# 9

# Visualizing Program Semantics

Guofu Zhou and Zhuomin Du
*Wuhan University*
*China*

## 1. Introduction

Generally, any program is designed for computing one problem based on one special architecture computing machine. It is unavoidable that some machine constraint will be transferred to the program code. When the program is formalized for being verified or validated the machine constraints will be treated as properties of program .

Turing model,the theory model of program(mainstream ), has two special features: Only one storage tape that determines the changing status is sequent; Only one write-read head that determines the sequence of action operating is serialization; Accordingly, research on program semantics is focus on two areas: the first viewpoint thinks that a program is a set of status. the operating is a procedure of status changing. So, for that the formalization tools describe status. The other one viewpoint thinks that a program is a set of processes. A status is abstracted for understanding the process. So, for that the formalization tools describe the processes, and a status only is the composition of processes by timeline.

One target of formalizing a program is,not for a special machine, to get one general and one abstract specification Bjorner et al. (1987); Spivey (1998). So, one program specification must describe the following basic properties:

- variables and theeir changing;
- the consumptive or the nonconsumptive resources;
- the semantics of operation;
- the control flow in program, not one in computing machine.

For the problems, Petri nets is an ideal theory tool. By both extending and applying Petri nets theory, the semantics of program can be formalized visually Breuer & Lano (1999); Girault & Valk (2002); Jensen et al. (2007).

## 2. Model

Informally, a program is composed of data structure and algorithm Harel (1987); Khedker et al. (2009); Woodcock & Davies (1996). We know data structure is a discrete description of entity attributes based on one special computing machine. For example, the data structure of tree can be described as in figure 1(if the tree only has such attributes)

From figure 1, we can conclude the tree is composed of attributes $right$,$left$ and $data$. But we can't make out any relation among $right$,$left$ and $data$. In fact, the relations are the key to

```
Tree: structure
    Right:tree;
    Left:tree;
    Data:integer;
End Tree.
```

Fig. 1. Tree structure

```
x := 1    o1
y := 1    o2
```

Fig. 2. Segment of algorithm

```
y := 1    o1
x := 1    o2
```

Fig. 3. Another segment of algorithm

understand that is a tree but not any others. So, the discrete description of attributes does not help to understand an entity .

And there is another characteristics in coding a program. For example, there is a segment of algorithm in figure 2, $o_1$ and $o_2$ are labeled as a statement on $x$ and a statement on $y$ respectively(ignore other unconcern processes).

Also, we can design another algorithm(figure 3) to fulfill the same function as in figure 2.

The reason to explain the case is in design phase designer only concerns the result produced by the algorithm, and thinks the algorithm is right if and only if the algorithm can produce an expected result. However, the same result the two algorithms can produce but not the same semantics the two algorithms have.

In a word, when designing an algorithm, the designer transforms some constraints into the algorithm unconsciously, for example, serialization(discussed above). Once the algorithm finished, the additional relations are fixed into the code. So, two aftereffects are forged:

1. Two independent operations will be serialized, a new relation is added; or
2. The two concurrent operations will be serialized and the relation of concurrence is lost.

For convenience, the relation between variables, variable and operation, or operations is called control.

**Asynchronous control**, A-control for short, is a kind of relation between operation and variable that has no directive or indirected causality. A-control has two types: concurrent control and noncorrelation control.

A concurrent control denotes there are one or more common direct or indirect causality sets among operations (variables), e.g., $o$ in figure 4 is a common result of $b_1$, $b_2$ and $b_3$. Dashed line shows there are more than one node, the straight line show there is a directive relation.

A noncorrelation control denotes there is no any common direct or indirect causality among operations (variables) . A noncorrelation control is no necessary in program. However, in a large-scale software system, noncorrelation controls are usual. The noncorrelative variables or operations in a system may be associated for building a new system. For example, in figure 5(1), $\{a_1, a_2, \cdots, a_i\}$ is non correlative with $\{b_1, b_2, \cdots, b_i\}$ in a system, however, in figure

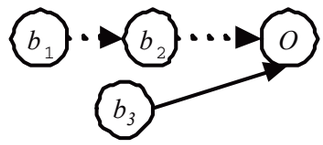



Fig. 4. Concurrent control

5(2), $\{a_1, a_2, \cdots, a_i\}$ is associated with $\{b_1, b_2, \cdots, b_i\}$ because of $c_i$ in a new system. $c_i$ is a common result among $\{a_1, a_2, \cdots, a_i\}$ and $\{b_1, b_2, \cdots, b_i\}$. Obviously, the noncorrelation control between $\{a_1, a_2, \cdots, a_i\}$ and $\{b_1, b_2, \cdots, b_i\}$ disappears.

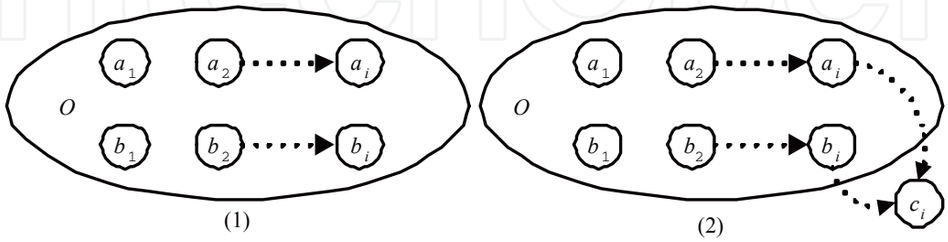



Fig. 5. Asynchronous Control

**Linear control**, L-control for short, describes causality among variables and operations. L-control has two kinds of types: direct associated and indirect associated.

one variable is a direct composition of another one, e.g., $X$ is composed of $b$ as in figure 6(1); or one variable is a indirect sub-composition of another one, e.g., in figure 6(2), $Y$ is directly composed of both $a$ and $b$. Specially, $c$ is an indirect sub-composition of $Y$.


Fig. 6. Linear Variables

For operations, L-control denotes one operation is a run condition of another one , e.g., in figure 7(1), $c_i$ run only after $a_i$ finished (direct associated); or one operation is one of run conditions of another one (indirect associated), e.g., in figure 7(2), $c_i$ can not run immediately after $a_i$ finish if $b_i$ do not finish.


Fig. 7. Linear Operations

*Loop control* is a special L-control(figure 8). because of the recursive attributes not being permitted, A loop control can only describe relations among operations.

**Parallel control** , P-control for short, denotes variables or operations are parallel. On one hand, parallel operations have direct common pre-operations or post-operations as in figure 9(1). $A$ is parallel to $B$ because $C$ is a direct common post-operation of $A$ and $B$.

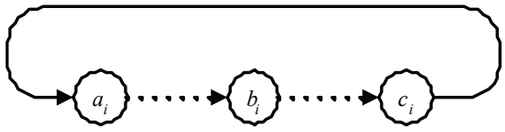
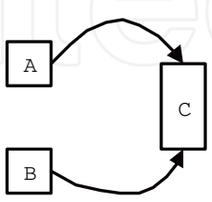



Fig. 8. Linear LOOP

On the other hand, parallel variables have direct common target variables. In figure 9(2), both *right* and $left$ construct a tree, a parallel control exists between $left$ and *right*.



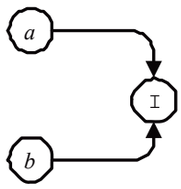

(1)

```
tree:
    left: tree;
    right:tree;
    P-control(left,  right)
```

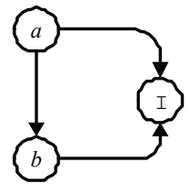
(2)

Fig. 9. Parallel Control

P-control is relative. In figure 10(1), $a$ is parallel to $b$. But, if another P-control is inserted between $a$ and $b$ (figure 10(2)), then the new $a$ is the direct run condition of $b$. Obviously, the new $a$ is not parallel to $b$ and the P-control between $a$ and $b$ losts.




(1)




(2)

Fig. 10. Relative Parallel Control

**Trigger Action** Control is the logic relation that regulates how to run, delay and terminate operations. Any operation can only run after it gets (satisfied) a control handle of computing machine. In other words, computing machine determines how to run a program.

In figure 11, a control handle arrives at the statement *doingsomething*. Here, $controlofprogram = true$ and controls of other statements can not be satisfied ($controlofo_i = false$), the statement *doingsomething* is running. and the other statements do not run. So, Control mechanism can help to describe the dynamic properties.

**Regulate Order** Single control handle determines a program is sequent. If a program has more than one control handles, how about a program should be? For example, a program in figure 11 has one control handle, statements only run in turn. When the program has two control handles, the program can run by many ways, such as in figure 12.

**Resource** is physical and depend on a computing machine. Running a program needs resources, such as RAM, CPU, communication channel, and so on. Generally, resources are managed by an operation system.

$o_1$: *control* of $o_1$ = *false* ;

...

**doing something** : *control* of program = *true* ;

...

$o_2$: *control* of $o_3$ = *false* ;

...

Fig. 11. Trigger Action

| | | | | | |
|---|---|---|---|---|---|
| $o_1$: | contro of o $_1$ = *true* ; | $o_1$: | contro of o $_1$ = *true* ; | $o_1$: | contro of o $_1$ = false ; |
| $o_2$: | contro of o $_2$ = *true* ; | $o_2$: | contro of o $_2$ = *false* ; | $o_2$: | contro of o $_2$ = *true* ; |
| $o_3$: | contro of o $_3$ = *false* ; | $o_3$: | contro of o $_3$ = *true* ; | $o_3$: | contro of o $_3$ = *true* ; |
| | (1) | | (2) | | (3) |

Fig. 12. Regulate Order

Resource can be either consumable or inconsumable . Consumable resource will be consumed out during a program run. Consumable resources are exclusive, such as RAM ,CPU. Inconsumable resource can be shared, such as system clock and any program can read the system clock any time.

We know, the mathematics model of program is Turing machine. Accordingly, a program is a set of variables and read-writes.

- Operation is to process variables, such as to update the current value of variable;
- Variable records the result produced by operation(s). An operation can be described by the track of changing value of variable.

Any operation of program can be broken down a set of more fine grained read-write. Generally, read-write operation appears in assignment statement.

**Assignment** is composed of read and write(not concerns arithmetic). Read refers the current value of variable. Write updates the current value of variable. In an assignment statement, read variable(s) is located in the right side and write variable(s) is located in the left side , such as

$$X := \underline{a + b + c}$$

Where,$a$,$b$,$c$ are read variables. The current values of $a$,$b$ and $c$ are not modified. In the following

$$\underline{X} := a + b + c$$

$X$ is a write variable. The current value of $X$ will be updated.

Read-write not only refers the current value of variable, but also updates the current value. The read-write variable will appear both sides of assignment statement, such as $X$ in the following,

$$\underline{X := X} + c$$

Accordingly, the model of program should be composed of :

1. resources, including consumable resource and nonconsumable resource;
2. variables, recording status;
3. operations, including read and write;

4. relations, including read, write and control.

## 3. Extending definitions

Compare to general Petri net theory Jensen et al. (2007); Reisig (1985), Here concepts are extended.

**Place**

two kinds of places: control place (Figure 13.a) and variable place (Figure 13.b). The definition of control place is same as the definition of place in Petri net. Resources described by variable place are inconsumable. Generally, variable place can describe all data types, such as real type, boolean type, structure type and so on. The graphic representation of variable place is same as that of control place. However, the type of arc connecting with control place is different to that with variable place. Control place can only connect with control arc (flow); Variable place can connect with read arc, write arc or the combinatorial arc of read arc and write arc, but not with control arc. The arc types are discussed in the later this section.

(a) control place (b) variable place

Fig. 13. Extending Place

**Token**

As same as Petri net, token is expressed by symbol "•". Here, token has two functions: firstly, the flowing track of token denotes control flow; secondly, token in a variable place is expressed by the value instead of symbol "•", and the number of tokens in variable place is the current value of variable place, such as in figure 14.b. To hold consistency, the number of token in control place is mapped to a value of integer type. In figure 14.a, the data type denoted by the place is integer and the current value is 1. So, the number of token in place is 1.


(a) (b)

Fig. 14. Extending Token

**Arc**

four kinds of arcs: control arc, read arc, write arc and read-write arc. The graphic representations of arcs are drawn as in figure 15.

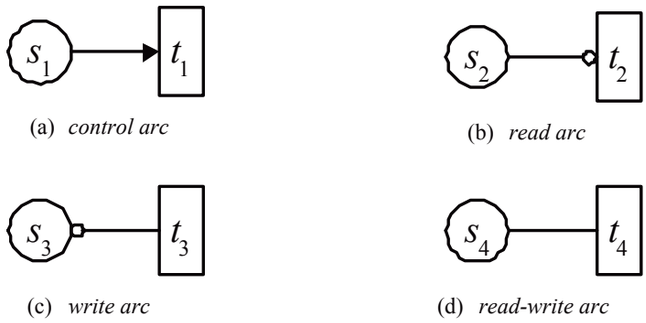
**(a)** represents the relation of control flow which denotes the firing of transition $t_1$ will consume resource $s_1$;

**(b)** represents the read relation which denotes the firing of transition $t_2$ will read the current value of $s_2$ but not modify the current value of $s_2$;

**(c)** represents the write relation which denotes the firing of transition $t_3$ will modify the current value of $s_3$;

**(d)** represents the read-write relation which is the combination of a read relation and a write relation, i.e. the firing of transition $t_4$ not only read the current of $s_4$, but also modifies the current value of $s_4$.

The four kinds of relations are classified into two categories: flow relationship(a) and read-write relationship(b, c, d). Flow relationship is a kind of consumable relationship. Control place can only hold flow relationship(control arc)and tokens in a control place must flow. Read-write relationship is a kind of inconsumable relationship;Read-write do not consume any token and tokens in a variable place can not flow around. Variable place can only hold read-write relationship(read arc, write arc or both).




(a) *control arc* (b) *read arc*

(c) *write arc* (d) *read-write arc*

Fig. 15. Extending Arc

**Transition**

Extension of transition defines a new kind of transition called variable transition. Variable transition can not only connect with control place but also variable place. In other words, the firing of transition may either consume resources(e.g. need some tokens) or not. Contrast to variable transition, transition defined in Petri net is called control transition . The structure of control transition is represented as in figure 16.a. Guard is a mechanism to control the firing of control transition. Control guard holds false only if the firing condition is not satisfied, i.e., the transition can not fire; When the firing condition is satisfied, the control guard hold true and the transition fire. The structure of variable transition is composed of three parts (figure 16.b):

**E** denotes a control guard which is same as control guard of control transition;

**B** is called variable guard which denotes the condition satisfied by variables(places).

**I** operations script .

Because Petri nets is a self-explain system, places need not be declared explicitly. Variables are described by variable places. In figure 17, places are variable places. Assignment $X := a + b + c$ can be described as a variable transition(figure 18), where $a$,$b$ and $c$ connect with the variable transition. Read operation is described by read arc between a variable transition and a variable place. Write operation is described by write arc between a variable transition and a variable place. Figure 20 describes assignment $X := a + b + c$, where Figure 19(1) is read operation. Figure 19 (2) is write operation.

When a variable is both read and write, readwrite relation can be applied. Figure 20 is a readwrite relation. Consumable resource is described by control place. The number of tokens

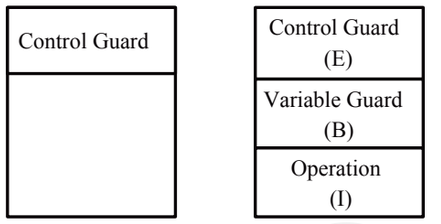



Fig. 16. Extending Transition

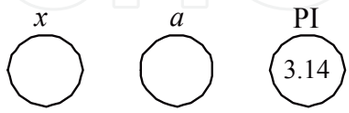



Fig. 17. Variable Place

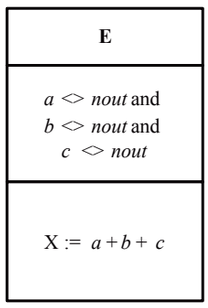



Fig. 18. Inside Transition

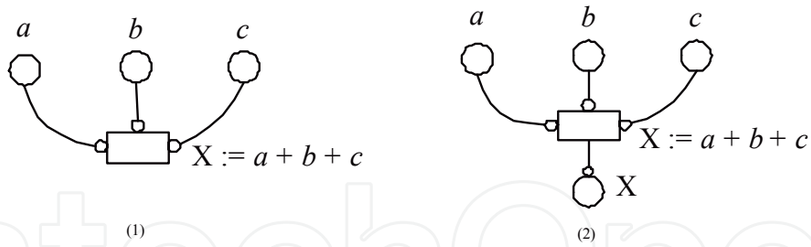



Fig. 19. Read and Write

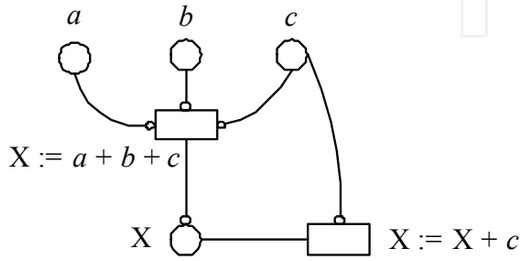



Fig. 20. ReadWrite

in a control place denotes the number of available resource described by the control place. The weight of arc denotes the number of resource consumed by a transition. For example, RAM consumption can be described as in figure **??**. Place denotes the resource RAM; the number of tokens in place denotes the available number of RAM; $o_1$ denotes an operation; the weight of arc denotes that operation $o_1$ will consume two units of RAM during firing.

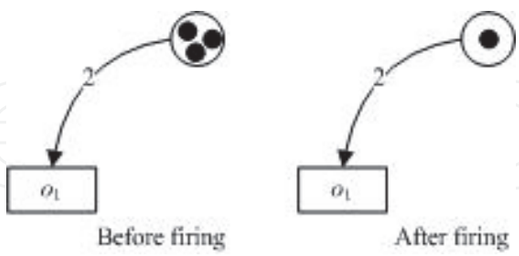



Fig. 21. Resource consumption

## 4. Formal definitions

For a convenience, the extension is called *rwPN* (*r*eadable and *w*ritable Petri Net ). The following discussion will refer some concepts of Petri Net or Coloured Petri Net.

**Definition 4.1** ( *rwPN* ). 6-*tuple* $N = (S_c, S_v, T; R, W, F)$, *where*

1. $S_c \cap S_v = \emptyset$;
   $(S_c \cup S_v) \cap T = \emptyset$;
   $S_c \cup S_v \cup T \neq \emptyset$;
2. $R \subseteq S_v \times T$;
   $W \subseteq T \times S_v$;
   $F \subseteq S_c \times T \cup T \times S_c$ ;
3. $S_v = dom(R) \cup cod(W)$;
   $S_c \subseteq dom(F) \cup cod(F)$;
   $T \subseteq dom(F) \cup dom(W) \cup cod(F) \cup cod(R)$.

where $S_c$, $S_v$, $T$ are finite sets and $R$, $W$, $F$ are relations respectively.

Let $\mathscr{S}$ be a function from a set of *place*s to a set of colors, such as

$$\mathscr{S} : S \to \mathbb{C}$$

$$S = S_v \cup S_c$$

where $S$ is a set of *place*s , $S_v$ is a set of *Variable place*s , $S_c$ is a set of *control place*s , $\mathbb{C}$ is a set of colors. Here and afterward, sets are finite .

Because $dom(F) \cup cod(F)$ can contain *place*s , *transition*s , or both. For a convinence and in a discussion context, $dom(F) \cup cod(F)$ can be viewed as a set of *place*s , a set of *transition*s or both on need.

$F$ only connects between $S_c$ and $T$ or $T$ and $S_c$, and is defined as

$$F \subseteq S_c \times T \cup T \times S_c$$

where $T$ is a set of *transition*s . $F$ describes a situation in which *token*s are flowable, i.e. *token*s can flow out of this *place* and into another one along *Control Flow* $F$.

$R$ only connects between $S_v$ and $T$, and is defined as

$$R \subseteq S_v \times T$$

$R$ is represented graphically by a line with circle $\multimap$ which points to a *transition* .

$W$ also only connects between $T$ and $S_v$, and is defined as

$$W \subseteq T \times S_v$$

$W$ is represented graphically by a line with circle $\multimap$ but which points to a *variable place* .

Semantically similar to Coloured Petri Net, a *transition* is also controlled by a *GUARD* that is a boolean value. Let $\mathscr{G}_c$ denote this kind of *GUARD* , called *control guard* .

$$\mathscr{G}_c : T \to BOOL$$

$T$ is a set of *transition*s , and $\mathscr{G}_c$ changes from *TRUE* to *FALSE* alternatively. When the *preset* of a *transition* has enough *token*s , the *control guard* of *transition* is *TRUE*. Otherwise, the *control guard* of *transition* is *FALSE*.

Additionally, there has another *GUARD* to monitor the *Variable place*s in the *preset* of *transition* . Such a *GUARD* called *variable guard* . Let $\mathscr{G}_v$ be a *variable guard* and

$$\mathscr{G}_v : T \to BooleanExpression$$

where the boolean expression specifies a variable-related condition to fire a *transition* . Any operand in $BooleanExpression$ either holds a constant or just is a *variable place* .

Accordingly, the two *GUARD*S codetermine whether a *transition* can be fired.

The third part of a *transition* is statements which describes the expected action of a *transition* . Statements can be written by a program language, such as C, Java, etc. Let function $\mathscr{L}$ be the action script of a *transition* , then

$$\mathscr{L} : T \to statement$$

Summarily, a *transition* has three components: $\mathscr{G}_c$, $\mathscr{G}_v$ and $\mathscr{L}$. Let function $\mathscr{H}$ be

$$\mathscr{H} : T \to \mathscr{G}_c \cup \mathscr{G}_v \cup \mathscr{L}$$

Obviously, $\forall t \in T$,

when $R(t) = \emptyset$ and $W(t) = \emptyset$ then $\mathscr{L}(t) = \emptyset$ and $\mathscr{G}_v(t) = \emptyset$.

when $\mathscr{G}_v(t) = \emptyset$ and $\mathscr{L}(t) = \emptyset$, the *transition* is the same as that of Coloured Petri Net.

when $\mathscr{G}_c(t) = \emptyset$, the *transition* can fire without any *Control Flow* .

when $\mathscr{G}_v(t) = \emptyset$, the *transition* can fire when the *preset* of *transition* has enough *token*s .

As a default, if $\mathscr{G}_v(t) = \emptyset$, let $\mathscr{G}_v(t) = TRUE$, and if $\mathscr{G}_c(t) = \emptyset$, let $\mathscr{G}_c(t) = TRUE$.

**Definition 4.2** ( *cvNet* ). *given a* rwPN $N = (S_c, S_v, T; R, W, F)$, $N_{cv}$ *is called* cvNet *(control view net) of* $N$, *and*

$$N_{cv} = (S_c, T|_{S_c}; F)$$

*where* $T|_{S_c}$ *is a set of* transitions *connecting with* control places .

**Definition 4.3** ( *dvNet* ). *given a* rwPN $N = (S_c, S_v, T; R, W, F)$, $N_{dv}$ *is called* dvNet *(data view net) of* $N$, *and*

$$N_{dv} = (S_v, T|_{S_v}, R, W)$$

*where* $T|_{S_v}$ *is a set of* transitions *connecting with* Variable places .

**Theory 4.1.** *If* $N_{cv}$ *is* cvNet *and* $N_{dv}$ *is* dvNet *of* rwPN $N$ *respectively, then*

$$N = N_{cv} \cup N_{dv}$$

**Proof:** where $\cup$ is the operator of graph union, based on definitions *rwPN* $N$, *cvNet* $N_{cv}$ and *dvNet* $N_{dv}$, the conclusion holds obviously.

So, if a system is modeled by *rwPN* , the system features can be captured from two views. One view is from *Control View Net* which describes the control flow relationship of system. The other one is from *Data View Net* which describes the entity relationship of system.

**Definition 4.4** ( *rwPNs* ). 4-*tuple* $\Sigma = (N, \mathscr{S}, \mathscr{H}, M_0)$, *where*

1. $N = (S_c, S_v, T; R, W, F)$ *is a* rwPN .
2. $\mathscr{S} : S \to \mathbb{C}$,
   $S = S_c \cup S_v$;
   $\mathbb{C} = Integer \cup DATATYPE$ *is a set of colors;*
   $\forall s_c \in S_c, \mathscr{S}(s_c) = Integer$;
   $\forall s_v \in S_v, \mathscr{S}(s_v) = DATATYPE$;
3. $\mathscr{H} : T \to \mathscr{G}_c \cup \mathscr{G}_v \cup \mathscr{L}$
   $\mathscr{G}_c : T \to BOOL$
   $\mathscr{G}_v : T \to BooleanExpression$
   $\mathscr{L} : T \to statement$
4. $M$ *is the marking of* $N$, $M = M_c \cup M_v$; *and*
   $M_c : S_c \to \mathbb{N}$
   $M_v : S_v \to \{1\}$
   $M_0$ *is the initial marking;* $M_{C_0}$ *is the initial marking of* $S_c$; $M_{v_0}$ *is the initial marking of* $S_v$;

Specially, if let color set $\mathcal{C} = \mathbb{C} \cup Integer \cup DATATYPE \cup BOOL \cup BooleanExpression \cup statement$ and $cd = \mathscr{S} \cup \mathscr{H}$, *rwPNs* can be mapped to Coloured Petri Net$\mathcal{N} =< P, T, Pre, Post, \mathcal{C}, cd >$ . The mapping from *rwPNs* to Coloured Petri Netwill further be discussed in the next section.

**Definition 4.5** ( *cvNets* ). *Given* rwPNs $\Sigma = (N, \mathscr{S}, \mathscr{H}, M_0)$, *call* $N^{cs}$ cvNets *and*

$$N^{cs} = (N_{cv}, \mathscr{S}|_{S_c}, \mathscr{H}|_{S_c}, M_0|_{S_c})$$

*where*
$M_0|_{S_c} = M_{c_0}$; *the initial marking of* control places ;

$\mathscr{S}|_{S_c} : S_c \to integer$; *the colors of* control places ;
$\mathscr{H}|_{S_c} = \mathscr{G}_c$; *only the* Control Guards *of* transitions *are concerned.*

**Definition 4.6** ( *dvNets* ). *Given* rwPNs $\Sigma = (N, \mathscr{S}, \mathscr{H}, M_0)$, *call* $N^{ds}$ dvNets *and*

$$N^{ds} = (N_{dv}, \mathscr{S}|_{S_v}, \mathscr{H}|_{S_v}, M_0|_{S_v})$$

*where*
$M_0|_{S_v} = M_{v_0}$; *the initial marking of* Variable places ;
$\mathscr{S}|_{S_v} : S_v \to DATATYPE$ ; *the colors of* Variable places ;
$\mathscr{H}|_{S_v} = \mathscr{G}_v \cup \mathscr{L}$;*the* Variable Guards *of* transitions *and the action script are concerned.*

**Theory 4.2.** *If* $N^{cs}$ *is* cvNets *and* $N^{ds}$ *is* dvNets *of* rwPNs $N$ *respectively, then*

$$N = N^{cs} \cup N^{ds}$$

**Proof:** where $\cup$ is the operator of graph union, based on definitions *rwPNs* $N$, *cvNets* $N^{cs}$ and *dvNets* $N^{ds}$, the conclusion also holds.

When the dynamic features of system are concerned, there are also two views to study the system, *cvNets* and *dvNets* . Specially, Because *GUARD* $\mathscr{G}_c$ and *GUARD* $\mathscr{G}_v$ in $\mathscr{H}$ will determine whether transitions can be fired, some properties of program, e.g.,the code coverage and key path in the program testing, can be computed automatically based on $\mathscr{H}$.

## 5. Dynamic semantics

**Definition 5.1.** $x \in S \cup T$,
$r_s(x) = \{a|(a,x) \in R \wedge x \in T\}$, *the set of* Variable places *read by* transition $x$;
$w_s(x) = \{a|(x,a) \in W \wedge x \in T\}$, *the set of* Variable places *written by* transition $x$;
$r_t(x) = \{a|(x,a) \in R \wedge x \in S\}$, *the set of* transition *read* variable place $x$;
$w_t(x) = \{a|(a,x) \in W \wedge x \in S\}$, *the set of* transition *write* variable place $x$;
$r(x) = r_s(x) \cup r_t(x)$, *read on* $x$;
$w(x) = w_s(x) \cup w_t(x)$, *write on* $x$;
${}^\bullet t = \{p|(p,t) \in F\}$, *the* preset *of* transition $t$;
$t^\bullet = \{p|(t,p) \in F\}$, *the* postset *of* transition $t$;

**Definition 5.2.** *In marking* $M = M_c \cup M_v$, $t$ *is C_enabled, iff*

$$C\_enabled(M,t) \equiv enabled(M_c,t)$$

*where,* $enabled(M_c,t)$ *denotes* ${}^\bullet t$ *has enough number of* tokens *and leads to* control guard $\mathscr{G}_c =$ TRUE. *If* ${}^\bullet t = \emptyset$, *let* $\mathscr{G}_c =$ TRUE.

$t$ *is V_enabled, iff*

$$V\_enabled(M,t) \equiv enabled(M_v,t)$$

*where,* $enabled(M_v,t)$ *denotes* variable guard $\mathscr{G}_v$ *holds. If* $r(t) \cup w(t) = \emptyset$, *let* $\mathscr{G}_v =$ TRUE.

**Definition 5.3.** $t$ *is firable in marking* $M$, *called* $M[t>$, *iff in marking* $M$, $t$ *is both C_enabled and V_enabled ,*

$$M[t> \equiv C\_enabled(M,t) \wedge V\_enabled(M,t))$$

Accordingly, *transition t* is controlled by both $\mathscr{G}_v$ and $\mathscr{G}_c$. $t$ can fire iff all required resource have been provided and all the related variables hold legal values.

**Definition 5.4.** *if $M[t>$, let $M'$ be the succession marking of $M$, then $M[t > M'$ , and $M'$ is, if $s \in S_c$:*

$$M'(s) = \begin{cases} M(s) - 1 \text{ if } s \in^\bullet t - t^\bullet \\ M(s) + 1 \text{ if } s \in t^\bullet - ^\bullet t \\ M(s) \quad \text{if } s \notin {}^\bullet t^\bullet \text{ or } s \in^\bullet t \cap t^\bullet \end{cases}$$

*if $s \in S_v$:*

$$M'(s) = \begin{cases} Val(\mathscr{H}(t)) \text{ if } s \in w(t), \\ M(s) \quad \text{if } s \in r(t) \end{cases}$$

*where $Val(\mathscr{H}(t))$ denotes the current value of $s$ updated by $t$.*

From above definition, a transition will be affected by three factors: tokens in control place, *variable guard*(B) and *control guard*(E).

**Definition 5.5.** *$t_1, t_2 \in T$ and $t_1 \neq t_2$. In marking $M$ , $M[t_1 > \wedge M[t_2 >$,*

1. *if $s \in {}^\bullet t_1 \cap {}^\bullet t_2, M(s) < 2$, Then, between $t_1$ and $t_2$, there exists* P_confilict*, call $P\_conflict(t_1, t_2)$.*
2. *if $t_1, t_2 \in T_v$, $w(t_1) \cap w(t_2) \neq \emptyset$ , then between $t_1$ and $t_2$, there exists* W_conflict*, call $W\_conflict(t_1, t_2)$.*
3. *if $t_1, t_2 \in T_v$, $(w(t_1) \cap r(t_2)) \cup (r(t_1) \cap w(t_2)) \neq \emptyset$ , then between $t_1$ and $t_2$, there exists* RW_conflict*, call $RW\_conflict(t_1, t_2)$.*

*$t_1$ is conflict with $t_2$, call $conflict(t_1, t_2)$, iff*

$$conflict(t_1, t_2) \equiv P\_conflict(t_1, t_2) \vee W\_conflict(t_1, t_2) \vee RW\_conflict(t_1, t_2)$$

**Definition 5.6.** *$t_1, t_2 \in T$ and $t_1 \neq t_2$, in marking $M$, $t_1$ is concurrent with $t_2$ , iff*

$$M[t_1 > \wedge M[t_2 > \wedge \neg conflict(t_1, t_2)$$

**Definition 5.7.** *A directed net $N = (B, E; F)$ is a general occurrence net, if $N$ satisfies the condition:*

$$F^+ \cap (F^{-1})^+ = \emptyset$$

*where,*

$F^+ = F \cup F \circ F \cup F \circ F \circ F \cup \cdots;$

$F^{-1} = \{(x, y) | (y, x) \in F\};$

$(F^{-1})^+ = F^{-1} \cup F^{-1} \circ F^{-1} \cup F^{-1} \circ F^{-1} \circ F^{-1} \cup \cdots.$

*"$\circ$" is a symbol of relation composition.*

A general occurrence net $N = (B, E; F)$, if $x, y \in E$ or $x, y \in B (x \neq y)$, and $(x, y) \in F^+$, then there is a partial order relation between $x$ and $y$, denoted as $x \prec y$, or more strictly $x \prec_N y$ .

**Definition 5.8.** *A general occurrence net $N' = (B, E; F')$, if there is a mapping between $N'$ and Net system $\Sigma = (N, C, I, M_0)$, i.e. : $\rho : N' \to \Sigma$ satisfies conditions:*

1. $B = B' \cup \{\varepsilon\}$;
   $\rho(B') \subseteq S_p \cup S_v, \rho(\varepsilon) = \emptyset$;
   $\rho(E) \subseteq T$;
   $\rho(F') \subseteq F \cup R \cup W$;
   $\exists (x,y) \in F' \wedge (x = \varepsilon \vee y = \varepsilon) \Rightarrow \rho(x,y) = \emptyset$;
2. $\forall (x,y) \in F' : \rho(x) \in S_p \vee \rho(y) \in S_p$
   $\Rightarrow \rho(x,y) = (\rho(x), \rho(y)) \in F$;
   *let* $\widetilde{N} = (B, E, F' - \{(x,y)\})$,
   $\rho(x) \in S_v \wedge (\rho(x), \rho(y)) \notin R$
   $\Rightarrow (\rho(x), \rho(y)) \in W - R \wedge^{\bullet} x \neq \emptyset \wedge \neg(x \prec_{\widetilde{N}} y)$;
   $\rho(y) \in S_v \wedge (\rho(x), \rho(y)) \notin W$
   $\Rightarrow (\rho(x), \rho(y)) \in R - W \wedge \left|y^{\bullet}\right| = 1 \wedge (\rho(y), \rho(y^{\bullet})) \in W$;
3. $\forall e \in E$ :
   $\rho({}^{\bullet}e) \cap S_p =^{\bullet} \rho(e)$;
   $\rho(e^{\bullet}) \cap S_p = \rho(e)^{\bullet}$;
   $r(\rho(e)) \subseteq \rho({}^{\bullet}e) \cap S_v$;
   $w(\rho(e)) \subseteq \rho(e^{\bullet}) \cap S_v$;
   $\forall b \in B$ :
   $|b^{\bullet}| > 1 \Rightarrow \rho(b) \in S_v \wedge \forall e \in b^{\bullet} : (\rho(b), \rho(e)) \in R - W$;
   $|{}^{\bullet}b| > 1 \Rightarrow \rho(b) \in S_v \wedge^{\bullet} b \subseteq r(\rho(b)) - w(\rho(b))$
   $\wedge \exists b' \in B : \rho(b) = \rho(b') \wedge^{\bullet} b \subseteq b'^{\bullet}$;
4. $\forall b_1, b_2 \in B : \rho(b_1), \rho(b_2) \in S_v \wedge \rho(b_1) = \rho(b_2)$
   $\Rightarrow b_1 \prec b_2 \vee b_2 \prec b_1 \vee b_2 = b_1$,
   $\forall e_1, e_2 \in E, \forall x \in S_v : x \in v(\rho(e_1)) \cap v(\rho(e_2))$
   $\Rightarrow \exists b(\rho(b) = x \wedge \{e_1, e_2\} \subseteq b^{\bullet}) \vee e_1 \prec e_2 \vee e_2 \prec e_1$;
5. $\forall b_1, b_2 \in B : b_1 \neq b_2 \wedge \rho(b_1) = \rho(b_2)$
   $\Rightarrow^{\bullet} b_1 \cap^{\bullet} b_2 = b_1^{\bullet} \cap b_2^{\bullet} = \emptyset$;
6. $\forall s \in S_p : \left|\{b | \rho(b) = s \wedge^{\bullet} b = \emptyset\}\right| \leq M_0(x) \neq nout$.

Then $(N', \rho)$ is a process of system $\Sigma$ . Where,

1. Process may have a special place $\varepsilon$ which is not appeared in *rwPNs* . $\varepsilon$ denotes the partial order of transitions. Transition labels in process are transition names in *rwPNs* . Place labels in process are place names in *rwPNs* or label $\varepsilon$ . Accordingly, the process may have special arc(s) (complementary arc) ,associated with $\varepsilon$, which are also not appeared in *rwPNs* .
2. $N$ will guarantee the less complexity through removing the redundant arcs.
3. Elements of $S_p$ mapped from ${}^{\bullet}e^{\bullet}$ are similar to places of Petri nets. Elements of $S_v$ mapped from ${}^{\bullet}e^{\bullet}$ can be :
   (a) variables read by $\rho(e)$ are mapped from the set ${}^{\bullet}e$ , maybe including variable(s) associated with complementary arc(s) and written by $\rho(e)$ ;
   (b) variables written by $\rho(e)$ are mapped from the set $e^{\bullet}$, maybe including variable associated with complementary arc(s) and read by $\rho(e)$.
4. If a variable accessed by two different transitions in occurrence net, the mapped elements of the two transitions in *rwPNs* either have a partial order relation or concurrently read the variable.

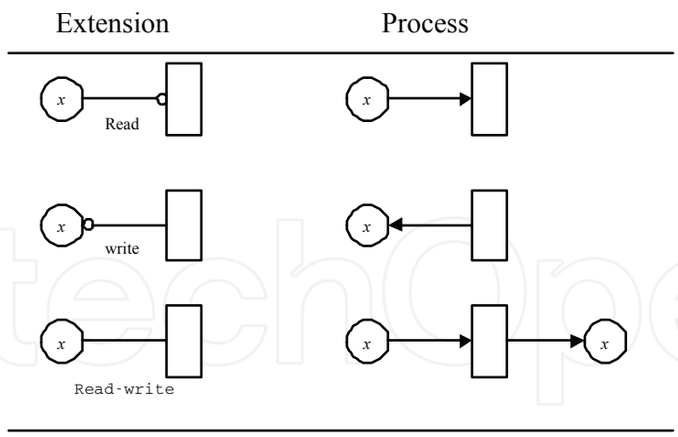



Fig. 22. Basic Place-Arc-Transition

5. The preset and the postset of transitions in $N$ must be labeled by different elements in $\Sigma$. This condition is similar to general petri net theory except that the tail part of condition is ${}^\bullet b_1 \neq {}^\bullet b_2 \wedge b_1^\bullet \neq b_2^\bullet$.
6. Not all processes need all initial resources described by the initial marking $M_0$. This condition requires $|\{b\}| \leq M_0(s)$ instead of $|\{b\}| = M_0(s)$. If the first operation upon an element of $S_v$ is read action, the element must have an initial value.

The process of Petri nets is an observational record of system . Based on the above definition of process, the mapping rules to process can be summarized by the graphic representation. In Figure 22, one variable place is only operated by one transition.

- *Read* relation is mapped to *input arc*(a out-arc with arrowhead);
- *write* relation is mapped to *output arc*(a in-arc with arrowhead);
- *read-write* relation is mapped to *input-output arc*(one read and one writečl′.

Figure 23 lists the process mapping rules on read-write concurrent or conflict relations between two transitions. When one transition writes after the other transition has read, place $\varepsilon$ is introduced to denote a sequential relation between the two transitions. When two transitions write the same place at the same time, place $\varepsilon$ is introduced to denote the sequential order relation.

Based on above rules, more complex mapping specification can be work out.

## 6. Analysis

To verify the formal specification, all petri net theory can be applied. In this section, we will discuss some special features.

**Theory 6.1.** *A program has only one place s,*${}^\bullet s = \emptyset$.

**Proof:** The computing target is initialized by only one computing unit. The unit must provide all initial condition for the computing target. Meanwhile the unit will also ask for cooperation with other units because the unit can not fulfill the special computing. If the number of ${}^\bullet s \neq \emptyset$ and greater than 1, the computing target will be more than 1 and are initialized by different

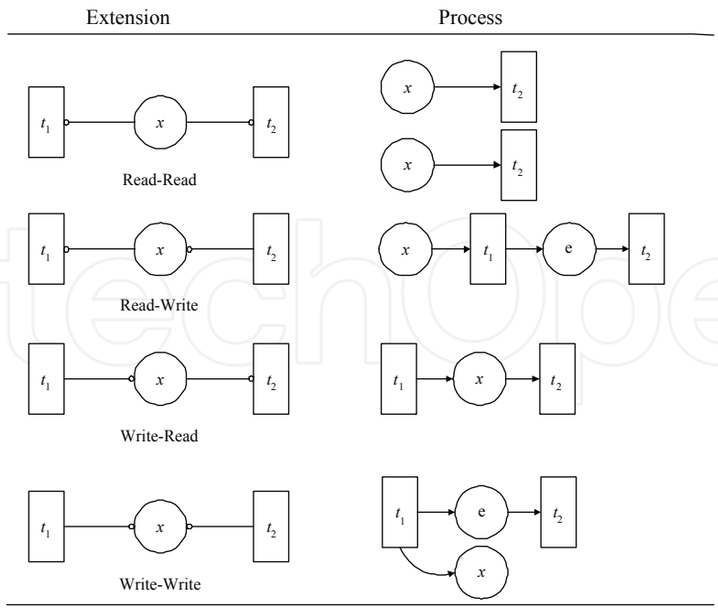



Fig. 23. One Place and Two Transition

computing units. In this case, we can make a more abstract computing unite $S$ covers these initialized computing units. Obviously, the new abstract computing unit $S$ is one and only.

**Theory 6.2.** *if* $s^{\bullet} = \emptyset$*, then* $s$ *is a finished status.*

**Proof:** When a computing task reach a status in which all other status are no changed anymore, i.e. the program reach a fixed point.

The above two theorems show there exists the boundary in a program specification.

Let $S$ is a step in marking $M$ of system, then $S$'s successive marking is $M$, $\forall p \in P$, $P$ is a set of all status.

$$M'(p) = M(p) + \sum_{t \in T} I_{+}(p,t)X(t) - \sum_{t \in T} I_{-}(p,t)X(t)$$

where, $X(t)$ denotes the number of transition $t$ appearing in step X. $M'$ is the succession of $M$, written as $M[X > M'$, i.e. , after $X$ firing, reaches from marking $M$ to $M'$. $I_{-}$ denotes the number of consumed tokens or the value for one variable. $I_{+}$ denotes the number of produced tokens or the updated value for one variable.

So, system will step into a new status, and

1. In the same step, more than one action can be fired.
2. Based on the preset of transition, $I_{-}$ , $I_{+}$ and $M$ , the marking $M'$ can be determined.

Resources or control flow are described by tokens, the number of tokens in system must be conserved. The property of conservation of token number is called invariant. Invariable(including T invariant and S invariant) is limited within boundary, i.e. local invariant and system invariant. The invariant can be computed through reference matrix.

From initializing computing to result returning, the task must be fulfilled in $n$ steps. occurrence sequence:

$$M_0[S_1 > M_1[S_2 > \cdots M_n[S_n > M$$

we can get two sequences:

1. sequence of status changing:$M_0M_1M_2 \cdots M_nM$;
2. sequence of transition changing:$S_1S_2 \cdots S_n$.

**Theory 6.3.** *Any marking $M_i$ can reach at from the initial marking $M_0$, i.e. there exists one sequence:*

$$M_0[S_1 > M_1[S_2 > \cdots M_j[S_j > M_i$$

**Proof:**

If such a sequence is not exists, then there are two cases:

- Net is not connective. So, there are at least two no connective subnet and the two subnet have no any intercourse. Because any subnet has its special computing target, the two computing targets are not interactive.

  That is inconsistency with the one and only of computing target. Therefore, the two computing targets are unsure.
- If net is connective, but there exists a marking $M_k$ can not reach. In $M_k$, place $p_k$ has no token. $p_k$ is not isolated and has pre-transition $t_k$. So, there is a function $I_+$. Because $p_k$ has no token, therefore $t_k$ is not enable, i.e., $t_k$ can not fire. In other words, the program has an operation and the operation will never be executed. So, such a computing procedure is not optimization and even wrong.

From above, any marking $M_i$ is reachable from the initial marking $M_0$.

**Theory 6.4.** *elements in net are partial order and there exists least upper bound and greatest lower bound*

**Proof:** In any system , every operation or status serves for the computing target. Therefore, all operations and status are correlation, i.e. all are useful for the computing target. So, there does not exist any independent subnet, and all elements are in the same partial order set. The initial set ${}^\bullet s = \emptyset$ is greatest lower bound. $s^\bullet = \emptyset$ is least upper bound.

In the following discussion, let $\mathscr{C}$ is a Coloured Petri Nets , $\mathscr{R}$ is a *rwPNs* . $\mathscr{C}$ is the Coloured Petri Netmapping of $\mathscr{R}$. For any element $x \in \mathscr{R}$, $x'$ is the mapping element in $\mathscr{C}$, and vice versa.

**Theory 6.1.** *if $t' \in \mathscr{C}$ is firable, then t is also firable.*

**Proof:** If $t' \in \mathscr{C}$ is firable , then three condition must be satisfied.

1. the *preset* of $t'$ has enough tokens;
2. the *postset* of $t'$ has enough capacity;
3. the arc expression ${}^\bullet t' \times \{t'\}$ can be evaluated correctly.

In $\mathscr{R}$, whether $t$ fires or not depends on *preset* of $t$. Let $p$ be the *preset* of $t$. $p$ may contain *control place* $p_p$,*variable place* $p_v$ or both. For a convenient, suppose $p_p$ and $p_v$ respectively contains only one single *place*. for *control place* , $p_p$ is same as the *preset* of $t'$. For *variable place* , If $(p_v, t) \in R$, then there are two corresponding arcs,

$(p'_v, t') \in F$;

$(t', p'_v) \in F$;

Because $t' \in \mathscr{C}$ is firable, $p_v$ has enough token and capacity.

If $(t, p_v) \in W$, then there are two corresponding arcs,

$(p'_v, t') \in F$;

$(t', p'_v) \in F$;

Because $t' \in \mathscr{C}$ is firable, $p_v$ has enough token and capacity.

The arc expression ${}^\bullet t' \times \{t'\}$ can be evaluated correctly, and the number of tokens in *variable place* remain unchanged. Meanwhile, the logical condition in arc expression ${}^\bullet t' \times \{t'\}$ are removed and put into the *variable guard* of $t$, consequently, the three condition of $t$'s firing

1. *control guard* holds;
2. *variable guard* holds;
3. the value of *variable place* can be evaluated correctly.

can be satisfied. Therefore, $t$ also is firable.

**Theory 6.2.** *If $b' \in \mathscr{C}$ is reachable, then $b \in \mathscr{R}$ is reachable.*

**Proof:**

Because $b'$ is reachable, therefore *transition*s in the *preset* of $b'$ can fire. Let the *preset* of $b'$ is $T'$ and the *preset* of $T'$ is $P'$. To prove $b$ is reachable, the *preset* of $b$ must be firable. Let the *preset* of $b$ is $T$ and the *preset* of $T$ is $P$.

for $\forall p \in P$, if $p$ is a *control place* , obviously $t$ is firable. If $p$ is a *variable place* , the expression of arc $(p', t')$ in $\mathscr{C}$ contains *variable guard* of $t$ as the logical conditions of arc expression. And the arc expression $(p', t')$ can be evaluated correctly , therefore $p$'s *variable guard* can hold *TRUE*, i.e., in *rwPNs* $t$ can fire, because $b$ is the *postset* of $t$, accordingly $b$ is reachable.

**Theory 6.3.** *Let $M'_1$ is a marking of $\mathscr{C}$ and $M_1$ is a marking of $\mathscr{R}$. if $M'_1$ is reachable, then $M_1$ is also reachable, and there is a occurrence sequence $M_0[t_1 > \cdots t_i > M_i[t_{t+1} \cdots t_n > M_1$.*

**Proof:** According to theorem 6.2, the theorem holds.

**Theory 6.4.** *if $\mathscr{C}$ is live, then $\mathscr{R}$ is also live.*

**Proof:** because $\mathscr{C}$ is live, therefore for any transition in $\mathscr{C}$ can fire and any marking of $\mathscr{C}$ can be reachable. Accordingly to 6.2 and 6.3, the theorem can be proved.

**Theory 6.5.** *Let $M'$ is the marking of $\mathscr{C}$. $\mathscr{R}$ is bounded iff there exists a natural number bound $n$ so that for every reachable marking $M' \leq n$.*

**Proof:** 1. If $M' \leq n$. $\forall p' \in M'$, if $p \in \mathscr{R}$ is a *control place* , obviously the number of tokens in $p$ is same as the number of tokens in $p'$. if $p$ is a *variable place* , then based on the definition of *variable place* in *rwPNs* , the number of tokens in a *variable place* always is 1. $\forall p \leq n$ holds. Therefore $\mathscr{R}$ is bounded.
2. If $\mathscr{R}$ is bounded. $\forall p \in \mathscr{R}$, if $p$ is a *variable place* , then the number in $p$ is 1. If $p$ is a *control place* , because $\mathscr{R}$ is bounded, therefore there exists a natural number bound $n$ and $p \leq n$ holds. Because the mapping rules don't change the number of tokens in a *place*, therefore $\forall p' \in \mathscr{C}$, the number in $p'$ is same as the number in $p$. Accordingly, there exists a natural number bound $n$ and $p' \leq n$. Therefore $M' \leq n$ holds.

## 7. Example

In this section, we formalize several programs to illustrate *rwPNs* usage.

### 7.1 Min

A function, $foo$, computes the value of $5 * min(x,y)$ with parameters, $x$ and $y$. Suppose the code is such as

```
int foo(int x,int y)
 {
    int z;
    if x<y
        z := x;
    else
        z:=y;
    z:=5*z;
    return z;
 }
```

Obviously *foo* contains three variables, $x$,$y$ and $z$. Let

$$S_v = \{x, y, z\}$$

Generally, one statement can be modeled as one transition. Omitting the function head , the variables declaration and statement *return*, three main statements are left,

1. if $x < y$ $z := x$
2. else $z := y$
3. $z := 5 * z$

Let *transition* $t_{if}$,*transition* $t_{else}$ and *transition* $t_{min}$ describe the three statements respectively, then $\mathscr{L}$ can respectively be

$$\mathscr{L}(t_{if}) : z := x$$
$$\mathscr{L}(t_{else}) : z := y$$
$$\mathscr{L}(t_{min}) : z := 5 * z$$

In the three main statements, statement 1 and statement 2 are enclosed by *if-else*. The boolean expressions in statement *if-else* can be *GUARD*S .

Let us focus on *transition* $t_{if}$. In $t_{if}$, two variables $z$ and $x$ are concerned. $t_{if}$ has two *GUARD*S . One, *control guard* , is to test whether ${}^\bullet t_{if}$ has enough *token*s . When ${}^\bullet t_{if}$ have one *token* , *control guard* is *TRUE*, otherwise *FALSE*. The *control guard* determines whether statement 1 can be continued or not. The other one, *variable guard* , is to test whether $x < y$ holds. The *variable guard* also determines whether statement $z := x$ can be continued or not. Therefore, the two guards are

$$\mathscr{G}_c(t_{if}) = \{TRUE, FALSE\}$$
$$\mathscr{G}_v(t_{if}) = \{x < y\}$$

In $\mathscr{G}_v(t_{if})$, variable $x$ and variable $y$ are read. Therefore, in $\mathscr{L}(t_{if})$, variables $z$ is written,and variable $y$ is read. Consequently, *transition* $t_{if}$ concerns three variables , $x$(read),$y$(read) and $z$(written). The specification can be represented as that in Figure 24. Similarly, *Transition* $t_{else}$ can be described in figure 25.

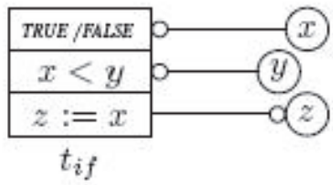



Fig. 24. Transition $t_{if}$

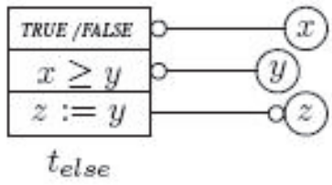



Fig. 25. Transition $t_{else}$

Statements in *if-else* are mutual exclusive, i.e. statement 1 and statement 2 are mutual exclusive. To guarantee the mutual exclusion, one *control place* is applied. Let $C_{ifelse}$ is the *control place* (Figure 26). When $C_{ifelse}$ has one *token* , there is only one is *TRUE*between $\mathscr{G}_c(t_{if})$ and $\mathscr{G}_c(t_{else})$. Suppose $\mathscr{G}_c(t_{if})$ is *TRUE*. After $t_{if}$ fires and the *token* is consumed, then $C_{ifelse}$ loses the *token* . And then $\mathscr{G}_c(t_{if})$ changes to *FALSE*. Generally, $\mathscr{G}_c(t_{if})$ will be *TRUE*or *FALSE*alternatively on whether $C_{ifelse}$ has enough *token*s .

In statement $z := 5 * z$, $z$ appears in both sides of the assignment that means $z$ will be both read and written. Moreover, statement $z = 5 * z$ only executes after statement *if-else* finishes, i.e. $t_{min}$ follows both $t_{if}$ and $t_{else}$. Let *control place C* determine the consecutive order. Then the program specification can be represented as that in figure 27.

$t_{min}$ will execute when *C* has one *token* , therefore the *variable guard* of $t_{min}$ is not necessary. Of course, any *GUARD* can be added if need.

In *rwPNs* , if all *Variable place*s , *Read Arc*s and *Write Arc*s are omitted, the *cvNets* of program can abstracted as figure 28. In fact, that in figure 28 is a P/T nets and the analysis techniques of Petri nets can be applied . Obviously *cvNets* describes the *Control Flow* framework of program.

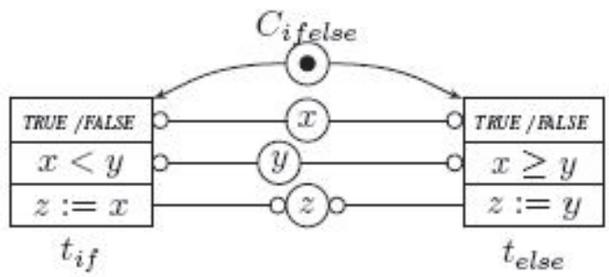



Fig. 26. Exclusive transitions

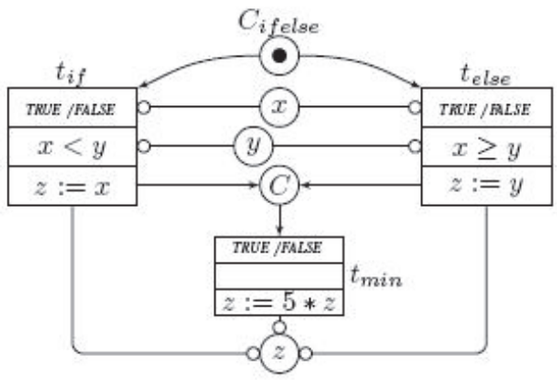



Fig. 27. *rwPN* Specification of $5 * min(x, y)$

Similarly, if all *control place*s , *Control Arc*s and *transition*s without write/read arcs are omitted, then the *dvNets* of program can be abstracted as that in figure 29. Because *dvNets* has not *Control Flow* , all transitions are concurrent theoretically. *dvNets* describes the dataflow relationship among entities of program.

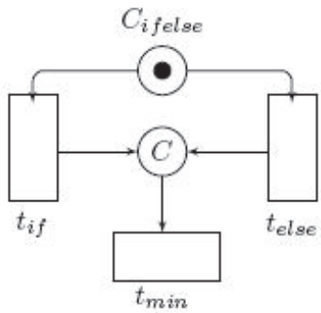



Fig. 28. *cvNets* of $5 * min(x, y)$

Informally, when an engineer starts to design a program, he can push *Control Flow* ( *cvNets* ) aside first, and only focus on data flow( *dvNets* ), i.e., entity relations. When a computing machine has be chosen to implement the program, the *Control Flow* ( *cvNets* ) can be integrated at later. Accordingly, the final model of program just is composed of *cvNets* and *dvNets* .

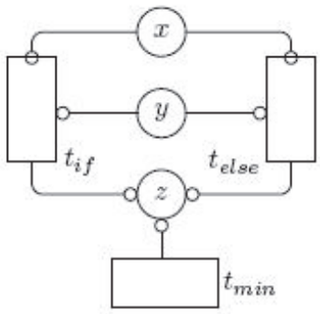



Fig. 29. *dvNets* of $5 * min(x, y)$

### 7.2 SORT

$n(n \geq 2)$ numbers should be sorted ascendingly. Let these $n$ numbers be $x_1, x_2, \cdots, x_n$. *rwPN* specification is described as in figure 30, where, $\Sigma = (N, C, I, M0)$, and $N = (S_p, S_v, T; R, W, F)$.

Control place set: $S_p = \phi$ ;

Variable control set: $S_v = \{v_i | 1 \leq i \leq n\}$;

(Variable) Transition set: $T = \{t_i | 1 \leq i < n\}$;

Control flow set: $F = \phi$ ;

Read relation (R) and write relation (W): $R = W = \{< v_i, t_i >, < v_{i+1}, t_i > | 1 \leq i < n\}$;

$C$ is the type set of $x_i$;

Initial markings $M_0$: $\{E_0(t_i) = false, M_{0v}(v_i) = x_i | 1 \leq i \leq n\}$;

$I$, the specification of transition $t_i$: for $\forall t_i T$: $B(t_i) \equiv v_i > v_{i+1}$;
$A(t_i) \equiv v_i, v_{i+1} = v_{i+1}, v_i$ ; This statement denotes that $v_i$ exchanges the value with $v_{i+1}$;

The semantics of marking varying is: If $\forall t_i \ T : M[t_i > M' \leftrightarrow M(v_i) > M(v_{i+1}) \wedge M(v_i) = M(v_{i+1}) \wedge M'(v_{i+1}) = M(v_i)$, then $E(t_i) = true$; If not, then $E(t_i) = false$.

In figure 30, transitions firing may reduce the number of descending arrays of $M(v_i)(1 \leq i \leq n)$. When marking satisfies $\forall v_i \ S_v \wedge i \neq n : M(v_i) \leq M(v_{i+1})$, then $\forall t_i \ T : B(t_i) = false \wedge E(t_i) = true$, and the sort arrives at the fixpoint, the sort accomplishes the purpose.


Fig. 30. SORT Specification

In figure 30, there is no sequential relation among $t_1, t_2, \cdots, t_{n-1}$, that is to say, transitions can fire concurrently. The semantics of sort in figure 30 is: Transition $t_i$ is to compare the current value of $v_i (M(v_i) = x_k)$ and that of $v_{i+1} (M(v_{i1}) = x_j)$. If the current value of $v_i$ is bigger than that of $v_{i+1}$, then $v_i$ exchanges the current value with $v_{i+1}$; and then the guard of transition $t_i$

($E(t_i)$ and $B(t_i)$) holds false. When every transition guard holds false, the sort arrives at the fixpoint and terminates.

The specification in figure 30 has not any resources restriction and $n$ transitions can fire concurrently. If resources are restricted, e.g., only one control handle, the new specification is described as in figure 31. Where, additional sets,

Control place set: $S_p = \{s_i | 1 \leq i < n\}$;

Control transition set: $T = \{g_i | 1 \leq i < n\}$;

Control flow set: $F = \{< s_i, t_i >, < s_i, g_i > | 1 \leq i < n\} \cup \{< t_i, s_{i+1} >, < g_i, s_{i+1} > | 1 \leq i < n-1\} \cup \{< t_{n-1}, s_1 >, < g_{n-1}, s_1 >\}$;

$M_{0p}$, initial marking of control place $s_i$ : $\{M_0(s_1) = 1\} \cup \{M_0(s_i) = 0 | 1 < i < n\}$;

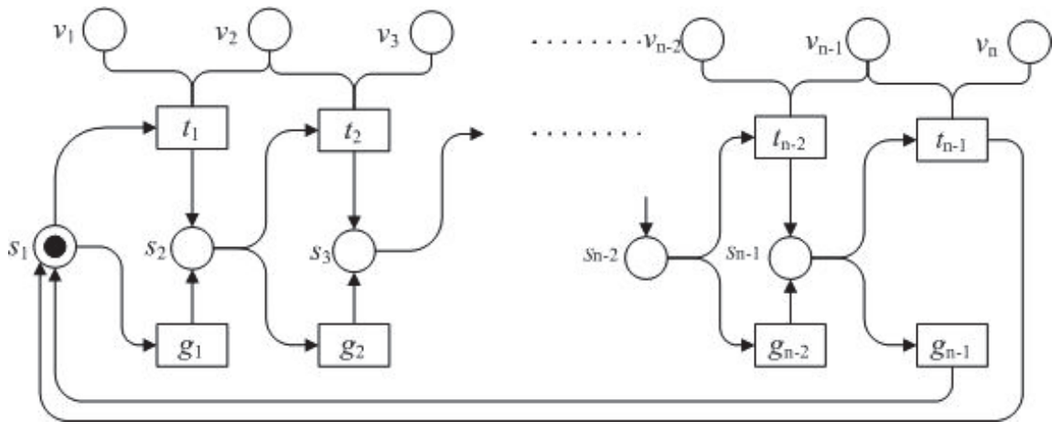



Fig. 31. Constraint SORT Specification

In figure 31, $t_i$ can fire only if $s_i$, which is the preset of $t_i$, has at least one token. After $t_i$ fires, the token in $s_i$ is consumed and a new token flows into $s_{i+1}$, which is the postset of $t_i$. Because of the nondeterminacy, the firing possibility of $g_i$ is the same as that of $t_i$. If there are not $g_i$, when $v_i v_{i+1}$, $t_i$ cant fire and the sort terminates though $t_j > t_{j+1} (j > i)$. While transitions is fired continually, the token flows along place $s_1, s_2, \cdots, s_{n-1}$, and $s_1$. In fact, the specification in figure 31 is bubble sort . Similarly, if other resources are restrained, many different sorts can be designed based on the specification in figure 30.

In system $\Sigma$, let $n = 4$, $S_v = \{v_1, v_2, v_3, v_4\}$, $T_v = \{t_1, t_2, t_3\}$, $C(v_i) = Integer$ and initial markings of $v_i$ and $t_i$ are: $M_0(v_1) = 4$, $M_0(v_2) = 2$, $M_0(v_3) = 3$, $M_0(v_4) = 1$, $M_0(t_i) = false$. We can get a process of system $\Sigma$ (Fig.9). This process describes the sort procedure from 4, 2, 3, 1 to 1, 2, 3, 4.

### 7.3 Dining philosophy

In this section, DPP(Dining philosophy problem) is treated as a program . Five philosophers, No.1 to No.5, sit around a table orderly and the No.1 is next to the No.5. Each philosopher is in one of three states: *thinking*, *hungry* and *eating*. Thinking independently for a while, each philosopher stops thinking to eat for hungriness. Each philosopher has one fork at his left hand, but he can eat only when he holds two forks, i.e. the left fork is owned by himself and the right fork is borrowed from his right neighbor. If borrows the right fork successfully, the philosopher can eat, otherwise he must wait for the right fork. After his

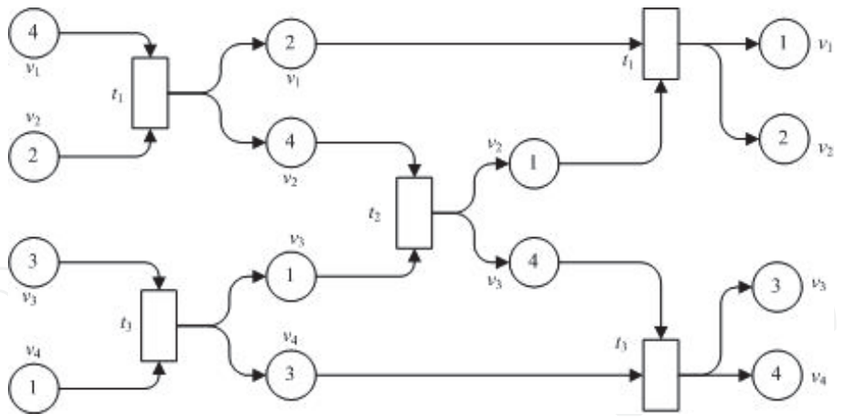



Fig. 32. SORT Process

eating, the philosopher returns the borrowed fork and frees his own fork for being lent, and then he starts to think. We suppose it is fair to use forks and there isn't any priority. In addition, a premise is that a philosopher doesn't free his holding fork(s) until he finishes his eating. Obviously, forks are the shared resources. To guarantee each philosopher can eat, each dining philosopher must finish his eating in a while and free his holding forks. Therefore, no philosopher can eat or be hungry forever.

Let $PHIL$ be the type philosophers belong to, $P_i \in PHIL$ denotes the $i$th philosopher. Variable place $P_i.state$ records the current state of philosopher, i.e. $thinking, hungry$ or $eating$. Concisely, $P_i.state = thinking, P_i.state = hungry$ and $P_i.state = eating$ are abbreviated as $P_i.thinking, P_i.hungry$ and $P_i.eating$ respectively, here the data type of $thinking, hungry$ and $eating$ are $Boolean$. Let $FORK$ be the type forks belong to, $f_i \in FORK$ denotes the fork owned by the $i$th philosopher. Variable transitions $T_i, H_i$ and $E_i$ denote operations that make philosopher $P_i$ to think, be hungry and eat respectively (see Fig.33). $T_i, H_i$ and $E_i$ all can modify the current value of $P_i.state$. In figure 33, *Control guard* locates at top level and its initial value is $false$. *Control guard* is $false$ denotes that the fire condition of transition can't be satisfied, so the transition can't fire. When the fire condition of transition is satisfied, *control guard* is *true* and the transition is *enabled*. *Control guard* is just the first control of transition's firing. *Variable guard* locates at the middle level and it represents the condition that the associated variables must satisfy. Similar to *Control guard*, when *variable guard* is *true*, the transition is *enabled*. *Variable guard* is the second control of transition's firing. The transition fires only when both *control guard* and *variable guard* are *true*. *Variable guard* is fixed once it is assigned, but *control guard* can alternate between *true* and $false$. Action description locates at bottom level and it is a collection of assignments. For example, the semantics of $T_i$ is: when *control flow* arrives at $T_i$ (*control guard* is *true*) and $P_i$ is *eating* (*variable guard* is *true*), $P_i$ changes its state (*eating*) and starts thinking. Meanwhile $P_i$ sets the current state to *thinking*. The semantics of $E_i$ and $H_i$ are similar to $T_i$'s.

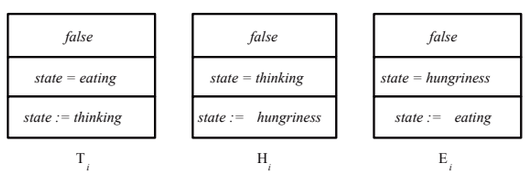



Fig. 33. Transition specification of *thinking*, *hungriness* and *eating*

In figure 34, the initial marking of $P_i$ is: $M_0(f_i) = 1, M_0(St_i) = 1$ and $P_i.state = thinking$, i.e. the initial state of $P_i$ is *thinking* and the next *enable* transition is $H_i$. When $H_i$ fires, $P_i$ is in the state of *hungriness* and requests two forks to eat, i.e. the *weight* of arc $(H_i, Sa_i)$ is 2. When $M(f_i) = 1$ and $M(Sa_i) \geq 1$, $Lf_i$ fires. Consequently, fork $f_i$ is held by $P_i$. Similarly, When $M(f_{i-1}) = 1$ and $M(Sa_i) \geq 1$, $Rf_i$ fires. Consequently, fork $f_{i-1}$ is held by $P_i$. If $Ff_i$ has two tokens, i.e. $M(Ff_i) = 2$, $E_i$ fires and $P_i$ starts to eat with two forks. Marking $M(Sf_i) = 1$ denotes $P_i$ is eating. $T_i$'s firing denotes $P_i$ finishes eating and start to think, at the same time, $P_i$ frees forks $f_i$ and $f_{i-1}$ ($P_1$ uses $f_1$ and $f_5$), i.e. $M(f_i) = 1$ and $M(f_{i-1}) = 1$. In Fig.34 and other figures, *arc with arrow* denotes *control flow* and *arc without arrow* denotes *read/write* relationship.

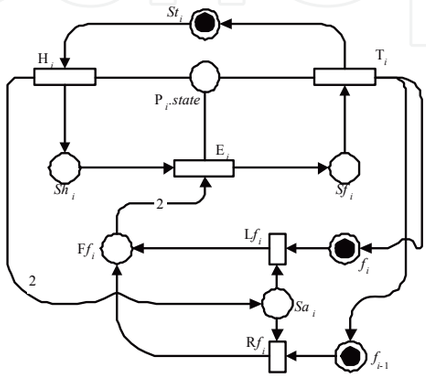



Fig. 34. Specification of $P_i$

In figure 34, *token* denotes the *control* inside $P_i$ and flows among *control places*. Although *control place* can't be read or written, we can observe the *control flow* through the change of *state*. For example, in figure 33, *variable guards* of $T_i, H_i$ and $E_i$ imply the change order of states. The current state of philosopher can be checked through *state*.

The specifications of five philosophers are similar to that in figure.34. With shared forks, the specification of DPP (figure.35) can be got from interaction among five individual philosopher specifications (figure.34).

Control place set: $S_p = \{St_i, Sh_i, Sf_i, f_i, Sa_i, Ff_i | i = 1, 2, 3, 4, 5\}$.

Variable place set: $S_v = \{P_i.state | i = 1, 2, 3, 4, 5\}$.

Transition set: $T_v = \{T_i, H_i, E_i | i = 1, 2, 3, 4, 5\}, T_p = \{Lf_i, Rf_i | i = 1, 2, 3, 4, 5\}$.

Read/Write relationship: $R = W = (P_i.state, T_i), (P_i.state, H_i), (P_i.state, E_i)$.

Control flow relationship: $F = \{(T_i, St_i), (St_i, H_i), (H_i, Sh_i), (Sh_i, E_i), (E_i, Sf_i), (Sf_i, T_i), (T_i, f_i), (T_i, f_{i-1}), (H_i, Sa_i), (Sa_i, Lf_i), (f_i, Lf_i), (f_{i-1}, Rf_i), (Sa_i, Rf_i), (Lf_i, Ff_i), (Rf_i, Ff_i), (Ff_i, E_i) | i = 1, 2, 3, 4, 5\}$, where $f_0 = f_5$.

The individual weight of other arcs is 1 respectively, but $\{W(Ff_i, E_i) = 2, W(H_i, Sa_i) = 2 | i = 1, 2, 3, 4, 5\}$.

Description of $(variables)$ *transition* $I$ is in figure.33.
Initial marking $M_0 : \{M_0(f_i) = 1, M_0(St_i) = 1, M_0(P_i.state) = thinking | i = 1, 2, 3, 4, 5\}$.

In figure.35, non-neighboring philosophers hasn't the shared resource (fork), so they can be concurrent, such as $P_1$ and $P_3(P_4), P_2$ and $P_4(P_5), P_3$ and $P_5(P_1)$. These properties can be

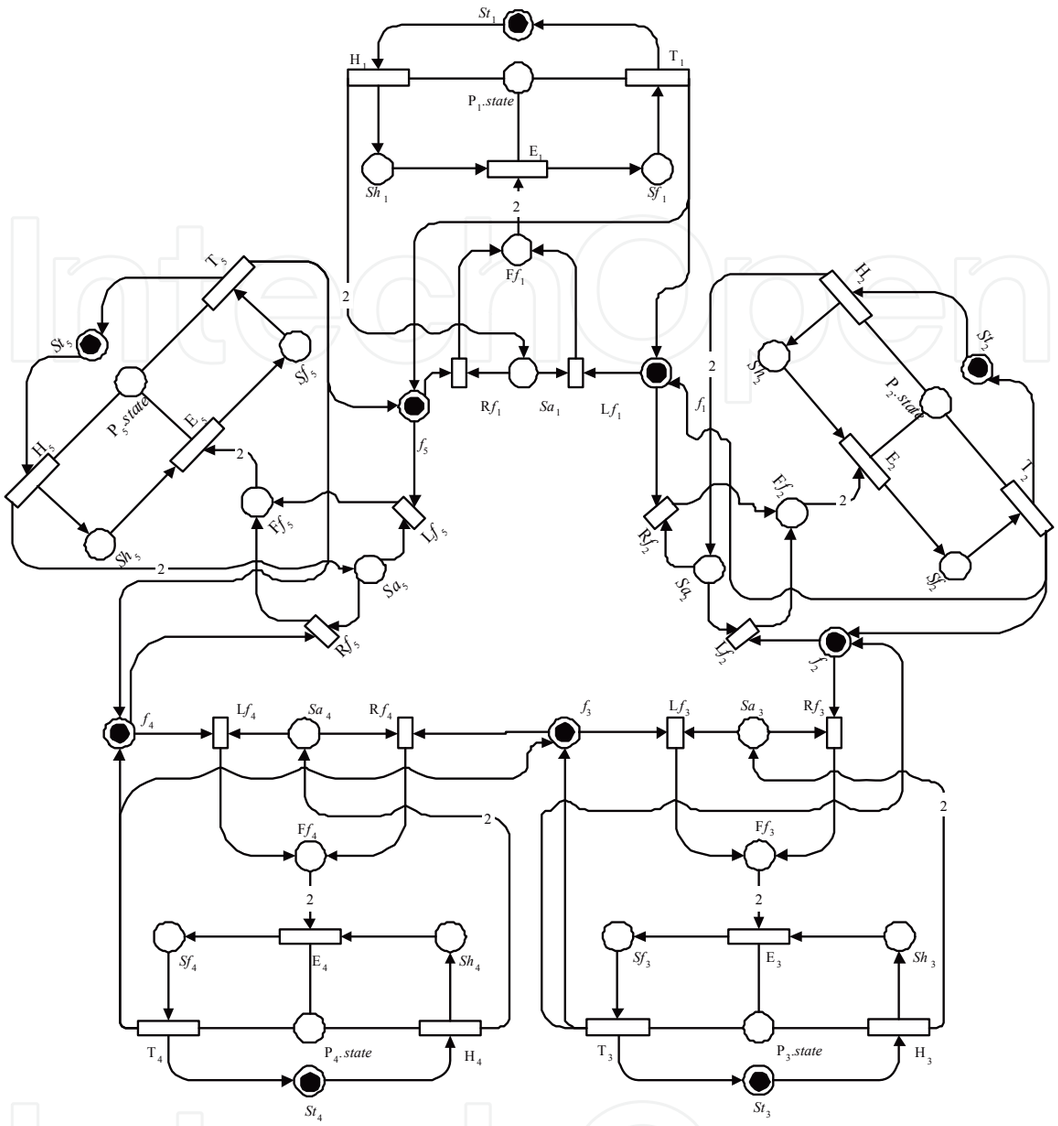



Fig. 35. *rwPN* Specification of Dining Philosopher Problem

observed from figure.35. For convenience, predicate $R_p(P_i, P_j)$ denotes that $P_i$ is next to $P_j$; predicate $T_f(f_i, f_j)$ denotes $f_i$ is next to $f_j$ and they are used by $P_i$,. Under no confusion circumstances, $M(St_i) = 1$ is abbreviated to $St_i$; and $\neg St_i$ to $M(St_i) = 0$; $f_i$ means fork $f_i$ is free and marking $M(f_i)$ is 1; $\neg f_i$ means fork $f_i$ is being used and marking $M(f_i)$ is 0.

Control places $Sf_i, St_i$ and $Sh_i$ are unreadable and to some extent are encapsulated. However, variable place *state* is readable and writable, so the state of philosopher can be observed through variable place *state*. We call *state* the *observe window* of the inner state of philosopher. Therefore, *control flow* inside a philosopher can be checked indirectly through *variable place*, and *variable place* also can be called as *interface*. On the other hand, properties on *state* can be proved through *control flow*. The following properties are expressed by UNITY logic but the proofs are Petri nets style.

| | |
|---|---|
| $P_i$.*thinking* ensures $P_i$.*hungriness*. | $P_i$ will stop thinking because of hungriness. |
| $P_i$.*hungriness* ensures $P_i$.*eating*. | The hungry philosopher has opportunity to eat. |
| $P_i$.*eating* ensures $P_i$.*thinking*. | The dining philosopher must finish his eating in a while and start to think. |

These properties ensure each philosopher is in one of three states by turn, i.e. any philosopher can't be in one of states forever. Therefore, the individual philosopher could be live.

**Proof:** Because $St_i$ is a *control place*, $M(St_i) = 1$ implies that $P_i$ is in the state of *thinking*, i.e. $St_i \rightarrow P_i$.*thinking*. Similarly, $Sh_i \rightarrow P_i$.*hungriness* and $Sf_i \rightarrow P_i$.*eating*. Therefore, property 1 can be rewritten as

$$St_i \text{ ensures } Sh_i$$
$$Sh_i \text{ ensures } Sf_i$$
$$Sf_i \text{ ensures } St_i$$

From figure.34, we can get a segment of process (figure.36). For conciseness, some detail parts are neglected in figure.36, e.g. $P_i$ must have an opportunity to hold two forks.

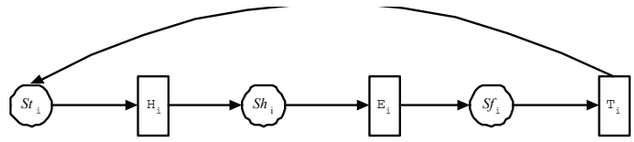



Fig. 36. A process segment of $P_i$

As showed in figure.36, *control*(token) flows from $St_i$ through $H_i$, and then into $Sh_i$. Accordingly, there is an *occurrence sequence* $St_i[H_i > Sh_i$. Obviously, $H_i$ is the operation which guarantees $St_i$ ensures $Sh_i$. Consequently, based on the premise $St_i \rightarrow P_i$.*thinking* and $Sh_i \rightarrow P_i$.*hungriness*, $H_i$ also guarantees $P_i$.*thinking* ensures $P_i$.*hungriness*.

Similarly, $P_i$.*hungriness* ensures $P_i$.*eating* and $P_i$.*eating* ensures $P_i$.*thinking* hold *true*.

**Property 1.** invariant $R_p(P_i, P_j) \rightarrow \neg(P_i.eating \wedge P_j.eating)$.

The neighboring philosophers can't eat at the same time.

**Proof:** Similar to property 1, property 2 can be rewritten as

$$invariant R_p(P_i, P_j) \rightarrow \neg(Sf_i \wedge Sf_j)$$

Suppose $P_i$ is in the state of *eating*, the firing condition of $E_i$ is

$$M(Ff_i) = 2$$

At that time, tokens in $Sf_i$ and $Sf_j$ must respectively be consumed, i.e.

$$M(Sf_i) = 0 \quad M(Sf_j) = 0$$

Similarly, $P_j$ is in state of *eating*, the firing condition of $E_j$ is

$$M(Ff_j) = 2$$

Because $P_i$ and $P_j$ are neighboring, and $M(Ff_i) = 2$ has caused $M(Sf_j) = 0$. Therefore, $Lf_j$ can't fire. Accordingly, $M(Ff_j) = 2$ can't be satisfied. So, $P_j$ can't be in the state of *eating*, i.e.

$$R_p(P_i, P_j) \rightarrow P_i.eating \wedge \neg P_j.eating \tag{1}$$

Similarly, when $P_j$ is in the state of *eating*, the firing condition of $E_i$ can't be satisfied. Accordingly, $P_i$ can't be in the state of *eating*, i.e.

$$R_p(P_i, P_j) \rightarrow \neg P_i.eating \wedge P_j.eating \tag{2}$$

When the neighboring philosopher $P_i$ and $P_j$ both are not in the sate of *eating*, obviously,

$$R_p(P_i, P_j) \rightarrow \neg P_i.eating \wedge \neg P_j.eating \tag{3}$$

From (1)(2)(3), we can find that

$$R_p(P_i, P_j) \rightarrow \neg(P_i.eating \wedge P_j.eating) \tag{4}$$

Because (4) has no other restrictive condition besides a premise that $P_i$ and $P_j$ are neighboring. Therefore, (4) holds *true* when the program is initializing. Therefore, (4) always holds *true*. Accordingly, property 2 holds *true*.

**Property 2.** $P_i.eating \longmapsto \neg P_i.eating \wedge f_i \wedge f_j \wedge T_f(f_i, f_j)$.

the dining philosopher will finish eating and free the holding forks.

**Proof:** Similar to property 1, property 3 can be rewritten as:

$$Sf_i \longmapsto \neg Sf_i \wedge f_i \wedge f_j \wedge T_f(f_i, f_j)$$

From property 1, $Sf_i$ ensures $St_i$ holds *true*. A philosopher is thinking implies that the philosopher is not in the state of *eating*, i.e. $St_i \rightarrow \neg Sf_i$. Accordingly, $Sf_i$ *ensures*$\neg Sf_i$ hold *true*. From figure 35, we can get an *occurrence sequence* $Sf_i[T_i > \{St_i, f_i, f_j\}$, so $T_i$'s firing causes $M(f_i) = M(f_j) = 1$. Accordingly, the two forks used by $P_i(T_f(f_i, f_j))$ are free, i.e.

$$Sf_i \; ensures \; \neg Sf_i \wedge f_i \wedge f_j$$

Consequently, property $P_i.eating \longmapsto \neg P_i.eating \wedge f_i \wedge f_j \wedge T_f(f_i, f_j)$ holds *true*.

**Property 3.** $P_i.hungriness \longmapsto P_i.eating \wedge (\neg f_i \wedge \neg f_j) \wedge T_f(f_i, f_j)$.

The hungry philosopher has opportunity to eat with two forks.

**Proof:** Similar to property 1, property 4 can be rewritten as

$$Sh_i \longmapsto Sf_i \wedge (\neg f_i \wedge \neg f_j) \wedge T_f(f_i, f_j)$$

From figure 34, we can get a process segment as figure.37 ($f_0 = f_5$).

In figure.37, ${}^\bullet E_i = \{Sh_i, Ff_i\}$ and $E_i^\bullet = \{Sf_i\}$, accordingly, there is *occurrence sequence*:

$$\{Sh_i, M(Ff_i) = 2\}[E_i > \{Sf_i\} \tag{5}$$

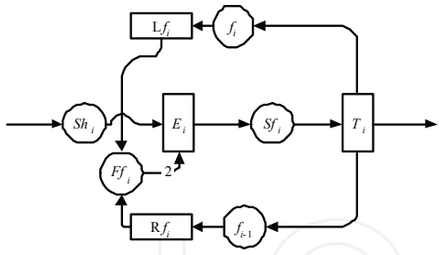



Fig. 37. The process segment of $P_i$'s eating

Moreover, there also is *occurrence sequence*:

$$\{f_i, f_j\}[\{Lf_i, Rf_i\} > \{M(Ff_i) = 2\} \tag{6}$$

therefore, $M(Ff_i) = 2$ cause $M(f_i) = 0$ and $M(f_j) = 0$.

According to property 1 and (5), $Sh_i$ ensures $Sf_i$ and $E_i$'s firing needs $M(Ff_i) = 2$. According to (6), $M(Ff_i) = 2$ causes $M(f_i) = 0$ and $M(f_j) = 0$, i.e. $E_i$'s firing causes $M(f_i) = 0$ and $M(f_j) = 0$. $f_i$ and $f_j$ are neighboring is the premise, so

$$Sh_i \; ensures \; Sf_i \wedge (\neg f_i \wedge \neg f_j) \wedge T_f(f_i, f_j)$$

i.e. $Sh_i \longmapsto Sf_i \wedge (\neg f_i \wedge \neg f_j) \wedge T_f(f_i, f_j)$ holds *true*.
Therefore, property 4 holds *true*.

**Proof of liveness**

$T$ is a set of transitions, $M$ is a set of system markings. Let $r = \{(m, m')|m, m' \in M \wedge t \in T : m[t > m'\}$, then

$$r^* = r^0 \cup r^1 \cup r^2 \cup \cdots = \bigcup_{i=0}^{\infty} r^i$$

is called as reachable relation, expression $m \; r^* \; m'$ denotes that marking $m$ can be changed to marking $m'$ after finite transitions fire.

If $m \in M, t \in T, m' \in M : mr^*m' \wedge m'[t >$, then the system is live.

If the system is live, and $m, m' \in M : mr^*m'$, then the system is circular.

From the above definition, we can prove the liveness of the dining problem.

**Proof:** The state of each philosopher changes in turn:

$$thinking \rightarrow hungry \rightarrow eating \rightarrow thinking \cdots$$

The state change of philosopher $P_i$ is controlled by a flow (directed circle): $\delta_i = \cdots T_i H_i E_i T_i \cdots$, The two neighbor philosophers must share one fork. $P_i$ requests the fork is controlled by (directed circle)

The right fork: $\varepsilon_{r_i} = f_i E_i S f_i T_i f_i$

The left fork: $\varepsilon_{l_i} = f_{i+1} E_i S f_i T_i f_{i+1}$

If $value(Pi) \neq nout$, the fire condition of the element in $T = \{T_i, H_i, E_i | i = 1 \cdots 5\}$ are $\{\cdots, Sf_i\}, \{\cdots, St_i\}$ and $\{\cdots, f_i, f_{i+1}, Sh_i\}$ respectively.

Given $m \in M$, the next firable transition $t$ must be an element of $T$, arbitrarily let $t = H_i$, obviously, if let $m' = \{\cdots, St_i\}$, the fire condition of $H_i$ can be satisfied and $H_i$ can fire.

Similarly, if let $t = T_i$ or $t = E_i$, there are $m' = \{\cdots, Sf_i\}$ or $m' = \{\cdots, f_i, f_{i+1}, Sh_i\}$ can satisfy the fire condition of $T_i$ or $E_i$ respectively. Thus,

$$t \in T, m' \in M : m'[t > \quad (1)$$

If $m' \subseteq m$, then $mr^*m'$ holds true obviously.

If $m' \nsubseteq m$, since there are directed circles $\delta_i$, $\varepsilon_{l_i}$ and $\varepsilon_{r_i}$, arbitrarily set $a_i$ be a place of the directed circles $\delta_i$, $\varepsilon_{l_i}$ or $\varepsilon_{r_i}$ in marking $m'$, and $m'(a_i) > 0$.

If $a_i \in \delta_i$. Because $\delta_i$ is a directed circle, therefore $\delta_i$ has a place $b_i$ in marking $m$, and there must be a path from $b_i$ to $a_i$ (the token in $b_i$ will arrive at $a_i$ at some time). Otherwise, the philosopher $P_i$ can't be in one state of three states in m. So, $b_i r^* a_i$.

Similarly, if $a_i \in \varepsilon_{l_i}$, $b_j \in \varepsilon_{l_i}$ , $b_j r^* a_i$; if $a_i \in \varepsilon_{r_i}$,$\exists b_k \in \varepsilon_{r_i}$ , $b_k r^* a_i$;

Hence, if $m' \nsubseteq m$, $m$ can reach $m'$. i.e.

$$mr^*m' \quad (2)$$

From (3.1) and (3.2), we conclude that the system is live.

Because there exists directed circles $\delta_i$, $\varepsilon_{l_i}$ and $\varepsilon_{r_i}$, the tokens must flow along the circles, so the markings of system change circularly. We can conclude that any marking $m$ can reach any other marking $m'$, i.e., $m, m' \in M : mr^*m'$. Hence, the system is circular.

## 8. References


Bjorner, D., Jones, C. B., Airchinnigh, M. M. a. & Neuhold, E. J. (1987). *VDM '87 VDM – A Formal Method at Work*, Vol. 252, Springer-Verlag, Germany. ø.

Breuer, P. T. & Lano, K. C. (1999). Creating specifications from code: Reverse-engineering techniques, *Journal of Software Maintenance: Research and Practice* 3: 145–162.

Girault, C. & Valk, R. (2002). *Petri Nets for Systems Engineering:a guide to modeling,verification,and applications*, Springer-Verlag.

Harel, D. (1987). *Statecharts: A Visual Formalism for Complex Systems*. Sci. Comput. Programming 8.

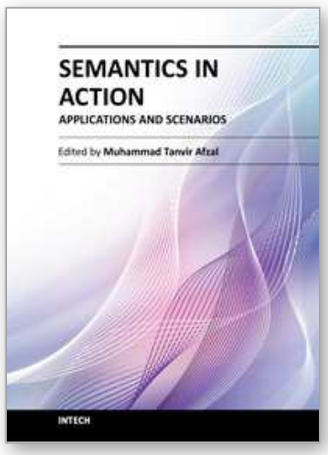
Jensen, K., Kristensen, L. M. & Wells, L. (2007). Coloured petri nets and cpn tools for modelling and validation of concurrent systems, *Int. J. Softw. Tools Technol. Transf.* 9(3): 213–254.

Khedker, U., Sanyal, A. & Karkare, B. (2009). *Data Flow Analysis: Theory and Practice*, CRC Press (Taylor and Francis Group).

Reisig, W. (1985). *Petri Nets, an Introduction*, EATCS Monographs in Theroetical Computer Science, Springer. EATCS Monographs in Theroetical Computer Science.

Spivey, J. M. (1998). The z notation: A reference manual.

Woodcock, J. C. P. & Davies, J. (1996). *Using Z-Specification,Refinement,and Proof*, Prentice-Hall.




**Semantics in Action - Applications and Scenarios**
Edited by Dr. Muhammad Tanvir Afzal

ISBN 978-953-51-0536-7
Hard cover, 266 pages
**Publisher** InTech
**Published online** 25, April, 2012
**Published in print edition** April, 2012

The current book is a combination of number of great ideas, applications, case studies, and practical systems in the domain of Semantics. The book has been divided into two volumes. The current one is the second volume which highlights the state-of-the-art application areas in the domain of Semantics. This volume has been divided into four sections and ten chapters. The sections include: 1) Software Engineering, 2) Applications: Semantic Cache, E-Health, Sport Video Browsing, and Power Grids, 3) Visualization, and 4) Natural Language Disambiguation. Authors across the World have contributed to debate on state-of-the-art systems, theories, models, applications areas, case studies in the domain of Semantics. Furthermore, authors have proposed new approaches to solve real life problems ranging from e-Health to power grids, video browsing to program semantics, semantic cache systems to natural language disambiguation, and public debate to software engineering.

**How to reference**
In order to correctly reference this scholarly work, feel free to copy and paste the following:

Guofu Zhou and Zhuomin Du (2012). Visualizing Program Semantics, Semantics in Action - Applications and Scenarios, Dr. Muhammad Tanvir Afzal (Ed.), ISBN: 978-953-51-0536-7, InTech, Available from: http://www.intechopen.com/books/semantics-in-action-applications-and-scenarios/visualizing-program-semantics

INTECH
open science | open minds

**InTech Europe**
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

**InTech China**
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

IntechOpen