PetriScript
Reference Manual
1.0

Alexandre Hamez
Xavier Renault

Contents

Introduction

The CPN-AMI platform provides many tools to work on Petri nets such as verifying or model-checking tools. It was easily possible to graphically design simple Petri nets with Macao[1], but various works made internally at LIP6 reveal that it was needed to automate such task.

Therefore PetriScript has been designed to provide some facilities in modelling places-transition and coloured Petri nets within the CPN-AMI platform[2].

Its main purpose is to automate modelling operations on Petri nets such as merging, creating, and connecting nodes. Thus, it supports almost everything needed like macros, loops control, lists, string and arithmetic expressions, ..., and avoids to the maximum the intervention of the user. Its syntax is more or less Ada-like.

Chapter 1 will introduce the basic aspects of PetriScript such as loop, variables, ..., while chapter 2 will explain the manipulation of Petri nets via PetriScript .

A Backus-Naur Form is provided in appendix A.

[1] `www-src.lip6.fr/logiciels/mars/MACAO`

[2] `www-src.lip6.fr/logiciels/mars/CPNAMI`

Chapter 1

PetriScript basics

Before learning the manipulation of Petri nets with PetriScript , which is presented in chapter 2, there are some aspects of the language to understand.

1.1 Using PetriScript

To launch PetriScript in Macao, select AMI-NET > Modelling facilities > Petri Net assembling > Execute list of commands. Then Macao opens a window, shown in figure 1.1.

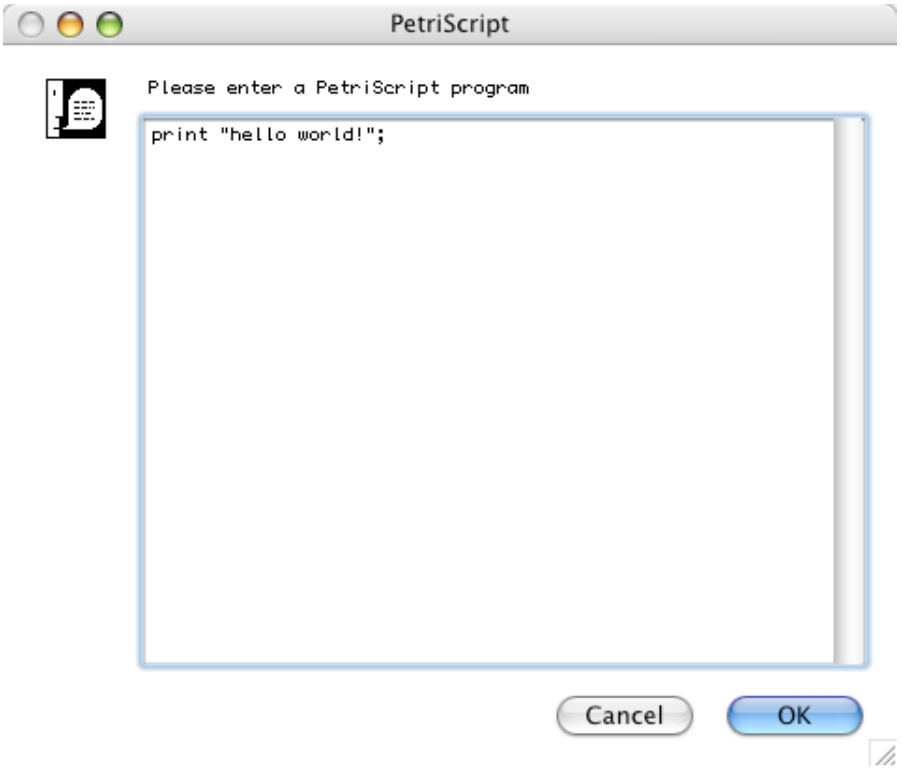

Figure 1.1: PetriScript window

It is strongly advised to type the program in a real text editor like Emacs[1], or at least

[1]PetriScript provides you a mode for Emacs

to copy your script before validating in order to keep the text in case of an error.

You can also choose to launch PetriScript scripts in a debug mode by selecting AMI-NET > Modelling Facilites > Petri Net assembling > Execute list of commands (debug mode). This time, if an error occurs, then Macao will display state in which state the net just was before the error happened.

1.2 Generalities

Typically, PetriScript consists in sequence of declarations followed by a sequence of instructions. It's also possible to define some macros anywhere in the text. Declarations, instructions and macros definitions all end with the symbol ';'. There can be as many spaces and carriage return as you wish. Like Ada, it is verbose, but unlike Ada, it is case-sensitive to preserve names and attributes of nodes.

1.3 Comments

Comments begin with '--' and finish with a carriage return.

1.4 The basic instruction: print

It prints the following string expression (see 1.8) into the historic window of Macao. So, a first very simple script would be like this:

```
print "Hello world!";
```

1.5 Macros

PetriScript uses m4[2] as its preprocessor. So a macro is defined this way:

```
define(macro,text)
```

It has the effect to replace all 'macro' which follow this definition in the script by 'text'. So the tiny following scripts produce "*Hello World!*" on the historic window of Macao:

```
define(HELLO,Hello World!)
print "HELLO";

define(HELLO,"Hello World!")
print HELLO;
```

You can use any facilities of m4 such as testing if a macro is defined. For more details, take a look at the m4 documentation.

1.6 Variables types

Three different types are available in PetriScript : integers, lists and strings. Names of variables which hold these types always begin with the symbol '$' and can contain numerical characters, letters and the underscore symbol '_'. It is completely legal to write $a_very_long_variable_name_993023.

[2] http://www.gnu.org/software/m4/m4.html

Variables can be affected with a value or with another variable of the same type at declaration or anywhere in the script using ':='.

Declarations are made at the beginning of the script before any instruction. Type is given before the variable name and an affectation if needed.

Example:

```
int $i;
string $chaine := "a little text";
```

- Integers
  They are declared by the keyword **int**.

  ```
  int $i;
  int $j := 10;
  ```

  Any arithmetic expression (see 1.7) can be affected to an integer variable.

  ```
  $i := 1 + 10 mod 2;
  $j := $i / 3;
  ```

- Lists
  They are declared by the keyword **list**.

  ```
  list $l;
  list $m := place {"a" , "b"};
  ```

  Elements of the list are separated by commas and surrounded by brackets.
  As their behaviour is completely related to the manipulation of nodes, the exact way of using these lists is described further on, at section 2.1.2, p. 11.

- Strings
  They are declared by the keyword **string**.

  ```
  string $s;
  string $t := "some text to print";
  ```

  Any string expression (see 1.8) can be affected to a string variable.

  ```
  $s := "text" & '10/2';
  $t := '$s' & "an other text";
  ```

1.7 Arithmetic expressions

PetriScript handles arithmetical operations on integers. The available operations are:

- multiplication '$*$'
- division '/'
- modulo '**mod**'
- subtraction '$-$'
- addition '+'

Operands of an arithmetic expression can be integers or variables.

As usual, multiplication, division and modulo operators have the same priority. They have priority over addition and subtraction. Obviously, if you have to deal with complex arithmetic expressions, you can use parentheses.

```
$i := 1+2*3;
$i := $i mod 3 * ( 2 / 5 );
```

If a list is given in an arithmetic expression, it returns its size. It is not legal to give a string expression.

1.8 String expressions

String expressions are used by the **print** instruction and to describe nodes (explained in chapter 2).

A string expression consists in the concatenation of different operands with the operator '&'. An operand can be a

- sequence of characters surrounded by '"' :

  ```
  "Some text"
  ```

- variable surrounded by ''' :

  ```
  '$variable'
  ```

  If the given variable is a string, it just returns its content; when it is an integer, it returns its value interpolated into a string. Finally, if the variable is a list, it returns a string representation of all its components.

- arithmetic expression surrounded by ''' :

  ```
  '10 / 5 +3'
  '2 + $i'
  ```

So, the following script will produce "*Result: 5 + 2*" on the historic window.

```
int $i;
string $s;

$i := 10 / 2;
$s := "Result: " & '$i' & " + " & '10 mod 8';

print '$s';
```

To insert a carriage return in a string expression, use the '/' character: "a line / another line"

1.9 Control structures

PetriScript provides the classic ways of controlling your scripts with conditional and loop structures.

1.9.1 Conditional structures

if *boolean_expression* ***then***
    instructions_sequence
end if

if *boolean_expression* ***then***
    instructions_sequence
else
    instructions_sequence
end if

A *boolean_expression* is composed of arithmetic expressions connected with boolean operators, showed in table 1.1.

Symbol	Comparison
=	Equal
/=	Different
<=	Inferior or equal
>=	Superior or equal
<	Inferior
>	Superior

Table 1.1: Boolean Operators

A boolean expression can be composed of two boolean expressions connected with **and** and **or**. It can also be negated with **not**.

Examples:

```
if $i = 0 then
        print "zero value";
end if;

if $i = 0 and $j = 1 then
        print "zero value";
else
        print "other value";
end if;

if not $i = 0 and $j = 1 then
        print "test passed";
end if;
```

You can also test if a value is in a given range with the boolean expression *arithmetic expression* ***in*** *arithmetic expression* **..** *arithmetic expression*.

Examples:

```
if $i in 0..10 then
        print "good";
end if;
```

```
if $i in 0..10 or $i = 20 then
        print "always good";
end if;
```

1.9.2 Loop structures

You can use two types of structures to loop, one which loops while the given boolean expression is verified, and another one which loops for a given range.

- **While** structure

__while__ boolean_expression __loop__
    instructions_sequence
__end loop__

Example:

```
while $i < 10 loop
        $i := $i + 1;
end loop;
```

- **For** structure

__for__ variable __in__ arithmetic_expression__..__arithmetic_expression __loop__
    instructions_sequence
__end loop__

Example:

```
for $i in 1..10 loop
        print '$i';
end loop;
```

1.10 Basic example

Here is a little script which presents every basic aspect of PetriScript :

```
-- a little example
-- which tests the parity of
-- a number

define(EVEN," even. ")
define(ODD," odd. ")

int $i;
string $s;

for $i in 1..10 loop
        if $i mod 2 = 0 then
```

```
                $s := '$s' & '$i' & EVEN;
        else
                $s := '$s' & '$i' & ODD;
        end if;
end loop;

print "Result: " & '$s';
```

It displays "*Result: 1 odd. 2 even. 3 odd. 4 even. 5 odd. 6 even. 7 odd. 8 even. 9 odd. 10 even.*" on the historic window.

Chapter 2

Manipulate Petri Nets

This chapter explains how to manipulate Petri nets with PetriScript.

2.1 Describe a node

Before acting on Petri nets , it is necessary to describe its components: nodes and arcs.

2.1.1 Single node

A node is designated by its name, which is a string expression and its type. The type of a node can be a **place**, a **queue**, an **immediate** or a **transition**. The form to designate a node is the following: *type node_name*. For example, the place *a* is simply described as

```
place "a"
```

When creating a node (see 2.2.1), you may want to give some attributes of the node to create. You can give a list of attributes: *type node_name attributes_list* Like the name of a node, most of these attributes are string expressions. The table 2.1 shows which attributes can be used with a type of node.

	place	queue	transition	immediate	type
name	*	*	*	*	string
x	*	*	*	*	arithmetic
y	*	*	*	*	arithmetic
r	*	*	*	*	arithmetic
t	*	*	*	*	arithmetic
domain	*	*			string
marking	*	*			string
guard			*	*	string
priority			*	*	string
delay			*		string
action			*		string
weight				*	string

Table 2.1: Attributes

Therefore, to describe a place named *place* with a marking equals to 1, it's done this way:

```
place "place" marking "1"
```

or

```
place "place" (marking "1")
```

If there are more attributes to write, the parentheses are mandatory and attributes are separated by a comma:

```
place "place" (marking "1", domain "red" )
```

As PetriScript supposes that each node has a unique name, it is sufficient to designate it with its name when modifying it, but you can type the whole description if you want to. If one attribute is wrong even if the name is good, then PetriScript will complain about not finding the node.

The attribute **name** is only useful when modifying a node (see 2.2.4). Note that writing **place** "place" (**name** "another name", **marking** "2") is strictly equivalent to **place** "another name"**marking** "2".

The Net node

The Net node is a particular node which has two attributes: **authors** and **declaration**. Only one operation can be made on this special node: the modification. For more details, see subsection 2.2.4, page 16.

Graphical positioning

The **x** and **y** attributes are the cartesian coordinates of a node in the Macao window, while **r** and **t** are its polar coordinates. When using these latter attributes, PetriScript uses **x** and **y** as the center of the orthonormal coordinate system in which polar coordinates are used, as shown in figure 2.1.
The upper left corner of the Macao window is used as the center of the nodes coordinates, the x axis is oriented from the left to the right and the y axis from the top to the bottom.

2.1.2 Lists of nodes

Lists are useful for fusions of sets of nodes. Before using one, it is necessary to declare it as described in 1.6, p. 4.
A such list is a sequence of nodes of the same type separated with a comma and included within '{' and '}'. The type is specified before the list:

```
place {"p_1" , "p_2" , "p_3"}
immediate {"i_11" , "i_12" , "i_13","i_14"}
```

Adding nodes

Here are the four ways to add nodes to an existing list:

- *variable* **:=** *type node_list*
  The content of the *variable* is lost (but the nodes it formerly contained are not deleted) and replaced with the given list. If the two given lists are of different types, *variable* takes the type of *node_list*.

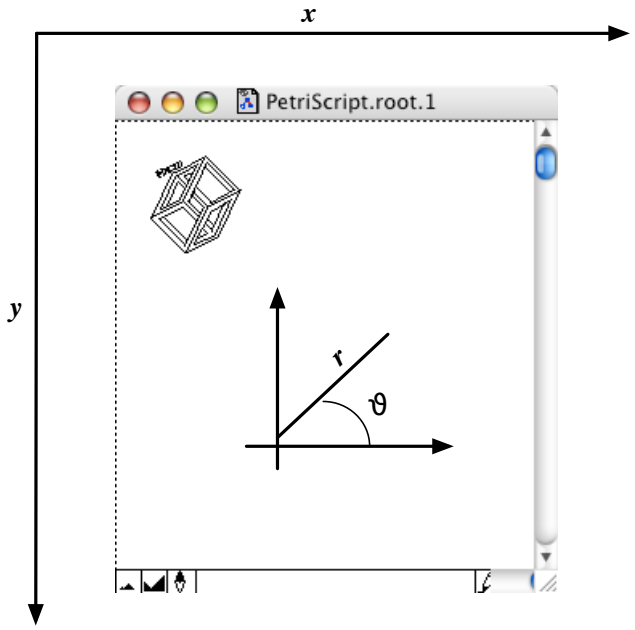

Figure 2.1: Cartesians and polar coordinates in PetriScript

```
$l := place {"p1", "p2"};
$m := transition {"t1","t2","t3"};
```

- ***append*** *node* ***to*** *variable*
  It simply adds the node *node* at the end of variable *variable*. If *node* and *variable* are of different types, an error is produced.

```
append place "p" to $l;
append transition "t" to $m;
```

- ***append*** *type node_list* ***to*** *variable*
  If *node_list* and *variable* are of different types, an error is produced.

```
append place {"p1","p2"} to $l;
append transition {"t1","t2","t3"};
```

- ***append*** *variable* ***to*** *variable*
  If *variable* and *variable* are of different types, an error is produced.

```
append $n to $l;
```

Note 1 You can't add a node that doesn't exist to a list. Otherwise, it will produce an error.

Note 2 If you add the same node two or more times to a list, only its first occurrence will be in the list.

Accessing node in list

Accessing to an element in a list is done by giving its position between brackets: *variable[position]*, where *position* is an arithmetic expression. It is so possible to designate a node via the list containing it. So it is possible to write things like this:

```
append $l[2] to $m;
```

Trying to access an element at a position which is beyond the list's size does nothing and doesn't produce an error. The first element is at position 0 and the last element is at position $\$l - 1$ for a list \$l. A list in an arithmetic expression returns its size.

As an example, the following script creates some places, then adds them to a list and finally prints this list element by element on the historic window.

```
list $l;
list $m;
int $i;

for $i in 1..3 loop
        create place "p" & '$i';
        create place "q" & '$i';
end loop;

create place "r";

$l := place {"p1","p2"};
$m := place {"q1", "q2","q3"};

append $m to $l;
append place "r" to $l;
for $i in 0..$l-1 loop
        print "l" & '$i' & " => " & '$l[$i]';
end loop;
```

List inter-dependencies

To prevent the presence of a non-existing node in a list, if a node is contained in more than one list, deleting or modifying it will also delete or modify it in every other list wich has a reference on it.

Regular Expressions

An important feature of PetriScript is that you can add nodes with regular expressions in lists. In fact, anywhere a list is expected, you can use these expressions. These expressions are surrounded by '%' and are the same as in Perl or Python. They return all nodes that match the given expression. For example, if they are three places *a*, *b* , *c* in a Petri net, the list \$l will contain place *a* and *b* :

```
list $l;
$l := place {%[a-b]%};
```

You can use these regular expressions for every attribute: if now places *b* and *c* have the same domain *d*, you can select them with the following script:

```
list $l;
$l := place {%.% domain %d%};
```

You can use these every time you need to construct a list, as shown in the following example which adds all places that contain "places" in their names to the list $l:

```
int $i :=0;
list $l;

for $i in 1..10 loop
        create place "place" & '$i';
end loop;

append place {%place%} to $l;
```

Note The regular expression like *ab* will match not only the text *ab*, but also the text *abba*. So if you want to match an exact word, you can use *^ab$* where '^' means the beginning of the text in which the search is done and '$' its end. You can easily find documentation on internet on this subject.

2.1.3 Arcs

They are simply designated this way: *(node1 , node2)*, which is the arc oriented from *node1* to *node2*. For example:

```
(place "p",transition "t")
```

If two nodes of the same type are provided, an error is raised.

2.2 Available operations

For each operation, if it is given a node that doesn't exist, the script fails. Creating a node or modifying a node with the characteristics of an existing one also produces an error.

2.2.1 Creation

create *node*

```
create place "p";
create place "q" ( domain "red", marking "1");
```

The following example shows how to create a set of places (added to a list) and transitions, using string expressions:

```
int $i;
list $l;

for $i in 1..10 loop
        create place "place_" & '$i' ( domain "green", marking
            '$i');
        append place "place_" & '$i' to $l;
        create transition "transition_" &  '$i' ( guard '$i-1')
            ;
end loop;

print '$l';
```

2.2.2 Connection

connect *valuation node1* ***to*** *node2*
connect *node1* ***to*** *node2*
connect inhibitor *valuation node1* ***to*** *node2*
connect inhibitor *node1* ***to*** *node2*

As Petri nets are oriented graphs, the resultant arc will be oriented from *node1* to *node2*. To make it inhibited, use the keyword **inhibitor**. *Valuation* is a string expression.

```
connect place "p" to transition "t";
connect transition "t" to place "p";
connect "<x>" place "p" to transition "t";
connect inhibitor transition "t" to place "p";
```

2.2.3 Deletion

delete *node*
delete *list*
delete *arc*

You can delete a sole node or a list of node. Delete an arc has the effect to disconnect the two given nodes, but it does not delete them.

```
delete place "place";
delete $l;
delete( place "p" , transition "t");
delete $l[$i+1];
```

Remember that

```
delete place "place";
```

is strictly equivalent to

```
delete place "place" (domain "red", marking "1");
```

as PetriScript differentiates nodes only with theirs names.

2.2.4 Modification

set *node* ***to*** *attributes_list*
set *arc* ***to*** *string_expression*

You can modify every attributes of a node. To rename a node, use the attribute **name**.

```
set place "p" to name "q";
set transition "t" to ( name "u", guard "3");
set place "q" to ( x 100 , y 10 );
set $l[3] to domain "blue";
```

Modifying an arc changes its valuation.

```
set ( place "p" , transition "t" ) to "<y>";
```

It's not possible to change a classic arc to an inhibited one, or the opposite. If you want to, delete the old arc and create a new one with the type you want.

The Net node

To modify the Net node, you would write:

```
set net "net" to (authors "a" , declaration "d");
```

But as there is only one net node, it's not mandatory to write its name. So you can simply write:

```
set net to (authors "a" , declaration "d");
```

2.2.5 Fusion

Four types of fusion are provided: a node with a node, a list with a node, a list with a list and a list into a single node. Trying to merge a node with itself produces an error. In all cases of fusion, the given node or lists are deleted.

Node with node

merge *node1* ***and*** *node2* ***into*** *attributes_list*
merge *node1* ***and*** *node2*

If no *attributes_list* is given, then PetriScript automatically computes the new name and attributes by concatenating *node1* and *node2* with an underscore. For example merging a place *a* with a marking *m* and a place *b* gives a place with a name *a_b* and a marking *b*.

Examples:

```
merge place "a" and place "b";
merge place "c" and place "d" into name "e" ;
merge place "c" and place "d" into (name "e" , marking "f);
```

Node with List

merge *node1* ***and*** *list* ***into*** *attributes_list*
merge *node1* ***and*** *list*
merge *list* ***and*** *node1* ***into*** *attributes_list*
merge *list* ***and*** *node1*

This is the extension of the previous fusion type. *Node1* is duplicated as many times as there are nodes in *list*. Then each copy of *node1* is merged with a node of *list*. This time, *attributes_list* is used as a pattern: each new resultant node will be created with the characteristics contained in *attributes_list* concatenated with a number starting from 1. If no pattern is given, PetriScript computes one itself.

As an example, the following script creates a place "a" and five places prefixed by "b_" in list $l. Then it merges this list with the place "a", giving a marking equal to 1 to the newly created nodes.

```
int $i;
list $l;

create place "a";

for $i in 1..5 loop
```

```
        create place "b_" & '$i';
end loop;
$l := place {%b_*%};

merge $l and place "a" into (marking "1");
```

List with list

merge *list1* ***and*** *list2* ***into*** *attributes_list*
merge *list1* ***and*** *list2*

Again, this is an extension to the previous fusion type. Now each node of *list1* is duplicated as many times as there are nodes in *list2* and reciprocally. Then each copy of *list1* is merged with the corresponding copy of *list2*, as shown in figure 2.4.

The following script creates two types of places: the ones which are prefixed by "a_" and the ones which are prefixed by "b_". It adds them into lists $l and $m, then merges these lists.

```
int $i;
list $l;
list $m;

for $i in 1..5 loop
        create place "a_" & '$i';
end loop;
$l := place {%a_*%};

for $i in 1..3 loop
        create place "b_" & '$i';
        append place "b_" & '$i' to $m;
end loop;

merge $l and $m;
```

Single list

merge *list* ***into*** *attributes_list*
merge *list*

Now, all nodes of *list* are merged into a single one, with the attributes of *attributes_list* if presents, or computed automatically if not.

Examples:

```
merge $l into name "another_place";
merge place {"a1","a2","a3"};
```

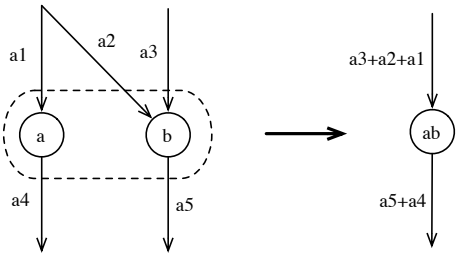

Figure 2.2: Nodes fusion

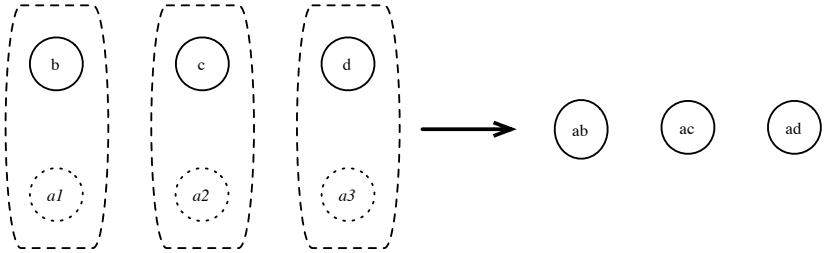

Figure 2.3: Fusion of a node a with a list $\$E = \{b, c, d\}$

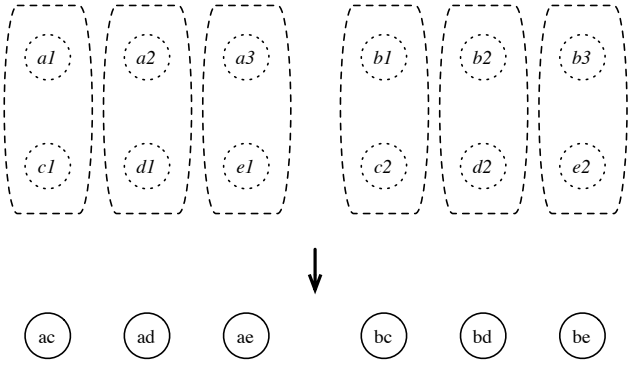

Figure 2.4: Fusion of a list $\$E_1 = \{a, b\}$ with a list $\$E_2 = \{c, d, e\}$

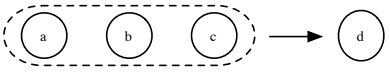

Figure 2.5: Fusion of a single list $\$E1 = \{a, b, c\}$ into a node c

Chapter 3

Examples

Here are some more realistic examples:

3.1 FIFO

The following script creates a fifo (shown in figure 3.1) using macros and looping instructions.

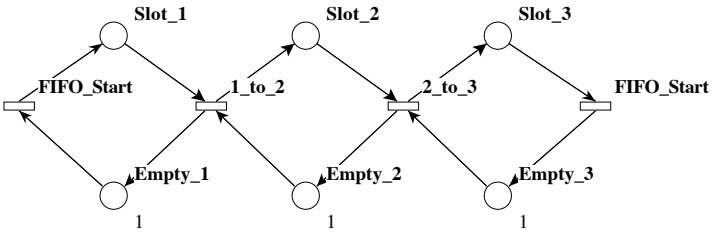

Figure 3.1: FIFO

```
define(FIFO_SIZE,3)
define(FIFO_BASE_X,100)
define(FIFO_BASE_Y,100)
define(FIFO_STEP,120)

int $wave := 0;

for $wave in 1..FIFO_SIZE loop
        create place "Slot_" & '$wave' (x FIFO_BASE_X + FIFO_STEP * $wave,
                y FIFO_BASE_Y);
        create place "Empty_" & '$wave' (x FIFO_BASE_X + FIFO_STEP * $wave,
                y FIFO_BASE_Y + 100, marking "1");
end loop;
for $wave in 1..FIFO_SIZE+1 loop
        create transition "t" & '$wave -1' & "_to_" & '$wave' (x FIFO_BASE_X + FIFO_STEP
            * $wave - FIFO_STEP / 2,
                y FIFO_BASE_Y + 50);
        if $wave < FIFO_SIZE+1 then
                connect "1" transition "t" &'$wave -1' & "_to_" & '$wave' to  place "
                    Slot_" & '$wave';
                connect "1" place "Empty_" & '$wave' to transition "t" &'$wave -1' & "
                    _to_" & '$wave';
        end if;
        if $wave > 1 then
                connect "1" transition "t" &'$wave -1' & "_to_" & '$wave' to  place "
                    Empty_" & '$wave - 1';
```

```
                connect "1" place "Slot_" & '$wave - 1' to transition "t" &'$wave -1' & "
                    _to_" & '$wave';
        end if;
end loop;

set transition "t0_to_1" to (name "FIFO_Start");
set transition "t" &  'FIFO_SIZE' & "_to_" & 'FIFO_SIZE + 1' to (name "FIFO_End");
```

3.2 Philosophers' dinner

This is a coloured version of the Philosophers' dinner, shown in figure 3.2.

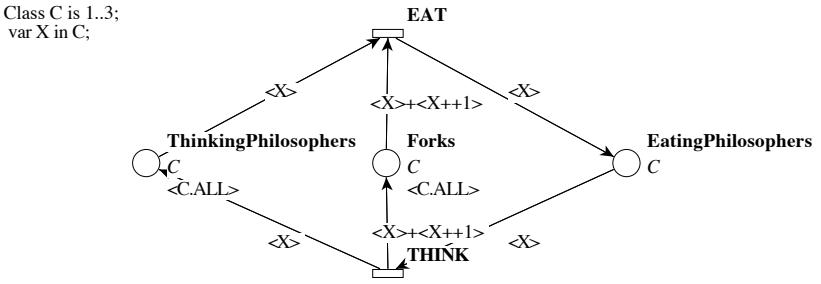

Figure 3.2: Philosophers' dinner

```
define(THK, ThinkingPhilosophers)
define(PHILOS,3)
define(EATING, EatingPhilosophers)
define(XPOS,100)
define(YPOS,100)
define(XINCR,150)
define(YINCR,75)

-- declare our variables in the net
set net to declaration "Class C is 1..PHILOS; / var X in C;";

create place "THK" ( domain "C" , marking "<C.ALL>", x XPOS , y YPOS);
create transition "EAT" ( x XPOS + XINCR , y YPOS - YINCR);
create place "EATING" ( domain "C" , x XPOS + 2 * XINCR , y YPOS) ;
create transition "THINK" ( x XPOS + XINCR , y YPOS + YINCR);
create place "Forks" ( domain "C" , marking "<C.ALL>" ,  x XPOS +  XINCR , y YPOS);

connect "<X>" place "THK" to transition "EAT";
connect "<X>" transition "EAT" to place "EATING";
connect "<X>+<X++1>" place "Forks" to transition "EAT";
connect "<X>" place "EATING" to transition "THINK";
connect "<X>+<X++1>" transition "THINK" to place "Forks";
connect "<X>" transition "THINK" to place "THK";
```

3.3 Another Philosophers' dinner

Much more complex to write, here is the non-coloured version of the philosophers' dinner problem. In fact, the script complexity is due to the graphical positioning of nodes (see figure 3.3).

```
define(X,500)
define(Y,500)
define(RADIUS,80)
```

```
define(STEP,40)

define(PHILOS,3)

define(ANGLE,360/PHILOS)
define(ADJUST,70)

int $i := 0;

for $i in 1..PHILOS loop

        create place "THINK_" & '$i' (marking "1" , x X , y Y, r RADIUS + 6*STEP, t ANGLE
            *$i);
        create transition "TAKE_LEFT_1_FORK_" & '$i' (x X , y Y, r RADIUS + 5*STEP , t
            ANGLE*$i-ADJUST/5);
        create transition "TAKE_RIGHT_1_FORK_" & '$i' (x X , y Y , r RADIUS + 5*STEP , t
            ANGLE * $i+ADJUST/5);
        create place "WAIT_RIGHT_FORK_" & '$i' (x X , y Y, r RADIUS + 4*STEP, t ANGLE*$i-
            ADJUST/4);
        create place "WAIT_LEFT_FORK_" & '$i' (x X , y Y, r RADIUS + 4*STEP, t ANGLE*$i+
            ADJUST/4);
        create transition "TAKE_LEFT_2_FORK_" & '$i' (x X, y Y, r RADIUS + 3*STEP , t
            ANGLE*$i+ADJUST/3);
        create transition "TAKE_RIGHT_2_FORK_" & '$i' (x X, y Y, r RADIUS + 3*STEP, t
            ANGLE*$i-ADJUST/3);
        create place "EAT_" & '$i' ( x X, y Y, r RADIUS + 2*STEP, t ANGLE*$i);
        create transition "RELEASE_FORK_" & '$i' (x X , y Y, r RADIUS + STEP, t  ANGLE*$i
            );

        connect place "THINK_" & '$i' to transition "TAKE_LEFT_1_FORK_" & '$i';
        connect place "THINK_" & '$i' to transition "TAKE_RIGHT_1_FORK_" & '$i';
        connect transition "TAKE_LEFT_1_FORK_" & '$i' to place "WAIT_RIGHT_FORK_" & '$i';
        connect transition "TAKE_RIGHT_1_FORK_" & '$i' to place "WAIT_LEFT_FORK_" & '$i';
        connect place "WAIT_RIGHT_FORK_" & '$i' to transition "TAKE_RIGHT_2_FORK_" & '$i
            ';
        connect place "WAIT_LEFT_FORK_" & '$i' to transition "TAKE_LEFT_2_FORK_" & '$i';
        connect transition "TAKE_RIGHT_2_FORK_" & '$i' to place "EAT_" & '$i';
        connect transition "TAKE_LEFT_2_FORK_" & '$i' to place "EAT_" & '$i';
        connect place "EAT_" & '$i' to transition "RELEASE_FORK_" & '$i';

        create place "FORK_LEFT_" & '$i'( x  X, y Y, r RADIUS, t ANGLE*$i - ADJUST);
        create place "FORK_RIGHT_" & '$i' (x  X, y Y, r RADIUS, t ANGLE*$i + ADJUST);

        connect transition "RELEASE_FORK_" & '$i' to place "FORK_LEFT_" & '$i';
        connect transition "RELEASE_FORK_" & '$i' to place "FORK_RIGHT_" & '$i';
        connect transition "RELEASE_FORK_" & '$i' to place "THINK_" & '$i';
        connect place "FORK_LEFT_" & '$i' to  transition "TAKE_LEFT_1_FORK_" & '$i';
        connect place "FORK_RIGHT_" & '$i' to  transition "TAKE_RIGHT_1_FORK_" & '$i';
        connect place "FORK_LEFT_" & '$i' to  transition "TAKE_LEFT_2_FORK_" & '$i';
        connect place "FORK_RIGHT_" & '$i' to  transition "TAKE_RIGHT_2_FORK_" & '$i';

end loop;

for $i in 1..PHILOS loop

if $i = PHILOS then
        merge place "FORK_LEFT_" & 'PHILOS' and place "FORK_RIGHT_1" into (name "FORK_" &
             '$i',marking "1");
else
        merge place "FORK_LEFT_" & '$i' and place "FORK_RIGHT_" & '$i+1' into (name "
            FORK_" & '$i', marking "1");
end if;

end loop;
```

3.4 Trains

Two trains circulate in the same direction on a circular railroad, divided in fifteen sections. The two trains can never, for security reasons, be on two contiguous segments. Traffic lights manage the access to each of these sections. The Petri net shown at figure 3.4, model this problem: sections are represented by places Section_1 to Section_15. The presence of a marking into of this place means that a train is present at this location. Traffic lights are modelled by places F1 to F15. The presence of the marking indicate that the light is green, the entry in the section it is guarding is then possible. The passage from a section x to a section y is done at the activation of the transition x to y.

It is a good example of the automatization made possible by PetriScript: it is sufficient to modify the macro SECTIONS to obtain a bigger net, without having to use the graphical interface.

```
define(X,250)
define(Y,350)
define(radius,50)
define(R,150)

define(SECTIONS,15)

define(INNER_ANGLE,360/SECTIONS)
define(OUTTER_ANGLE,360/(2*SECTIONS))

int $i := 0;
int $j := 0;

for $i in 1.. SECTIONS loop
        create place "F" & '$i' ( x X, y Y, r radius, t $i * INNER_ANGLE);
        create place "Section_" & '$i' ( x X, y Y,  r R, t $i * INNER_ANGLE);
        create transition "t" & '$i' & "_to_" & '$i mod SECTIONS + 1' ( x X, y Y, r R, t
            $i * INNER_ANGLE + OUTTER_ANGLE);
end loop;


for $i in 1.. SECTIONS loop
        connect place "Section_" & '$i'  to transition "t"&'$i' & "_to_" & '$i mod
            SECTIONS + 1';

        connect transition "t" & '$i' & "_to_" & '$i mod SECTIONS + 1' to place "Section_
            " & '$i mod SECTIONS + 1';

        if $i /= 1 then
                connect place "F" & '$i'  to transition "t" & '$i-1' & "_to_" & '$i';
        else
                connect place "F1"  to transition "t" & 'SECTIONS' & "_to_" & '1';
        end if;

        connect transition "t" &'$i mod SECTIONS + 1' & "_to_" & '($i+1) mod SECTIONS +
            1' to place "F" & '$i';

end loop;

for $i in 1.. SECTIONS loop
        if $i mod 3 = 0 then
                set place "Section_" & '$i' to marking "1";
        else
                set place "F" & '$i' to marking "1";
        end if;
end loop;
```

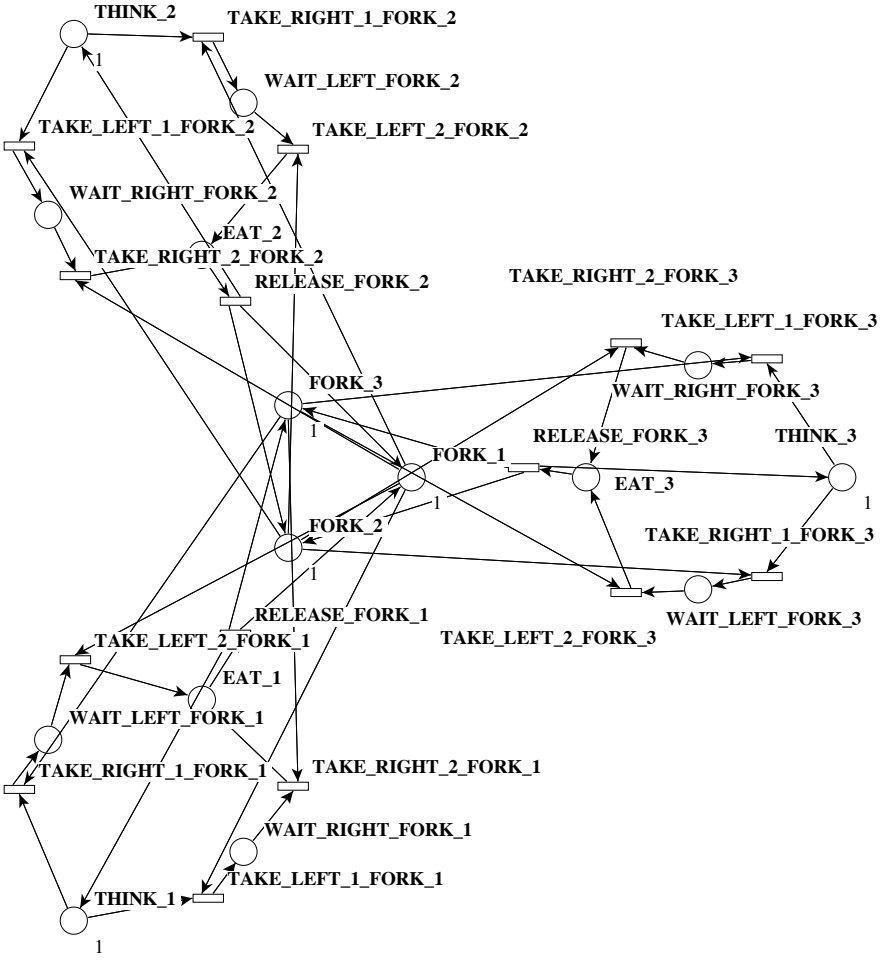

Figure 3.3: Philosopher's dinner

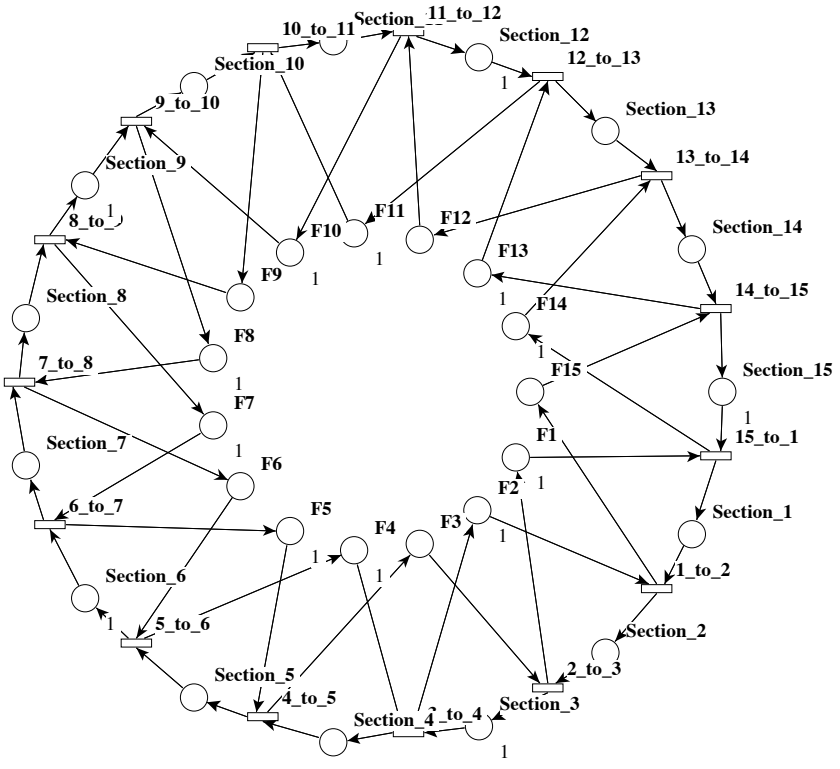

Figure 3.4: Trains

Appendix A

Backus-Naur Form

*Keywords are in **bold**.*

A few definitions

```
char            ::=     ASCII characters,except formatting characters

integer         ::=     [0-9]+

chars           ::=     sequence of char

identifier      ::=     '$'chars
```

Expressions used by instructions and node descriptions

```
arithmetic_operator     ::=     '*'
                        |       '/'
                        |       mod
                        |       '-'
                        |       '+'

arithmetic_expression   ::=     '('arithmetic_expression')'

                        |       arithmetic_expression
                                        arithmetic_operator
                                arithmetic_expression

                        |       integer

string_expression       ::=     str_exp_comp
                        |       str_exp_comp '&' str_exp

str_exp_comp            ::=     '"'chars'"'
                        |       '''identifier'''
                        |       '''arithmetic_expression'''

boolean_relator         ::=     '<'
                        |       '>'
                        |       '<='
                        |       '>='
                        |       '='
                        |       '/='

boolean_operator        ::=     or
                        |       and

boolean_expression      ::=     boolean_relation
                                        boolean_operator
```

```
                        boolean_relation

                |       not boolean_expression
                |       '(' boolean_expression ')'
                |       boolean_relation


boolean_relation        ::=     arithmetic_expression
                                        boolean_relator
                                arithmetic_expression

                |       arithmetic_expression
                                        in
                                arithmetic_epxression
                                        '..'
                                arithmetic_expression


regexp                  ::=     '%' chars '%'
```

Node description

```
node_type               ::=     place
                        |       queue
                        |       transition
                        |       immediate
                        |       net

attribute               ::=     name
                        |       x
                        |       y
                        |       r
                        |       t
                        |       domain
                        |       marking
                        |       guard
                        |       priority
                        |       delay
                        |       action
                        |       weight
                        |       authors
                        |       declaration

attribute_expression    ::=     string_expression
                        |       arithmetic_expression

node                    ::=     node_type node_description
                        |       list_access

node_description        ::=     string_expression
                        |       string_expression attribute attribute_expression
                        |       string_expression '(' attributes_list ')'

attributes_list         ::=     attribute attribute_expression
                        |       attribute attribute_expression ',' attributes_list
```

List of nodes

```
node_list       ::=     node_type '{' nodes_list '}'

nodes_list      ::=     list_component
                |       list_component ',' nodes_list

list_component  ::=     node_description
                |       node_regexp

node_regexp     ::=     regexp
```

```
                |       regexp attribute regexp
                |       regexp '('list_regexp')'

list_regexp     ::=     attribute regexp
                |       attribute regexp','list_regexp

list_access     ::=     identifier'['arithmetic_expression']'
```

Arcs

```
arc             ::=     '('node,node')'
```

Main body

```
body            ::=     preprocess
                |       declarations
                |       instructions

instructions    ::=     instruction ';'
                |       instruction ';' instructions

instruction     ::=     conditional
                |       looping

                |       printing
                |       appending
                |       affectation

                |       creation
                |       connection
                |       deletion
                |       fusion
                |       modification
```

Macros

For more informations on the preprocessor, please take a look at the m4 documentation.

```
preprocess      ::=     define'('chars','chars')'
```

Declarations

```
type            ::=     int
                |       string
                |       list

declarations    ::=     type identifier ':=' string_expression
                |       type identifier ':=' arithmetic_expression
                |       type identifier ':=' node_list
                |       type identifier
```

Conditional structures

```
conditional     ::=     if bool_exp then
                                instructions
                        end if ';'

                |       if bool_exp then
                                instructions
                        else
                                instructions
                        end if ';'

looping         ::=     while bool_exp then
                                instructions
```

```
                end loop ';'

        |       for identifier in ar_exp '..' ar_exp loop
                        instructions
                end loop ';'
```

Various instructions

```
printing        ::=     print string_expression

affectation     ::=     identifier ':=' identifier
                |       identifier ':=' string_expression
                |       identifier ':=' arithmetic_expression
                |       identifier ':=' node_list

appending       ::=     append node to identifier
                |       append node_list to identifier
                |       append identifier to identifier
```

Instructions to manipulate nodes

```
creation        ::=     create node

connection      ::=     connect inhibitor string_expression node to node
                |       connect inhibitor node to node
                |       connect string_expression node to node
                |       connect node to node

deletion        ::=     delete node
                |       delete node_list
                |       delete identifier
                |       delete arc

modification    ::=     set node to '('attributes_list')'
                |       set net to '('attributes_list')'
                |       set arc to string_expression

fusion          ::=     merge fusion_comp and fusion_comp into attributes_list
                |       merge fusion_comp and fusion_comp
                |       merge identifier into attributes_list
                |       merge node_list into attributes_list
                |       merge identifier
                |       merge node_list

fusion_comp     ::=     node
                |       identifier
                |       node_list
```

Index