAPending** that has the same message number.
- Let us make the scenario a little more interesting, by assuming that the communication lines between the two agents are very unreliable. Thus, the transitions **tCAR** and **tCRA** that represent the communication lines fire only 20% of the time.

COMMON_PRE:

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

switch trans.name
    case 'tAAct'  % create a new message
        msr = global_info.Message_Serial_Number;
        msr = msr + 1;
        global_info.Message_Serial_Number = msr;
        trans.new_color = int2str(msr);
        fire = 1;

    case {'tCAR', 'tCRA'} % transmit message (tCAR) or ack (tCRA)
        % the communication line is unreliable !!
        % thus, tCAR and tCRA fire only 20% of the time
        randomNr = rand;
        fire = gt(randomNr, 0.8);

    case 'tATerm' % process acknowledgement from Responder
        tokID1 = tokenAny('pAAckd', 1); % look into token in pAAckd
        colors = get_color('pAAckd', tokID1); % get the color
        %look for the same colored token in pAPending
        tokID2 = tokenEXColor('pAPending',1,colors);
        trans.selected_tokens = [tokID1 tokID2];
        fire = all([tokID1 tokID2]);
        if fire, disp(['Successful: ',colors{1}]);end%echo succeeded

    otherwise % case tREcho
        fire = 1; % just fire
end
```

Simulation result shows that due to the malfunctioning communication lines (**tCAR** and **tCRA**), the messages were not sent and acknowledged in the order of the generation. Some of the messages (messages '1' and '3') are still waiting in the buffers.

```
Successful: 2
Successful: 10
Successful: 9
Successful: 8
Successful: 7
Successful: 6
Successful: 5
Successful: 4
Final markings:  7pAArchive + 2pAPending + pRAck + 9pRMsg
Final virtual tokens:  pAAckd + pAPending + pASend
```

1.2 Modular Model Building

This subsection introduces the modular model building, which is new in GPenSIM (available only from version 10). In the modular model building, there are two fundamentally new issues:

- IO ports: For a module, transitions function as the input and output ports (IO ports).
- Modular processor files: in addition to specific and COMMON processor files, we can have modular processor files too.

1.2.1 Declaring modules with the IO Ports

Figure-1-2 shows a system with three **modules** namely, **Alfa**, **Beta**, and **Gamma**. For this system, there must be at least three PDFs, one for each module. Also, in each of these PDFs, **there must be a declaration about the IO ports of these modules**. For example, the PDF for the module Alfa:

```
% PDF for module Alfa:
function [png] = Alfa_pdf()
png.PN_name = 'Alfa'; % name of the module
png.set_of_Ps = {'pA1','pA2','pA3', 'pA4'};
png.set_of_Ts = {'tAI1','tAI2','tAO1','tAO2', 'tAx1','tAx2'};
png.set_of_As = {'tAI1','pA1',1, 'tAI2','pA2',1,...    % tAI1, tAI2
    'pA1','tAx1',1, 'tAx1','pA3',1, ...% tAx1
    'pA2','tAx2',1, 'tAx2','pA4',1, ...% tAx1
    'pA3','tAO1',1, 'pA4','tAO2',1};   % tAO1, tAO2
png.set_of_Ports = {'tAI1','tAI2','tAO1','tAO2'}; % IO ports of this module
```

The last statement states that there are four transitions (**tAI1**, **tAI2**, **tAO1**, and **tAO2**) that function as the IO ports of the module. **It is this statement that declares Alfa as a module and not as a segment**.

Let us take a look at the PDF for the module Beta:

```
% PDF for module Beta:
function [png] = Beta_pdf()
png.PN_name = 'Beta';  % name of the module
png.set_of_Ps = {'pB1','pB2','pB3', 'pB4'};
png.set_of_Ts = {'tBI1','tBO1', 'tBx1','tBx2'};
png.set_of_As = {...
    'pB1','tBx1',1, 'tBx1','pB3',1, ... % tBx1
    'pB3','tBO1',1, ... % tBO1
    'tBI1','pB2',1, ... % tBI1
    'pB2','tBx2',1, 'tBx2','pB4',1}; % tBx2
png.set_of_Ports = {'tBI1','tBO1'}; % IO ports for this module
```

Here again, the last statement of the PDF states that there are two transitions (**tBI1** and **tBO1**) that function as the IO ports. Because of this statement, the PDF represents the module Beta. Finally, the PDF for the module Gamma:

```
% PDF for module Gamma:
function [png] = Gamma_pdf()
png.PN_name = 'Gamma';  % name of the module
png.set_of_Ps = {'pG1','pG2','pG3', 'pG4'};
png.set_of_Ts = {'tGx1','tGx2'};
png.set_of_As = {...
    'pG1','tGx1',1, 'tGx1','pG3',1, ... % tGx1
    'pG2','tGx2',1, 'tGx2','pG4',1};    % tGx2
png.set_of_Ports = {}; % IO ports for this module
```

Even though the last statement of the PDF for Gamma declares that there are no IO ports, the presence of this statement declares Gamma as a module. Even though there are places at the gates of the modules (e.g., **pB1** and **pB4** in Beta module, and **pG1**-**pG4** in Gamma module), only transitions can function as IO ports.

NOTE: Only the transitions can function as IO ports of a module.

NOTE: In a PDF, if there is a statement (declaration) identifying the IO ports, then the PDF defines a module. Otherwise (no statement on IO ports), the PDF defines a segment.

1.2.2 Introducing the modular processor files

In this subsection, we are going to see two more processor files – the modular processor files. So far, we have used the following four processor files:

- Specific pre-processor file: specific pre-processor file is for coding pre-conditions for firing a specific transition. This means, for a Petri Net with ***n*** transitions, there can be up to ***n*** specific pre-processor files, where each pre-processor file coding the pre-conditions of a specific transition.
- COMMON_PRE file: This file declares the pre-conditions for all the enabled transition. COMMON_PRE has the universal scope, as all the enabled transitions are visible here.
- Specific post-processor file: specific post-processor file is for coding post-actions when a specific transition completes firing. This means, for a Petri Net with ***n*** transitions, there can be up to ***n*** specific post-processor files, each post-processor file coding the post-actions of a specific transition.
- COMMON_POST file: This file declares the activities that are to be done after transitions complete firing. COMMON_POST has the universal scope, as all the transitions that complete firing is visible here.

In addition to the four types of processor files mentioned above, when we use modules, **we can use two more modular processor files per module**: the modular pre-processor file **MOD_*_PRE** and the modular post-processor file **MOD_*_POST**, where the asterisk (*) represents the name of the module.

CAUTION! Names of the modular processor files must follow strict naming policy, as they will be chosen and run automatically. For example, if the name of a module is 'Alfa' (as declared in the PDF), then the modular pre-processor for the module 'Alfa' must be named 'MOD_Alfa_PRE.m'. Similarly, the modular post-processor file for the module 'Alfa' must be named 'MOD_Alfa_POST.m.'

The transitions residing inside a module (the internal transitions of a module) possess limited visibility (module visibility only, and not universal visibility); thus, the internal transitions are not visible in the common processor files (COMMON_PRE and COMMON_POST). They are visible only in the modular processor files (MOD_*_PRE and MOST_*_POST files), in addition to their specific processor files.

The input and output transitions of the module (the IO ports) are visible in the modular processor files, as they are part of the module. Also, the IO ports are also visible in the COMMON_PRE and COMMON_POST files.

- Modular pre-processor file (MOD_*_PRE file): This file defines the pre-conditions for all the transitions of a module (the internal transitions of the module as well as the IO ports of the module). However, the internal transitions are not visible in the COMMON_PRE file whereas the IO ports are visible there.

- Each module can have their own modular pre-processor file (MOD_*_PRE file). The same goes for the modular post-processor file (MOD_*_POST file).

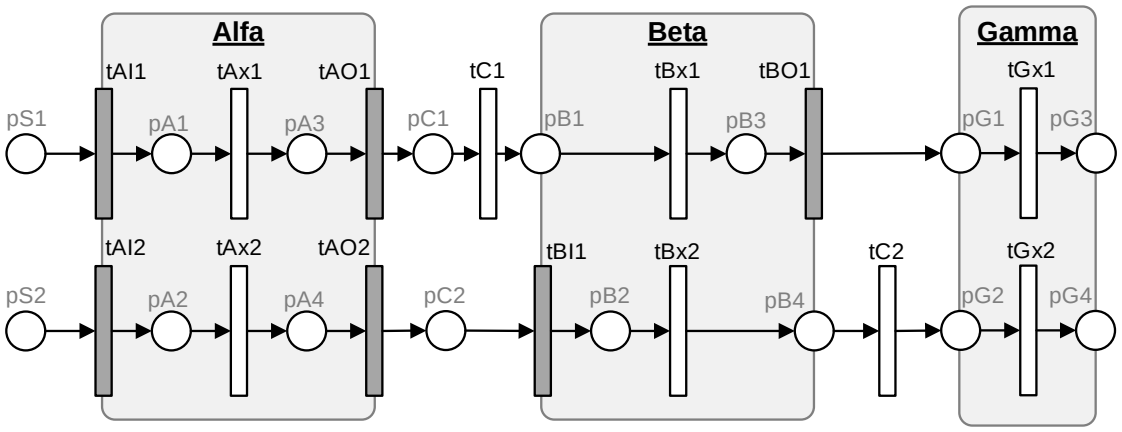

Figure 1-2: A modular Petri Net model

Let us look into the model shown in figure-1-2. The model consists of three modules, **Alfa**, **Beta**, and **Gamma**. The transitions of the system:

- **Alfa**: transitions **tAI1**, **tAI2**, **tAO1**, and **tAO2** are the IO ports. **tAx1** and **tAx2** are the internal transitions of Alfa.
- **Beta**: transitions **tBI1** and **tBO1** are the IO ports. **tBx1** and **tBx2** are the internal transitions of Beta.
- **Gamma**: there are no IO ports. **tGx1** and **tGx2** are the internal transitions of Gamma.
- **tC1** and **tC2** are the inter-modular connecting transitions; these two transitions do not belong to any modules.

The processor files for the system:

- Up to two common processor files: COMMON_PRE and COMMON_POST,
- Up to six modular processor files:
  - For Alfa module: pre-processor MOD_Alfa_PRE, post_processor MOD_Alfa_PRE,
  - For Beta module: pre-processor MOD_Beta_PRE, post_processor MOD_Beta_PRE, and
  - For Gamma module: pre-processor MOD_Gamma_PRE, post_processor MOD_Gamma_PRE
- Up to twenty-eight specific processor files: there are fourteen transitions in the system. Each transition can have their own specific pre- and post-processors.

The table-1-I to III summarizes the visibility of the different transitions of the system in the different processor files.

Table-1-I: Transitions of the module Alfa and their visibility in different processor files.

The processor file	***The IO ports*** ***(tAI1, tAI2, tAO1, and tAO2)***	***The internal transitions*** ***(tAx1 and tAx2)***
specific_pre	**Yes**, in their own specific_pre file.	
specific_post	**Yes**, in their own specific_post file.	
MOD_Alfa_PRE	**Yes**, as *all* the transitions of the **Alfa** module are visible in the **MOD_Alfa_PRE** file.	
MOD_Alfa_POST	**Yes**, as *all* the transitions of the **Alfa** module are visible in the **MOD_Alfa_POST** file.	

COMMON_PRE	**Yes**. The IO ports are visible in the COMMON_PRE.	**No**. Internal transitions are not visible in the COMMON_PRE.
COMMON_POST	**Yes**. The IO ports are visible in the COMMON_POST.	**No**. Internal transitions are not visible in the COMMON_POST.

Table-1-II: Transitions of the module Beta and their visibility in different processor files.

The processor file	***The IO ports (tBI1** and **tBO1)***	***The internal transitions (tBx1** and **tBx2)***
specific_pre	**Yes**, in their own specific_pre file.	
specific_post	**Yes**, in their own specific_post file.	
MOD_Beta_PRE	**Yes**, as *all* the transitions of the **Beta** module are visible in the **MOD_Beta_PRE** file.	
MOD_Beta_POST	**Yes**, as *all* the transitions of the **Beta** module are visible in the **MOD_Beta_POST** file.	
COMMON_PRE	**Yes**. The IO ports are visible in the COMMON_PRE.	**No**. Internal transitions are not visible in the COMMON_PRE.
COMMON_POST	**Yes**. The IO ports are visible in the COMMON_POST.	**No**. Internal transitions are not visible in the COMMON_POST.

Table-1-III: Internal transitions of the module Gamma and their visibility in different processor files (there are no IO ports in the module).

The processor file	***The internal transitions (tGx1** and **tGx2)***
specific_pre	**Yes**, in their own specific_pre file.
specific_post	**Yes**, in their own specific_post file.
MOD_Gamma_PRE, and **MOD_Gamma_POST**	**Yes**, as the internal transitions of the **Gamma** module are visible in the **MOD_Gamma_PRE** and the **MOD_Gamma_POST** files.
COMMON_PRE, and **COMMON_POST**	**No**. Internal transitions are not visible in the COMMON_PRE or COMMON_POST.

Table-1-IV: The inter-modular connecting transitions and their visibility in different processor files.

The processor file	***The connecting transitions tC1** and **tC2***
specific_pre	**Yes**, in their own specific_pre file.
specific_post	**Yes**, in their own specific_post file.
MOD_*_PRE **MOD_*_POST**	**No**. The connecting transitions do not belong to any modules. Thus, they are not visible in any MOD_*_PRE or MOD_*_POST files.
COMMON_PRE, and **COMMON_POST**	**Yes**. Since the connecting transitions are <u>not</u> internal to any modules, they are visible in the COMMON_PRE or COMMON_POST.

Summary:

- The **IO port tAI1** of the module **Alfa**: **tAI1** is visible in six processor files: 1) tAI1_pre, 2) tAI1_post, 3) MOD_Alfa_PRE, 4) MOD_Alfa_POST, 5) COMMON_PRE, and 6) COMMON_POST.
- The **internal transition tBx1** of the module **Beta**: **tBx1** is visible in at most four processor files: 1) tBx1_pre, 2) tBx1_post, 3) MOD_Beta_PRE, and 4) MOD_ Beta_POST.
- The connecting (**module-less**) transition **tC2**: **tC2** is visible in at most four processor files: 1) tC2_pre, 2) tC2_post, 3) COMMON_PRE, and 4) COMMON_POST.

In the following example, let us experiment with the visibility of transitions.

1.2.3 Example-62: Testing the visibility of transitions

The test model for this example is shown in figure-1-3.

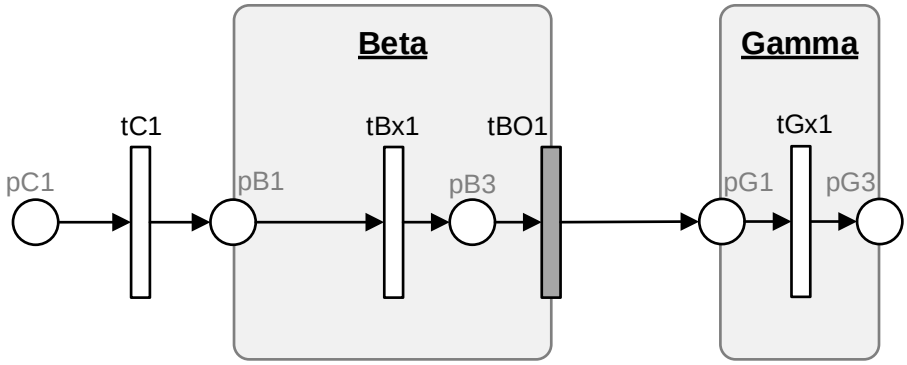

Figure 1-3: Testing the visibilities of transitions in the processor files

The model consists of just two simple modules namely, **Beta** and **Gamma**. There are three PDFs, the first PDF is for Beta (module), the second PDF is for Gamma (module), and the third for the connector (segment). The connector module consists of just two elements: **pC1** and **tC1**.

PDF for **Beta** module:

```
% Example-62: Testing visibility of transitions in processor files
function [png] = Beta_pdf()
png.PN_name = 'Beta';
png.set_of_Ps = {'pB1','pB3'};
png.set_of_Ts = {'tBO1', 'tBx1'};
png.set_of_As = {'pB1','tBx1',1, 'tBx1','pB3',1, ... % tBx1
                 'pB3','tBO1',1}; % tBO1

png.set_of_Ports = {'tBO1'}; % IO ports for this module
```

PDF for **Gamma** module:

```
function [png] = Gamma_pdf()
png.PN_name = 'Gamma';
png.set_of_Ps = {'pG1','pG3'};
png.set_of_Ts = {'tGx1'};
png.set_of_As = {'pG1','tGx1',1, 'tGx1','pG3',1}; % tGx1

png.set_of_Ports = {}; % IO ports for this module
```

PDF for **connector** (segment):

```
function [png] = connector_pdf()
png.PN_name = 'inter-modular connections';
png.set_of_Ps = {'pC1'};
png.set_of_Ts = {'tC1'};
png.set_of_As = {'pC1','tC1',1, 'tC1','pB1',1, ... % tC1
                 'tBO1','pG1',1}; % tBO1
```

The **MSF**:

```
global global_info
```

```
global_info.STOP_AT = 100;

pns = pnstruct({'Beta_pdf', 'Gamma_pdf', 'connector_pdf'});

dyn.m0 = {'pC1',1};
dyn.ft = {'allothers',1}; %
pni = initialdynamics(pns, dyn);
sim = gpensim(pni);
```

What we are trying to test is simple. We are to create the two common processor files (COMMON_PRE and COMMON_POST) and the modular processor files for the modules Beta and Gamma (MOD_Beta_PRE, MOD_Beta_POST, MOD_Gamma_PRE, and MOD_Gamma_POST). We are not going to create the specific processor files for the individual transitions as it is obvious that each transition is visible in their own specific pre- and post-processor files.

In all the six processor files, we are going to add only one statement: just print the name of the enabled transition that is executing the processor file. For example, in **COMMON_PRE**:

```
function [fire, trans] = COMMON_PRE(trans)
disp([mfilename, ' is checked for ', trans.name]);
fire = 1;
```

COMMON_POST:

```
function [] = COMMON_POST(trans)
disp([mfilename, ' is checked for ', trans.name]);
```

MOD_Beta_PRE:

```
function [fire, trans] = MOD_Beta_PRE(trans)
disp(['* ',mfilename, ' is checked for ', trans.name]);
fire = 1;
```

MOD_Beta_POST:

```
function [] = MOD_Beta_POST(trans)
disp(['* ', mfilename, ' is checked for ', trans.name]);
```

MOD_Gamma_PRE is exactly the same as MOD_Beta_PRE, and MOD_Gamma_POST is the same as MOD_Beta_POST.

The simulation results is shown below:

```
COMMON_PRE is checked for tC1
COMMON_POST is checked for tC1
* MOD_Beta_PRE is checked for tBx1
* MOD_Beta_POST is checked for tBx1
* MOD_Beta_PRE is checked for tBO1
COMMON_PRE is checked for tBO1
* MOD_Beta_POST is checked for tBO1
COMMON_POST is checked for tBO1
* MOD_Gamma_PRE is checked for tGx1
* MOD_Gamma_POST is checked for tGx1
>>
```

The result shows that the connecting transition **tC1** (which does not belong to any modules) is visible only in the common processor files. **tBO1**, being an IO port of a module is also visible

in the common processor files. Also, **tBO1** is visible in the modular processor files for Beta, as **tBO1** is a member of the Beta module.

tBx1 is an internal member of the module Beta. Thus, **tBx1** is visible only in the modular processor files for Beta. Finally, as an internal member of the module Gamma, **tGx1** is visible only in the modular processor files for Gamma. (Of course, all these transitions are visible in their own specific processor files, which we did not bother to test.) The findings are summarized below:

The visibility of the transitions in the processor files:
There are three types of transitions in modular Petri Net models:

1. Transitions that are IO ports of modules: these IO ports are visible in the common processor files. Also, they are visible in the modular processor files of the modules they belong to.
2. Transitions that are internal elements of modules: these internal transitions of modules are visible only in the modular processor files of the modules they belong to.
3. Transitions that are stand-alone (module-less, they are not part of any modules): these stand-alone transitions are visible only in the common processor files.
4. (all the transitions are visible in their own specific processor files)

1.2.4 Example-63: Modular model for Cross-Talk

In this example, we are going to make a modular version of the example-61 'Actor-Responder.' In the example-61, we grouped related places and transitions into three segments, each represented by their own PDF file. In doing so, we will have smaller PDF files that are easy to create, and for checking against errors. There were no other benefits. In this example, we shall see how we can modularize a Petri Net model. By modularization, we can reap the following benefits:

- Testing the modules individually, and then
- Connecting the modules to form the complete model: when joining the modules together, the codes for the modules should not be changed, and the code for joining must be minimal.

In the previous example-61, we had three segments, namely the actor, the responder, and the connector. In this example, we will change the three segments into modules: **Actor**, **Responder**, and **Transmitter**, respectively.

1.2.5 The Actor module

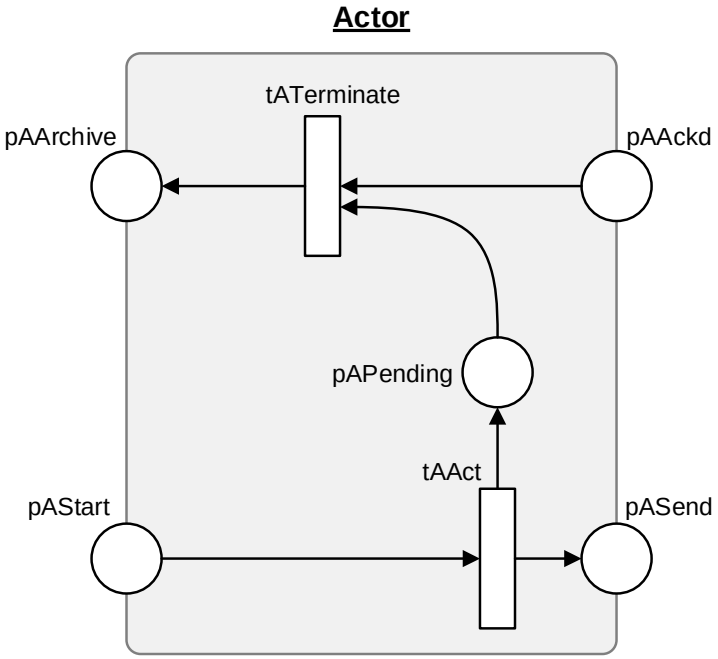

Figure 1-4: The Actor module

The actor module is shown in the figure-1-4. It looks almost the same as the segment shown in the figure-1-1. However, the actor module is different, as in its PDF there is the declaration "**`png.set_of_Ports = {}`**" that declares the actor as a module (a module with no IO ports).

PDF for the actor module 'Actor_pdf.m':

```
% Example-63: Modular Crosstalk with Actor Responder
function [png] = Actor_pdf()
png.PN_name = 'Actor';
png.set_of_Ps = {'pAAckd','pAArchive','pAPending', 'pASend'};
png.set_of_Ts = {'tAAct','tATerm'};
png.set_of_As = {...
    'tAAct','pASend',1, 'tAAct','pAPending',1, ...% tAAct outputs
    'pAPending','tATerm',1,'pAAckd','tATerm',1,...% tATerm inputs
    'tATerm','pAArchive',1,...                    % tATerm outputs
     };
png.set_of_Ports = {}; % no input/out ports for this module
```

The actor module can be independently tested with testing drivers and stub, as shown in figure-1-5. To test the Actor module independently of the other modules, we need to create only the modular pre-processor for Actor, MOD_Actor_PRE file. There is no need for the modular post-processor file as there are no post-firing actions for the transitions **tAAct** and **tATerm**. Once we have created MOD_Actor_PRE and thoroughly tested it, there is no need to change it unless new functionality has to be added to the module. Also, MOD_Actor_PRE will contain **ONLY** the firing pre-conditions for **the transitions (tAAct and tATerm) of the Actor module**. Thus, MOD_Actor_PRE will isolate and confine the firing pre-conditions of the transitions of Actor. The code for **tAAct** and **tATerm** will not be visible in common processors. **This is data abstraction**.

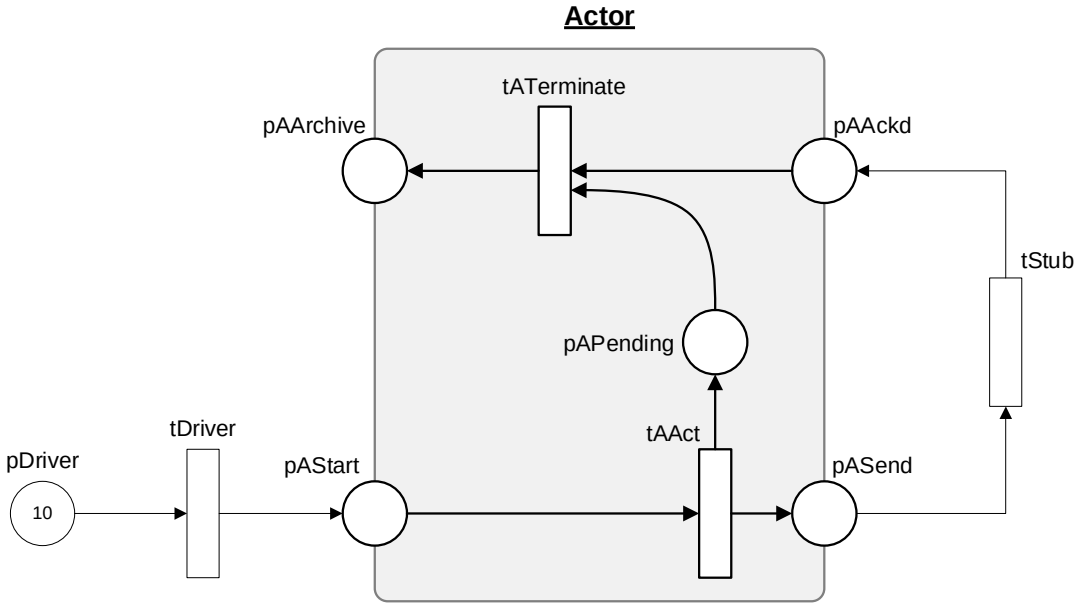

Figure 1-5: Testing the Actor module

```
% The modular pre-processor for the Actor module
function [fire, trans] = MOD_Actor_PRE(trans)
% The following transitons are visible here: tAAct, tATerm
global global_info
switch trans.name
    case 'tAAct' % create a message
        SR = global_info.Shift_Register;
        SR = circshift(SR,1);  % make an integer (1,2, or 3) rotationally
        global_info.Shift_Register = SR;
        one_two_three = SR(1);
        msr = global_info.Message_Serial_Number;
        msr = msr + 1;
        global_info.Message_Serial_Number = msr;
        trans.new_color = {int2str(one_two_three), int2str(msr)};
        fire = 1;

    case 'tATerm' % process an acknowledgement from Responder
        tokID = tokenAny('pAAckd', 1); % look into the token in pAAckd
        colors = get_color('pAAckd', tokID);
        tokID2 = tokenEXColor('pAPending',1,colors);
        trans.selected_tokens = tokID2;
        fire = tokID2;
    otherwise %
        error('not possible');
end
```

1.2.6 The Responder module

The responder module is shown in figure-1-6, together with a testing driver **tDrv2**. Just like the Actor module, the Responder module is also declared as a module by the statement "**`png.set_of_Ports = {}`**" in its PDF. PDF for the Responder module 'Responder_pdf.m':

```
function [png] = Responder_pdf()
png.PN_name = 'Responder';
png.set_of_Ps = {'pRAck', 'pRRecv', 'pRMsg'};
png.set_of_Ts = {'tREcho'};
```

```
png.set_of_As = {'pRRecv','tREcho',1, ...        % tREcho inputs
    'tREcho','pRAck',1, 'tREcho','pRMsg',1};     % tREcho outputs
png.set_of_Ports = {}; % no input/out ports for this module
```

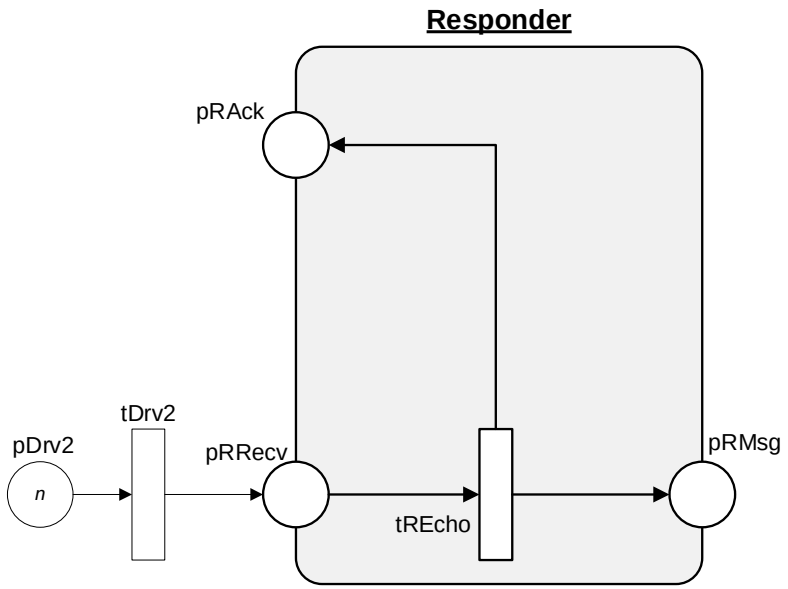

Figure 1-6: The Responder module

There is no need to create a modular pre- or post-processor file for the Responder module. This is because there are no pre-conditions or post-actions for the transition **tREcho**. However, just to make the Responder module visible, we may want to create an empty modular pre-processor. MOD_Responder_PRE:

```
% The modular pre-processor for the Responder module
function [fire, trans] = MOD_Responder_PRE(trans)
% ONLY the internal transition "tREcho" is visible here.
% Since there is no pre-conditions attached with tREcho,
% just allow it to fire.
fire = 1;
```

1.2.7 The Transmitter module

The Transmitter module is shown in the figure-1-7, together with testing drivers and stubs. PDF for the Transmitter module 'Transmitter_pdf.m':

```
% PDF for Transmitter module (Transmitter_pdf.m)
function [png] = Transmitter_pdf()
png.PN_name = 'Transmitter';
png.set_of_Ps = {};
png.set_of_Ts = {'tTAR','tTRA'}; % transmitters in both directions
png.set_of_As = {};
png.set_of_Ports = {}; % No IO ports for this module
```

In the modular pre-processor for the Transmitter module, we will implement the firing conditions for the transitions **tTAR** and **tTRA**.

```
% The modular pre-processor for the Transmitter module
function [fire, trans] = MOD_Transmitter_PRE(trans)
```

```
% The following transitons are visible here: tTAR, tTRA

switch trans.name
    case {'tTAR', 'tTRA'}
        % the communication line is unreliable !!
        % works only 20% of the time
        randomNr = rand;
        fire = gt(randomNr, 0.8);
end
```

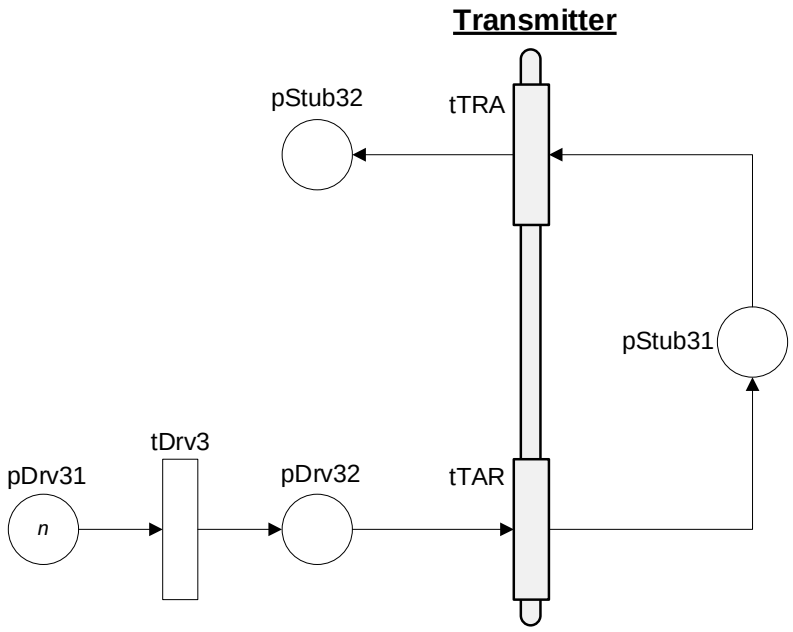

Figure 1-7: The Transmitter module

1.2.8 The complete model: Putting all the modules together

The complete Actor-Responder model is shown in figure-1-8. This model is obtained simply by placing the three modules (Actor, Responder, and Transmitter) together. Also, we need the connector (segment) to connect the three modules to make the whole system. The PDF for the connector segment (there will not be any statement declaring the IO ports):

```
% PDF for the segment Connector (connector_pdf.m)
function [png] = connector_pdf()
png.PN_name = 'Connector';
png.set_of_Ps = {'pSource'};
png.set_of_Ts = {'tSource'};
png.set_of_As = {...
    'pSource','tSource',1,  'tSource','pAStart',1, ...  % tSource
    'pASend','tTAR',1, 'tTAR','pRRecv',1, ...     % tTAR
    'pRAck','tTRA',1, 'tTRA','pAAckd',1, ...     % tTAR
    };
```

The Main Simulation File for the model is given below.

MSF:

```
% Example-63: Modular Actor-Responder
global global_info
```

```
global_info.Message_Serial_Number = 0;  % MSNo starts with 0
global_info.STOP_AT = 100;

% we have three modules and one connector
pns = pnstruct({'Actor_pdf', 'Responder_pdf', 'Transmitter_pdf',...
    'connector_pdf'});

dyn.m0 = {'pSource',10};
dyn.ft = {'tTAR',10, 'tTRA',10, 'allothers',1}; %
pni = initialdynamics(pns, dyn);
sim = gpensim(pni);

prnstate('Final markings: ');
prnVirtualState('Final virtual tokens: ');
prnfinalcolors(sim);
```

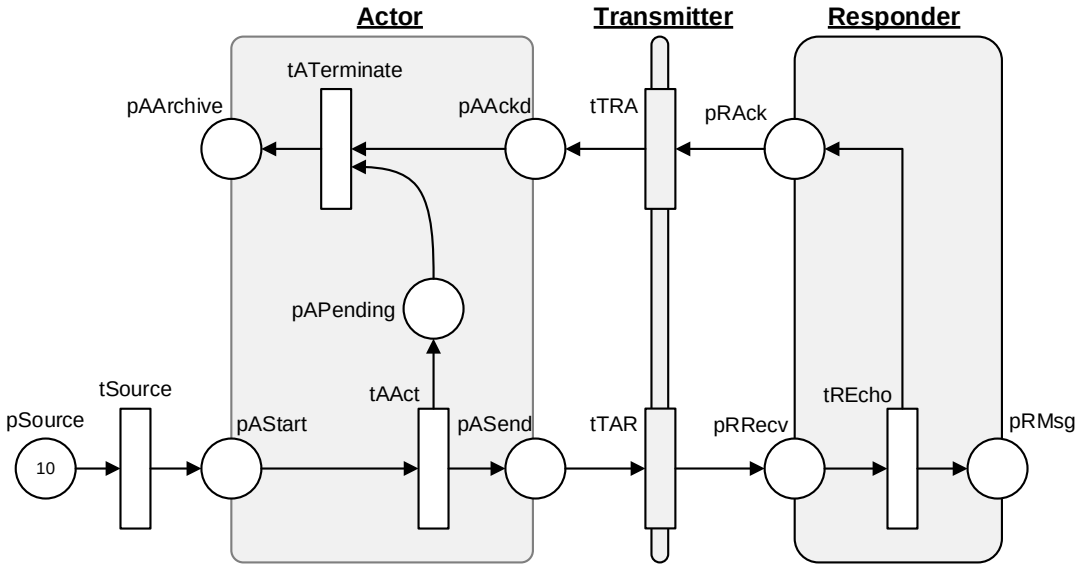

Figure 1-8: The modular Actor-Responder model

1.3 Advantages of modular modeling building

So, what's the big fuzz about the modular model building? First of all, there are no common processor files (COMMON_PRE and COMMON_POST) in the modular example-63. This is because we have only one transition (**tSource**) that is visible in the common processors and even this transition does not have any firing pre-conditions or post actions to be coded in the processor files. Usually, in a non-modular model, we end up with a large COMMON_PRE file, collecting all the enabling conditions in one place. However, in the modular approach, the enabling conditions (and post actions) of a group of transitions that belongs to a module are kept in their own modular processor, thus leaving the common processor files slim. Suppose different groups of programmers are involved in developing the modules, these groups of programmers can focus just on developing their modules, and testing them independently. The final model will consist of several independent modular processor files and a thin common processor files which only have to deal with the connecting transitions, and the interfacing transitions (IO ports) of the modules, if any. In summary, modular approach paves:

- Independent development and testing of the modules,

- Slim common processors that only deals with connecting (inter-modular) transitions, and interfacing transitions (IO ports), and
- Modular processor hides away the implementation details of internal transitions of the modules (data abstraction).
- Since different groups of programmers can focus on developing different modules at the same time, the development time is also reduced.

Note: Modular model development paves independent development and testing of the modules. Different groups of programmers can focus on developing different modules at the same time. Thus, development time is also reduced.

Note: There are no 'shared transitions' in GPenSIM. Some Petri Net tools provide 'shared transitions' to be used to synchronize transitions that are residing inside different modules. With the absence of this shared transition, it is not possible to synchronize activities (transitions) that reside in different modules in GPenSIM. Synchronization of different modules must happen at the inputs (IO ports) of the modules. Thus, synchronization of different modules is one of the purposes of IO ports.

References

- R. Davidrajuh, "Modular Petri Net models of Communicating Agents: A GPenSIM Approach," International Joint Conference SOCO'17, León, Spain, September 6–8, 2017, Proceedings, Advances in Intelligent Systems and Computing 649, Springer, DOI 10.1007/978-3-319-67180-2_32