Deadlock

CS 2SD3

Ryszard Janicki

Department of Computing and Software, McMaster University, Hamilton, Ontario, Canada

Deadlock

Concepts:	system deadlock: no further progress four necessary & sufficient conditions
Models:	deadlock - no eligible actions
Practice:	blocked threads

Aim: deadlock avoidance - to design systems where deadlock cannot occur.

Deadlock: four necessary and sufficient conditions

- **Serially reusable resources:**

  the processes involved share resources which they use under mutual exclusion.

- **Incremental acquisition:**

  processes hold on to resources already allocated to them while waiting to acquire additional resources.

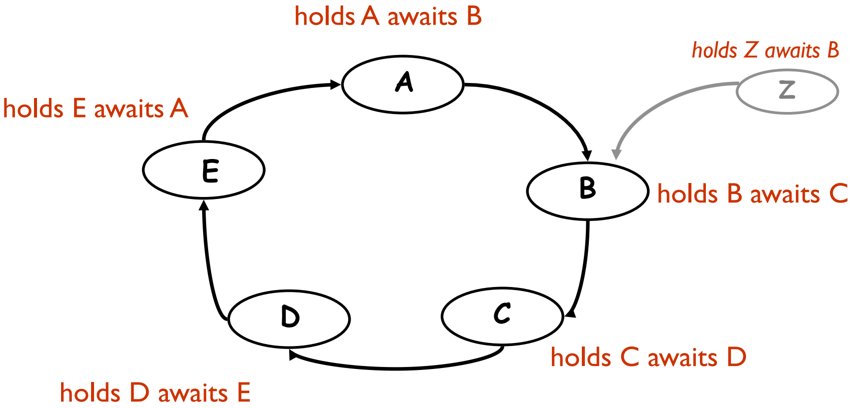
- **No pre-emption:**

  once acquired by a process, resources cannot be pre-empted (forcibly withdrawn) but are only released voluntarily.

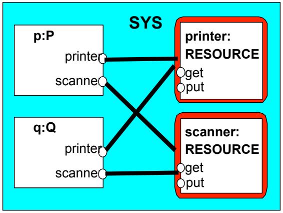
- **Wait-for cycle:**

  a circular chain (or cycle) of processes exists such that each process holds a resource which its successor in the cycle is waiting to acquire.

Wait-for cycle



Deadlock may arise from the parallel composition of interacting processes.



```
RESOURCE = (get->put->RESOURCE).
P = (printer.get->scanner.get
      ->copy
      ->printer.put->scanner.put
      ->P).
Q = (scanner.get->printer.get
      ->copy
      ->scanner.put->printer.put
      ->Q).
||SYS = (p:P||q:Q
      ||{p,q}::printer:RESOURCE
      ||{p,q}::scanner:RESOURCE
      ).
```

Deadlock Trace?

Avoidance?

$$p : P = (p.printer.get_{\bullet} \rightarrow p.scanner.get \rightarrow p.copy \rightarrow p.printer.put \rightarrow p.scanner.put \rightarrow p : P)$$

$$q : Q = (q.scanner.get_{\bullet} \rightarrow q.printer.get \rightarrow q.copy \rightarrow q.scanner.put \rightarrow q.printer.put \rightarrow q : Q)$$

$$\underbrace{\{p, q\} :: printer : RESOURCE}_{pqpR} = (p.printer.get_{\bullet} \rightarrow p.printer.put \rightarrow pqpR \mid q.printer.get \rightarrow q.printer.put \rightarrow pqpR)$$

$$\underbrace{\{p, q\} :: scanner : RESOURCE}_{pqsR} = (p.scanner.get \rightarrow p.scanner.put \rightarrow pqsR \mid q.scanner.get_{\bullet} \rightarrow q.scanner.put \rightarrow pqsP)$$

Deadlock sequence: $p.printer.get \rightarrow q.scanner.get$

- $\bullet$ - denote states where processes deadlock

A Possible Solutions

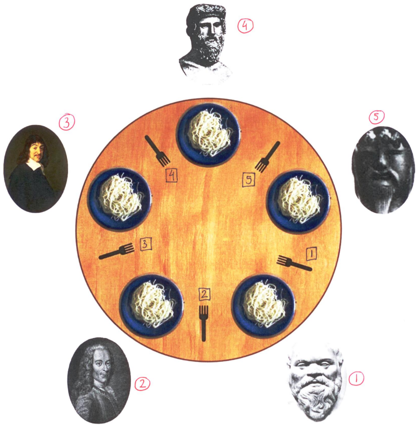
- Acquire resources in the same order, i.e. printers always before scanners.
- Timeout:

```
P          = (printer.get-> GETSCANNER),
GETSCANNER = (scanner.get->copy->printer.put
                              ->scanner.put->P
             |timeout -> printer.put->P
             ).
Q          = (scanner.get-> GETPRINTER),
GETPRINTER = (printer.get->copy->printer.put
                              ->scanner.put->Q
             |timeout -> scanner.put->Q
             ).
```

- No deadlock but the sequence:
  $printer.get \rightarrow timeout \rightarrow printer.put \rightarrow$
  can be repeated infinite number of times!
  NOBODY COPIES ANYTHING!

Dining Philosophers

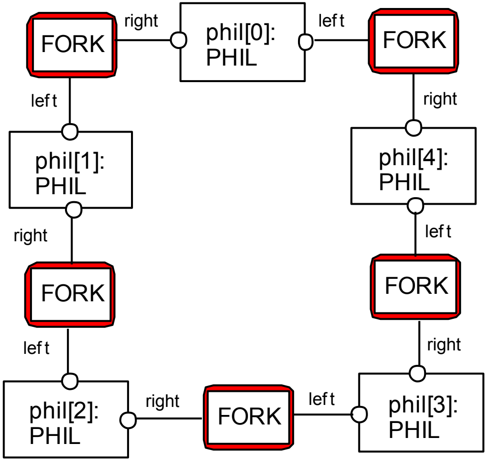
- Five philosophers sit around a circular table. Each philosopher spends his life alternately thinking and eating. To eat, a philosopher needs two forks, but unfortunately there are only five forks on the circulr table and each philosopher is only allowed to use the two forks nearest to him.



Dining Philosophers - model structure diagram

Each FORK is a shared resource with actions **get** and **put**.

When hungry, each PHIL must first get his right and left forks before he can start eating.



Hungry, Simple Minded Philosophers

- $i \oplus 1 = \mathit{if}\ i < 5\ \mathit{then}\ i+1\ \mathit{else}\ 1$

$$
\begin{aligned}
& FORK = (get \to put \to FORK) \\
& PHIL = (think \to right.get \to left.get \to eat \to right.put \to \\
& \qquad\qquad left.put \to PHIL) \\
& \| \ DINERS(N=5) = forall[i:1..N] \\
& \qquad\qquad (phil[i] : PHIL \ \| \ \{phil[i].right, phil[i \oplus 1].left\} :: FORK)
\end{aligned}
$$

- More intuitively (for get_j^i, put_j^i, i - philosopher number, j - fork number):

$$
\begin{aligned}
& FORK_1 = (get_1^1 \to put_1^1 \to FORK_1 \mid get_1^5 \to put_1^5 \to FORK_1) \\
& FORK_2 = (get_2^2 \to put_2^2 \to FORK_2 \mid get_2^1 \to put_2^1 \to FORK_2) \\
& FORK_3 = (get_3^3 \to put_3^3 \to FORK_3 \mid get_3^2 \to put_3^2 \to FORK_3) \\
& FORK_4 = (get_4^4 \to put_4^4 \to FORK_4 \mid get_4^3 \to put_4^3 \to FORK_4) \\
& FORK_5 = (get_5^5 \to put_5^5 \to FORK_5 \mid get_5^4 \to put_5^4 \to FORK_5) \\
& PHIL_1 = (think_1 \to get_1^1 \to get_2^1 \to eat_1 \to put_1^1 \to put_2^1 \to PHIL_1) \\
& PHIL_2 = (think_2 \to get_2^2 \to get_3^2 \to eat_2 \to put_2^2 \to put_3^2 \to PHIL_2) \\
& PHIL_3 = (think_3 \to get_3^3 \to get_4^3 \to eat_3 \to put_3^3 \to put_4^3 \to PHIL_3) \\
& PHIL_4 = (think_4 \to get_4^4 \to get_5^4 \to eat_4 \to put_4^4 \to put_5^4 \to PHIL_4) \\
& PHIL_5 = (think_5 \to get_5^5 \to get_1^5 \to eat_5 \to put_5^5 \to put_1^5 \to PHIL_5) \\
& \| \ DINERS = (FORK_1 \ \| \ \dots \ \| \ FORK_5 \ \| \ PHIL_1 \ \| \ \dots \ \| \ PHIL_5)
\end{aligned}
$$

Hungry, Simple Minded Philosophers

- **Obvious deadlock! Everyone picks right fork.**

$Trace_1 =$
$phil.1.think \rightarrow phil.1.right.get \rightarrow$
$phil.2.think \rightarrow phil.2.right.get \rightarrow$
$phil.3.think \rightarrow phil.3.right.get \rightarrow$
$phil.4.think \rightarrow phil.4.right.get \rightarrow$
$phil.5.think \rightarrow phil.5.right.get$

$think_1 \rightarrow get_1^1 \rightarrow$
$think_2 \rightarrow get_2^1 \rightarrow$
$think_3 \rightarrow get_3^1 \rightarrow$
$think_4 \rightarrow get_4^1 \rightarrow$
$think_5 \rightarrow get_5^1$

What if not 'Simple Minded'?

$FORK = (get \rightarrow put \rightarrow FORK)$

$PHIL = THINK$
$THINK = (think \rightarrow$
$\quad (right.get \rightarrow left.get \rightarrow EAT \mid left.get \rightarrow right.get \rightarrow EAT))$

$EAT = (eat \rightarrow$
$\quad (right.put \rightarrow left.pt \rightarrow THINK \mid left.put \rightarrow right.put \rightarrow THINK$

$\| \; DINERS(N = 5) = forall[i : 1..N]$
$\qquad (phil[i] : PHIL \parallel \{phil[i].right, phil[i \oplus 1].left\} :: FORK)$

- Unfortunately a freedom of choosing either right or left fork does not solve the problem. The same trace leads to a deadlock. However in "real" implementation, it will make it happen less often.

Still 'Simple Minded' but not so 'Hungry'

$FORK = (get \to put \to FORK)$
$PHIL = THINK$
$THINK = (think \to right.get \to$
$\quad (left.get \to EAT \mid giveup \to right.put \to THINK))$
$EAT = (eat \to right.put \to left.put \to THINK)$
$\| DINERS(N = 5) = forall[i : 1..N]$
$\quad (phil[i] : PHIL \parallel \{phil[i].right, phil[i \oplus 1].left\} :: FORK)$

- **There is no deadlock now!**
  $Trace_1 \to phil.i.giveup \to phil.i.right.put \to \ldots$
- However we might get:
  $Trace_1 \to Trace_2 \to$ and so on,
  where: $Trace_2 = phil.1.giveup \to phil.1.right.put \to$
  $phil.2.giveup \to phil.2.right.put \to$
  $phil.3.giveup \to phil.3.right.put \to$
  $phil.4.giveup \to phil.4.right.put \to$
  $phil.5.giveup \to phil.5.right.put \to$
- No philosopher will ever eat!
  Starvation!

'Hungry' and 'Asymmetrically Simple Minded', or 'Some Discipline Added'

- Philosophers 1, 3 and 5 always perform '*left.get* $\rightarrow$ *right.get*', while 2 and 4 always perform '*right.get* $\rightarrow$ *left.get*'.

$$
\begin{aligned}
&FORK = (get \rightarrow put \rightarrow FORK)\\
&PHIL = (when(i = 1 \vee i = 3 \vee i = 5)\ think \rightarrow left.get \rightarrow\\
&\qquad right.get \rightarrow eat \rightarrow left.put \rightarrow right.put \rightarrow PHIL\\
&\qquad |\ when(i = 2 \vee i = 4)\ think \rightarrow right.get \rightarrow\\
&\qquad left.get \rightarrow eat \rightarrow right.put \rightarrow left.put \rightarrow PHIL)\\
&\|\ DINERS(N = 5) = forall[i : 1..N]\\
&\qquad (phil[i] : PHIL \parallel \{phil[i].right, phil[i \oplus 1].left\} :: FORK)
\end{aligned}
$$

- **Works! Neither deadlock nor starvation.**

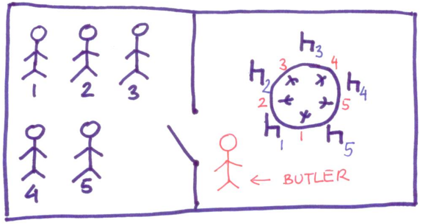
Asymmetrically Simple Minded Philosophers

- Notation: for get_j^i, put_j^i, i - philosopher number, j - fork number

$$
\begin{aligned}
&FORK_1 = (get_1^1 \rightarrow put_1^1 \rightarrow FORK_1 \mid get_1^5 \rightarrow put_1^5 \rightarrow FORK_1)\\
&FORK_2 = (get_2^2 \rightarrow put_2^2 \rightarrow FORK_2 \mid get_2^1 \rightarrow put_2^1 \rightarrow FORK_2)\\
&FORK_3 = (get_3^3 \rightarrow put_3^3 \rightarrow FORK_3 \mid get_3^2 \rightarrow put_3^2 \rightarrow FORK_3)\\
&FORK_4 = (get_4^4 \rightarrow put_4^4 \rightarrow FORK_4 \mid get_4^3 \rightarrow put_4^3 \rightarrow FORK_4)\\
&FORK_5 = (get_5^5 \rightarrow put_5^5 \rightarrow FORK_5 \mid get_5^4 \rightarrow put_5^4 \rightarrow FORK_5)\\
&PHIL_1 = (think_1 \rightarrow get_2^1 \rightarrow get_1^1 \rightarrow eat_1 \rightarrow put_2^1 \rightarrow put_1^1 \rightarrow PHIL_1)\\
&PHIL_2 = (think_2 \rightarrow get_2^2 \rightarrow get_3^2 \rightarrow eat_2 \rightarrow put_2^2 \rightarrow put_3^2 \rightarrow PHIL_2)\\
&PHIL_3 = (think_3 \rightarrow get_4^3 \rightarrow get_3^3 \rightarrow eat_3 \rightarrow put_4^3 \rightarrow put_3^3 \rightarrow PHIL_3)\\
&PHIL_4 = (think_4 \rightarrow get_4^4 \rightarrow get_5^4 \rightarrow eat_4 \rightarrow put_4^4 \rightarrow put_5^4 \rightarrow PHIL_4)\\
&PHIL_5 = (think_5 \rightarrow get_1^5 \rightarrow get_5^5 \rightarrow eat_5 \rightarrow put_1^5 \rightarrow put_5^5 \rightarrow PHIL_5)\\
&\| \, DINERS = (FORK_1 \parallel \ldots \parallel FORK_5 \parallel PHIL_1 \parallel \ldots \parallel PHIL_5)
\end{aligned}
$$

‘Hungry’ and ’Simple Minded’ but outside control, i.e. ‘Butler’

- **No more than 4 philosophers are sitting at the table.**



$FORK = (get \rightarrow put \rightarrow FORK)$
$PHIL = (think \rightarrow sitdown \rightarrow right.get \rightarrow left.get \rightarrow eat \rightarrow$
$\quad right.put \rightarrow left.put \rightarrow getup \rightarrow PHIL)$
$BUTLER(K = 4) = COUNT[0]$
$COUNT[i : 1..4] = (when(i < K)\ sitdown \rightarrow COUNT[i + 1] \mid$
$\quad getup \rightarrow COUNT[i - 1]]$
$\| DINERS(N = 5) = (forall[i : 1..N]$
$\quad (phil[i] : PHIL \parallel \{phil[i].right, phil[i \oplus 1].left\} :: FORK)$
$\quad \parallel \quad \underbrace{\{phil[i : ..N]\}}_{\{phil[1], phil[2], phil[3], phil[4], phil[5]\}} \quad :: BUTLER(K = 4))$

‘Butler’ Solution

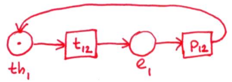
- ‘Butler’ solution works. No deadlock and no starvation.
- *FORK*'s are *passive* processes (monitors), hence they always can be presented as:
  $FORK = (get \rightarrow put \rightarrow FORK)$
- *PHILOSOPHER*'s are *active* processes.

A Solution with Simultaneity

- No philosopher is allowed to grab one fork only, he must take both left and right at the same time if they are available.
- Modeling simultaneity is **not** *natural* in FSP approach, it is possible but looks artificial.
- Modeling simultaneity is *natural* when Petri nets are used.

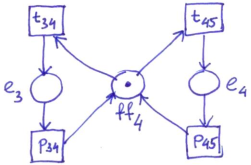
Individual Philosophers and Free Forks as Petri Nets

- Philosopher No. 1:



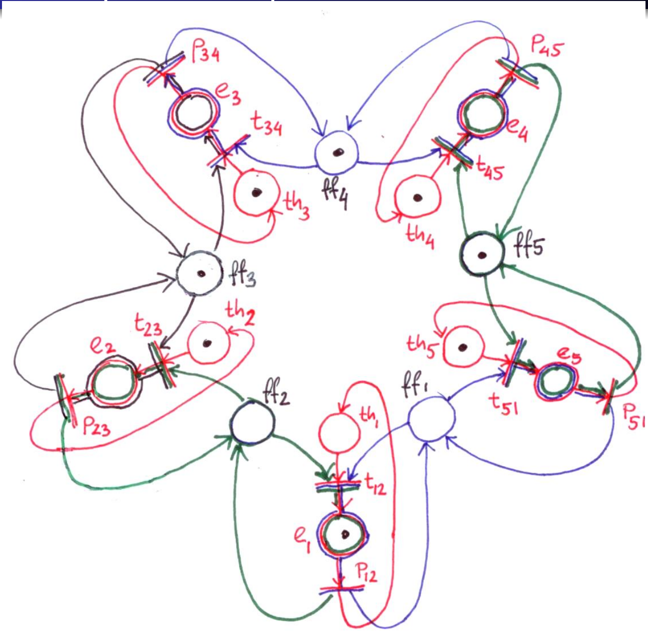
tt_1 - Philosopher No.1 thinks
e_1 - Philosopher No.1 eats
t_{12} - Philosopher No.1 takes up forks 1 and 2
p_{12} - Philosopher No.1 puts down forks 1 and 2

- Free Fork No. 4:



ff_4 - Fork No. 4 is on the table
e_3 - Philosopher No. 3 eats using Fork No. 4
t_{34} - Fork No. 4 is taken by Philosopher No.3
p_{34} - Fork No. 4 is put down Philosopher No.3
e_4 - Philosopher No. 4 eats using Fork No. 4
t_{45} - Fork No. 4 is taken by Philosopher No. 4
p_{45} - Fork No. 4 is put down Philosopher No. 4

Dining Philosophers Composed

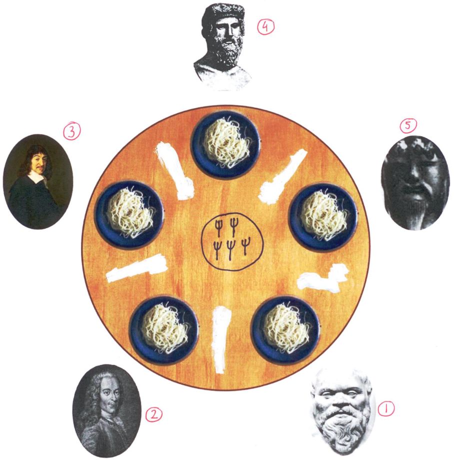


A Solution with Simultaneity for FSP

$$FORK = (take.right \rightarrow put.right \rightarrow FORK \mid$$
$$take.left \rightarrow put.left \rightarrow FORK)$$
$$PHIL = (think \rightarrow takeboth \rightarrow eat \rightarrow putboth \rightarrow PHIL)$$

$$\| \; DINERS(N = 5) = forall[i : 1..N]$$
$$(phil[i] : PHIL \| \{phil[i].right, phil[i \oplus 1].left\} :: FORK))$$
$$/\{takeboth.1/take.right.1, takeboth.1/take.left.2,$$
$$takeboth.2/take.right.2, takeboth.2/take.left.3,$$
$$takeboth.3/take.right.3, takeboth.3/take.left.4,$$
$$takeboth.4/take.right.4, takeboth.4/take.left.5,$$
$$takeboth.5/take.right.5, takeboth.5/take.left.1,$$
$$putboth.1/put.right.1, putboth.1/put.left.2,$$
$$putboth.2/put.right.2, putboth.1/put.left.3,$$
$$putboth.3/put.right.3, putboth.3/put.left.4,$$
$$putboth.4/put.right.4, putboth.4/put.left.5,$$
$$putboth.5/put.right.5, putboth.5/put.left.1\}$$

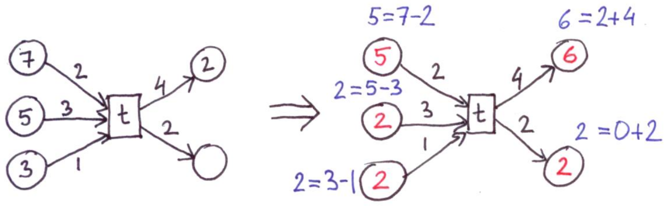
Different Dining Philosophers and Some Limits of Process Algebras



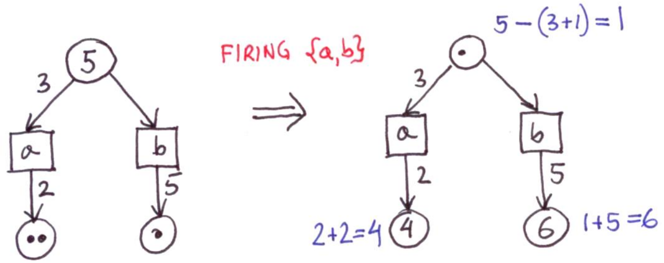
- All forks in one bowl. Forks are not distinguishable and philosophers pick them randomly.

Place/Transitions Nets (P/T-Nets)

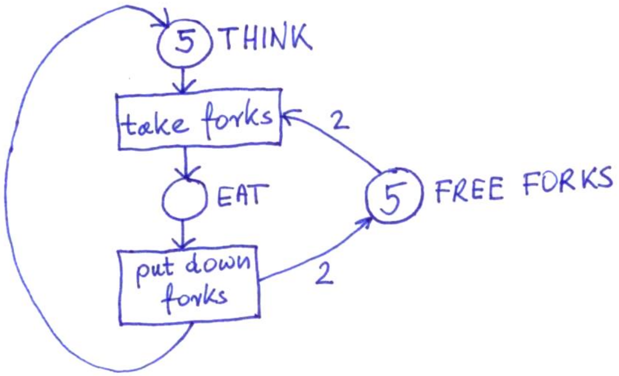
- Firing rules:



- Different kind of simultaneity:

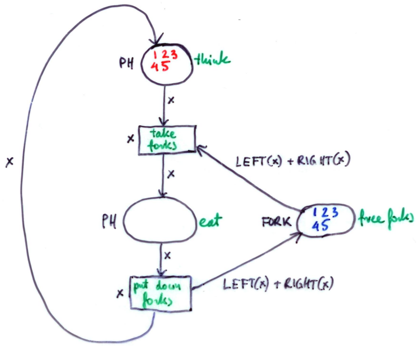


P/T-Nets Solution to 2nd Dining Philosophers Problem



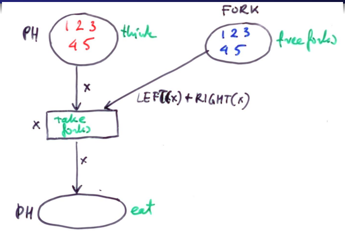
- It does not work for the original Dining Philosophers Problem.
- Both philosophers and forks are represented by tokens.
- State machines represent *generic behaviours*.
- Impossible to model directly with FSP.

Coloured Petri Nets



colour PH = *with ph*1 | *ph*2 | *ph*3 | *ph*4 | *ph*5
colour Fork = *with f*1 | *f*2 | *f*3 | *f*4 | *f*5
LEFT : *PH* → *FORK*,   *RIGHT* : *PH* → *FORK*
var x : *PH*
fun LEFT x = *case of ph*1 ⇒ *f*2 | *ph*2 ⇒ *f*3 | *ph*3 ⇒ *f*4 |
*ph*4 ⇒ *f*5 | *ph*5 ⇒ *f*1
fun RIGHT x = *case of ph*1 ⇒ *f*1 | *ph*2 ⇒ *f*2 | *ph*3 ⇒ *f*3 |
*ph*4 ⇒ *f*4 | *ph*5 ⇒ *f*5

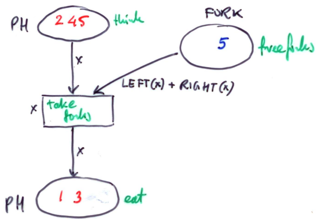
Firing



$$\Downarrow$$

Firing occurrence: $(take\ forks, \underbrace{x = ph1}_{binding}) + (take\ forks, \underbrace{x = ph3}_{binding})$

$$\Downarrow$$



Multisets (or Bags)

- A multiset m, over a non-empty and finite set S is a function $m : S \to \mathbb{N} = \{0, 1, 2, \ldots\}$
- $m(s)$ is the number of appearances of s in m.
- notation: M is usually represented by:

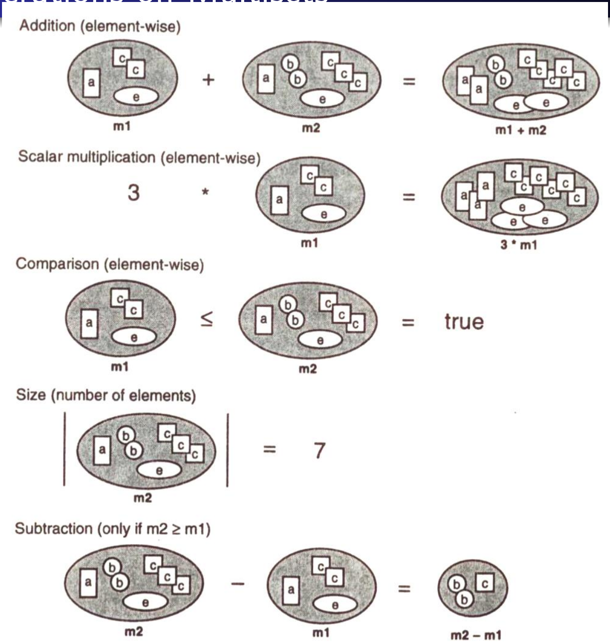
$$\sum_{s \in S} m(s)s$$

$S = \{a, b, c, d, e\}$,
$m(a) = 3, m(b) = 1, m(c) = 0, m(d) = 183, m(e) = 4$

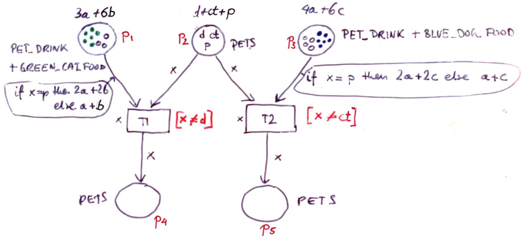
$$m = 3a + b + 183d + 4e$$

- $s \in m \iff m(s) \neq 0$
- $m(s)$ is a *coefficient*
- the *empty multiset* $m = \emptyset \iff m(s)$ for each $s \in S$.

Some Operations on Multisets


Addition (element-wise)
m1 + m2 = m1 + m2
Scalar multiplication (element-wise)
3 * m1 = 3 * m1
Comparison (element-wise)
m1 ≤ m2 = true
Size (number of elements)
| m2 | = 7
Subtraction (only if m2 ≥ m1)
m2 – m1 = m2 – m1
a
b
c
e

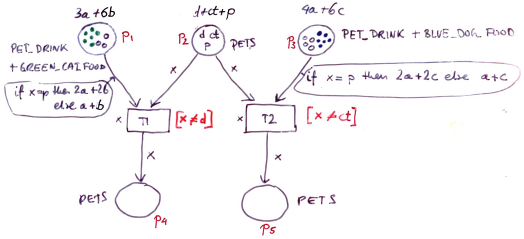
colour PET_DRINK = with a;
colour GREEN_CAT_FOOD = with b;
colour BLUE_DOG_FOOD = with c;
colour PETS = with dog | cat | pig; (pig eats both cat food and dog food)
var x : PETS; (in the drawing: dog=d, cat =ct, pig = p)



3a +6b
d+ct+p
4a +6c
PET_DRINK + GREEN_CAT_FOOD
P1
P2
d ct p
PETS
P3
PET_DRINK + BLUE_DOG_FOOD
x
x
if x=p then 2a+2b else a+b
if x=p then 2a+2c else a+c
x
T1
[x≠d]
x
T2
[x≠ct]
x
x
PETS
P4
PETS
P5

• In place p_1 we have 6 green cat food servings and 3 drinks (no dog food as it not allowed here, it is allowed in place p_3)
• Firing transition $T1$ corresponds to allow cat or pig or both to eat. The cat eats 1 serving of cat food and 1 drink while the pig eats 2 servings of food and 2 drinks.
• If both cat and pig eat and drink, 3 drinks and 3 servings of cat food disappear from place p_1, and p, ct disappear from place p_2.
• Similarly for places p_3, p_5 and transition $T2$.

colour PET_DRINK = with a;
colour GREEN_CAT_FOOD = with b;
colour BLUE_DOG_FOOD = with c;
colour PETS = with dog | cat | pig; (pig eats both cat food and dog food)
var x : PETS; (in the drawing: dog=d, cat =ct, pig = p)

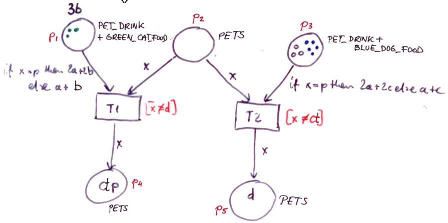


3a +6b
d+ct+p
4a +6c
P1
P2
PETS
P3
PET_DRINK + BLUE_DOG_FOOD
PET_DRINK + GREEN_CAT_FOOD
if x=p then 2a+2b else a+b
if x=p then 2a+2c else a+c
x
T1
[x≠d]
T2
[x≠ct]
PETS
P4
PETS
P5

⇓

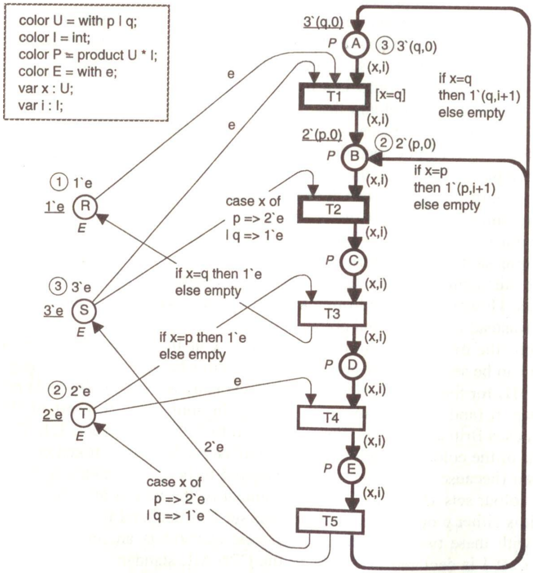
Firing occurrence:$(T1, x = c) + (T1, x = p) + (T2, x = d)$
Interpretation:All three pets eat.

⇓

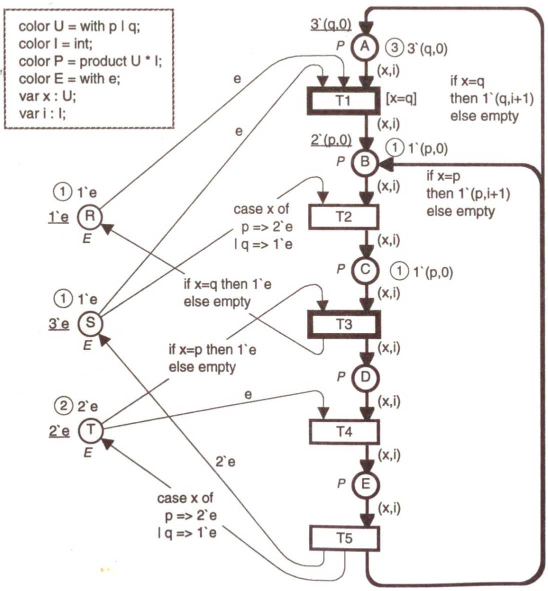


3b
P1
PET_DRINK + GREEN_CAT_FOOD
P2
PETS
P3
PET_DRINK + BLUE_DOG_FOOD
if x=p then 2a+2b else a+b
if x=p then 2a+2c else a+c
x
T1
[x≠d]
T2
[x≠ct]
ct p
P4
PETS
d
P5
PETS

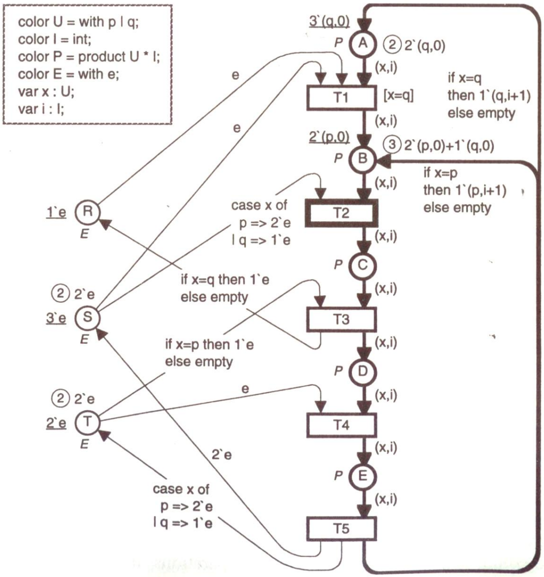
Some resource allocation syatem in its initial marking M_0



M_1 reachable from M_0 by $(T2, \langle x = p, i = 0 \rangle)$


```
color U = with p | q;
color I = int;
color P = product U * I;
color E = with e;
var x : U;
var i : I;
```

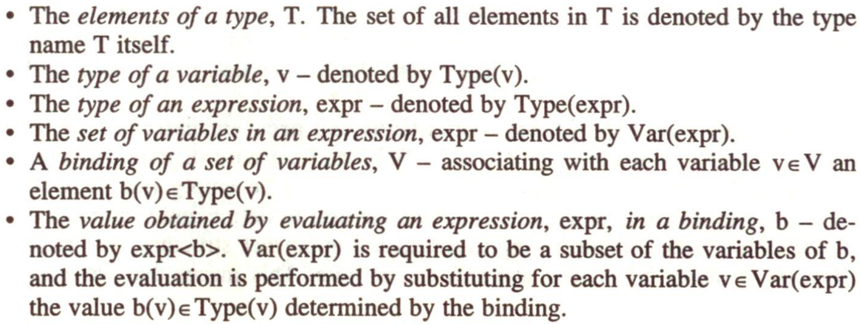
M_2 reachable from M_0 by $(T2, \langle x = q, i = 0\rangle)$



A *Coloured Petri Net* is a tuple: $N = (P, T, A, \Sigma, C, N, E, G, I)$ where:

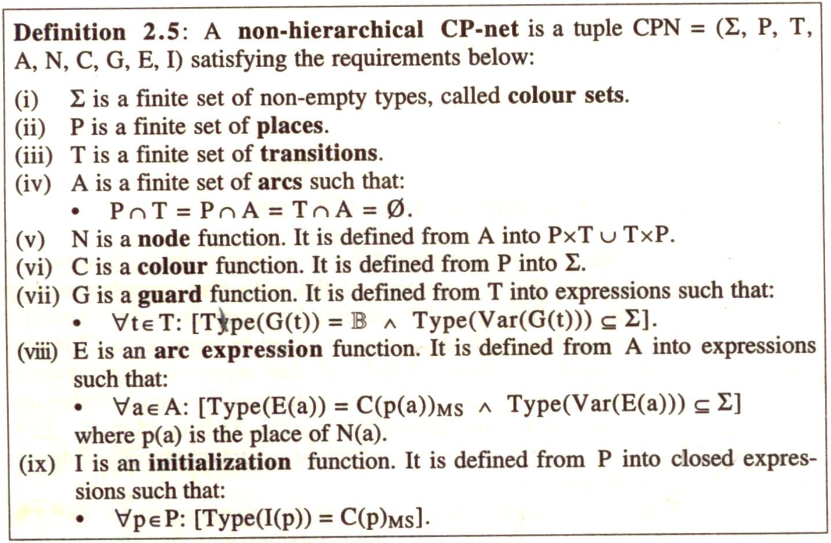
- P is a set of places and T is a set of transitions.
- T is a set of transitions.
- A is a set of arcs
- In CPNs sets of places, transitions and arcs are pairwise disjoint $P \cap T = P \cap A = T \cap A = \emptyset$
- Σ is a set of color sets defined within CPN model. This set contains all possible colors, operations and functions used within CPN.
- C is a colour function. It maps places in P into colors in Σ.
- N is a node function. It maps A into $(P \times T) \cup (T \times P)$.
- E is an arc expression function. It maps each arc $a \in A$ into the expression e. The input and output types of the arc expressions must correspond to type of nodes the arc connected to.
- G is a guard function. It maps each transition $t \in T$ into guard expression g. The output of the guard expression should evaluate to Boolean value true or false.
- I is an initialization function. It maps each place p into an initialization expression i. The initialization expression must evaluate to multiset of tokens with a color corresponding to the

Some Concepts Needed



- The *elements of a type*, T. The set of all elements in T is denoted by the type name T itself.
- The *type of a variable*, v – denoted by Type(v).
- The *type of an expression*, expr – denoted by Type(expr).
- The *set of variables in an expression*, expr – denoted by Var(expr).
- A *binding of a set of variables*, V – associating with each variable v ∈ V an element b(v) ∈ Type(v).
- The *value obtained by evaluating an expression*, expr, *in a binding*, b – denoted by expr<b>. Var(expr) is required to be a subset of the variables of b, and the evaluation is performed by substituting for each variable v ∈ Var(expr) the value b(v) ∈ Type(v) determined by the binding.

'Official' Formal Definition of Petri Nets



Definition 2.5: A **non-hierarchical CP-net** is a tuple CPN = (Σ, P, T, A, N, C, G, E, I) satisfying the requirements below:

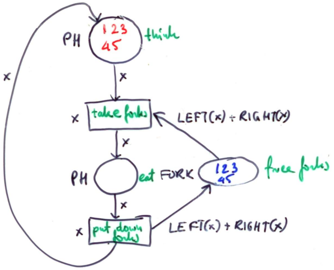
(i) Σ is a finite set of non-empty types, called **colour sets**.
(ii) P is a finite set of **places**.
(iii) T is a finite set of **transitions**.
(iv) A is a finite set of **arcs** such that:
- $P \cap T = P \cap A = T \cap A = \emptyset$.

(v) N is a **node** function. It is defined from A into $P \times T \cup T \times P$.
(vi) C is a **colour** function. It is defined from P into Σ.
(vii) G is a **guard** function. It is defined from T into expressions such that:
- $\forall t \in T: [\mathrm{Type}(G(t)) = \mathbb{B} \wedge \mathrm{Type}(\mathrm{Var}(G(t))) \subseteq \Sigma]$.

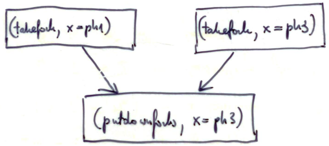
(viii) E is an **arc expression** function. It is defined from A into expressions such that:
- $\forall a \in A: [\mathrm{Type}(E(a)) = C(p(a))_{MS} \wedge \mathrm{Type}(\mathrm{Var}(E(a))) \subseteq \Sigma]$

where p(a) is the place of N(a).
(ix) I is an **initialization** function. It is defined from P into closed expressions such that:
- $\forall p \in P: [\mathrm{Type}(I(p)) = C(p)_{MS}]$.

Behaviours



Sequence:
$(take\ forks, x = ph1)(take\ forks, x = ph3)(putdown\ forks, x = ph3)$
Step-sequence:
$\{(take\ forks, x = ph1)(take\ forks, x = ph3)\}\{(putdown\ forks, x = ph3)\}$
Partial order:


(takeforks, x=ph1)
(takeforks, x=ph3)
(putdownforks, x=ph3)