Modeling and Verifying Distributed Systems with Petri Nets :

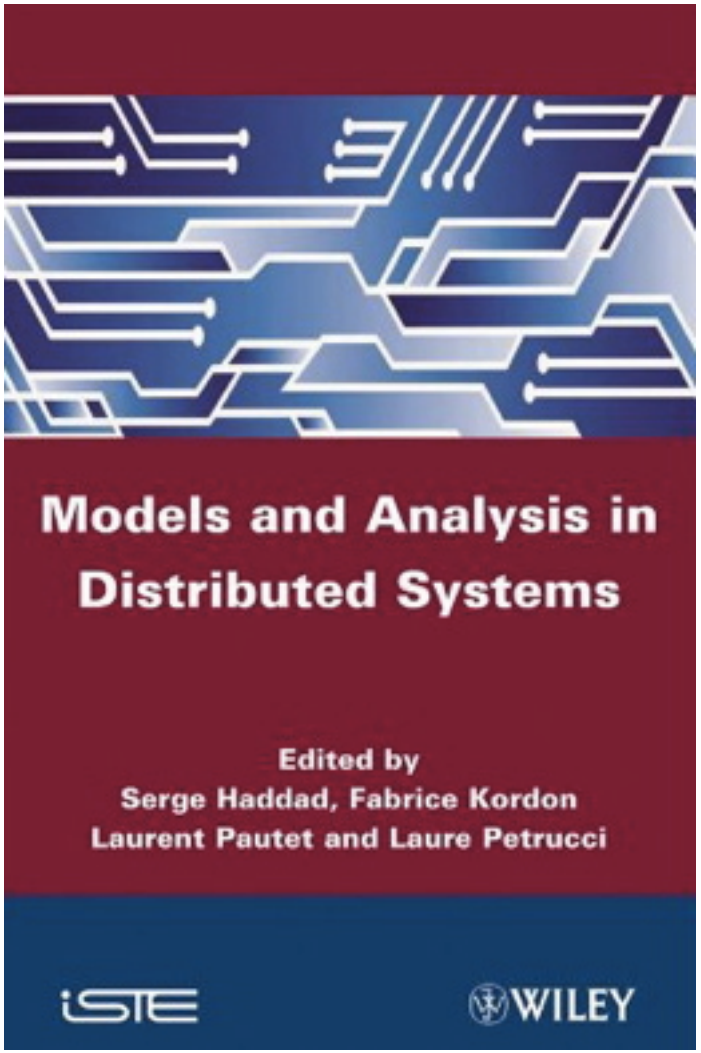
Place-Transition Nets and temporal logic

Souheib Baarir, Fabrice Kordon

First.last@lip6.fr

Labrotaoire d'Informatique de Paris 6

References (1/2)



References (2/2)

[1] E. Clarke, O.Grumberg, and A. Peled. Model Checking. MIT Press, 2000.

[2] M. Diaz. R´eseaux de Petri, Modèles fondamentaux. Traité IC2, série Informatique et systèmes d'information. Hermes Science, June 2001.

[3] C. Girault and R. Valk. Petri Nets for Systems Engineering. Springer Verlag - ISBN: 3-540-41217-4, 2003.

[4] S. Haddad, F. Kordon, and L. Petrucci, editors. Méthodes Formelles pour les Systèmes Répartis et Coopératifs. Traité IC2 - Hermes, 2006.

[5] S. Haddad, F. Kordon, L. Pautet, and L. Petrucci, editors. Models and Analysis in Distributed Systems. Wiley, 2011.

Context : Formal verification

Formal Verification is the act of *proving or disproving* the correctness of intended *systems* with respect to a certain *formal specification (property),* using *formal methods of mathematics.*

Known approaches:

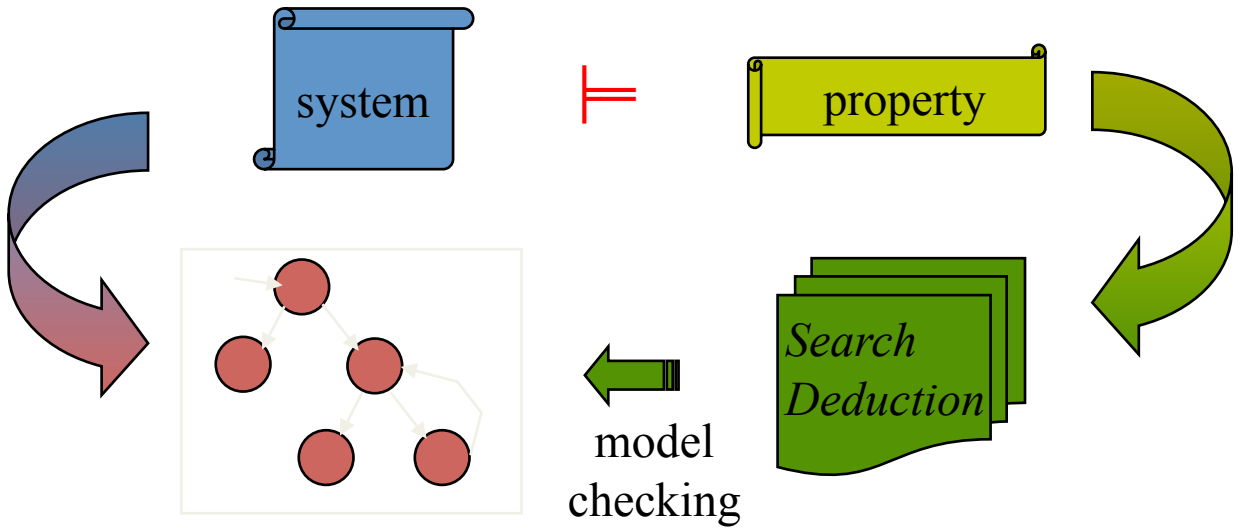
- Deductive verification.
- Program derivation.
- Model checking.

Context : Model Checking

Model Checking : consists of a *systematically exhaustive exploration* of the mathematical model of the ***system,*** to prove that it satisfies (or not) a given ***property (specification).***

Which abstraction for the system ?
From which formalism do we get the desired abstraction ?

Which kind of properties ?
How to express these properties ?



To represent discrete systems…

- At each instant, the system has a given ***state.*** In a given state, some ***atomic propositions*** are satisfied when others are not:
  - *description of the value of each global variable,*
  - *the calling stack(s),*
  - *the program counter(s),*
  - *the contents of the communicating channels, etc.*

- From a given state, a system can reach another state by executing a ***transition***.

Kripke structure

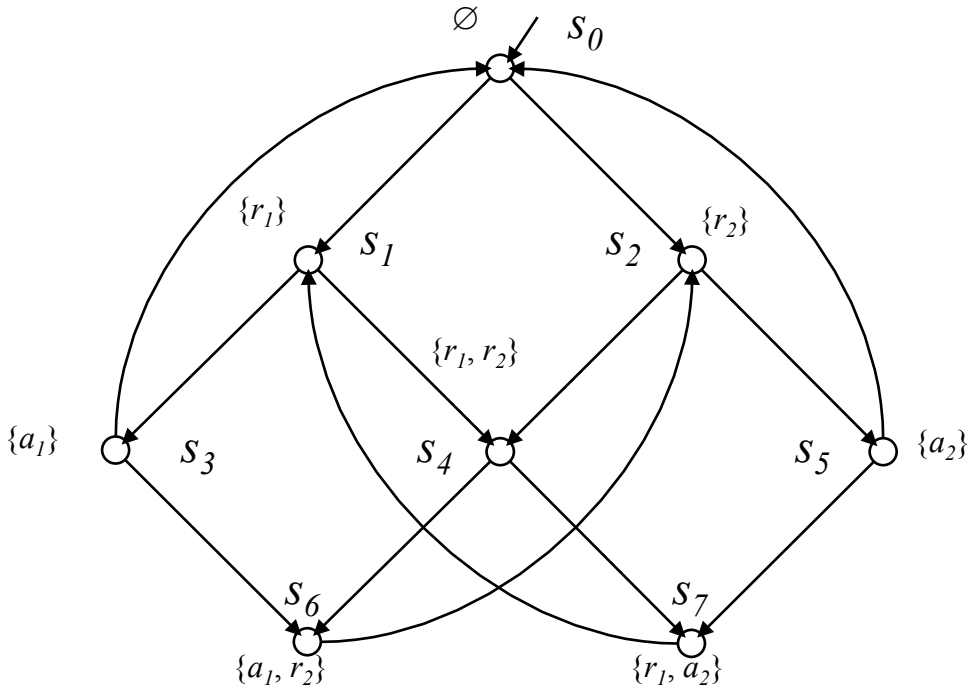
- When a system can reach only a *finite* number of states, the previous requirements are exactly captured by *Kripke structures*.
- Let AP be a finite set of atomic propositions over the system.
- A Kripke structure is a tuple $\langle S, \ell, \rightarrow, s_0 \rangle$ where
  - S is the finite set of states
  - $\ell : S \mapsto 2^{AP}$ is the labeling function
  - $\rightarrow \subseteq S \times S$ is the transition relation
  - $s_0 \in S$ is the initial state

Example

Let $AP = \{r_1, r_2, a_1, a_2\}$

Possible interpretation :

- r_i : *request to the critical section by Process i and*
- a_i : *critical section access by Process i.*



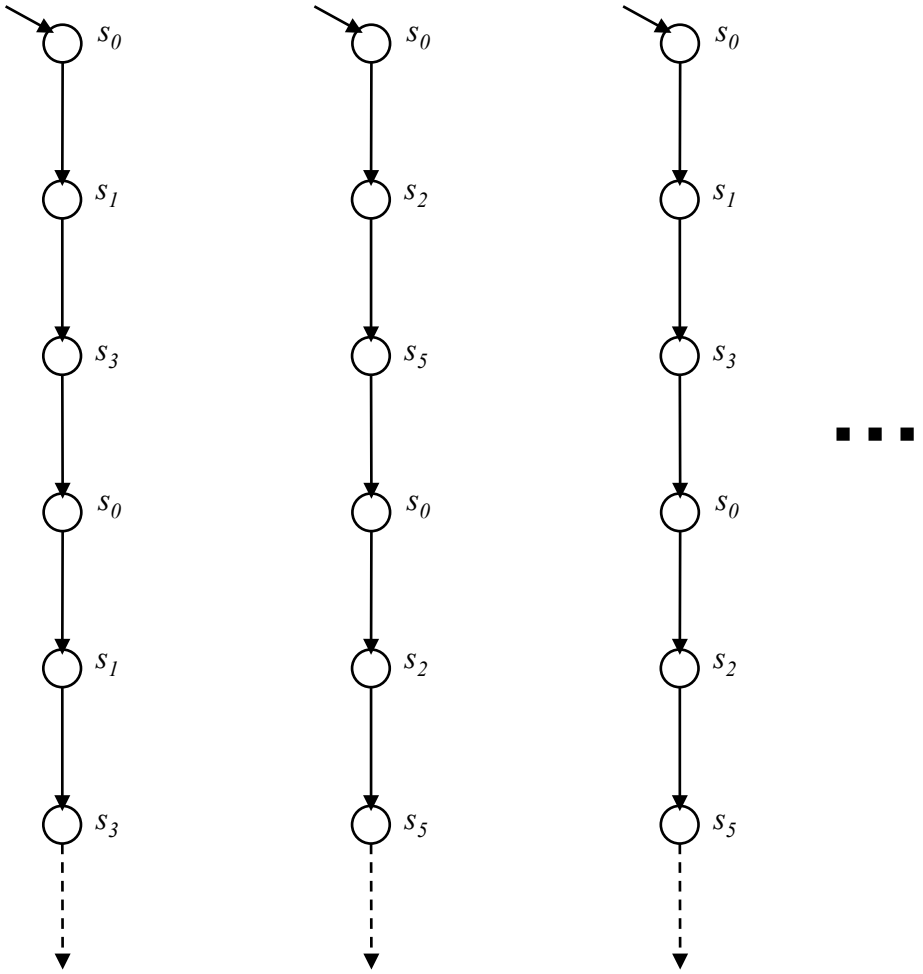
Semantics of a system

- Two particular semantics are generally admitted:
  - the ***linear time*** semantics,
  - the ***branching time*** semantics.

- For the linear time semantics, an execution of the system is an infinite path in the Kripke structure.

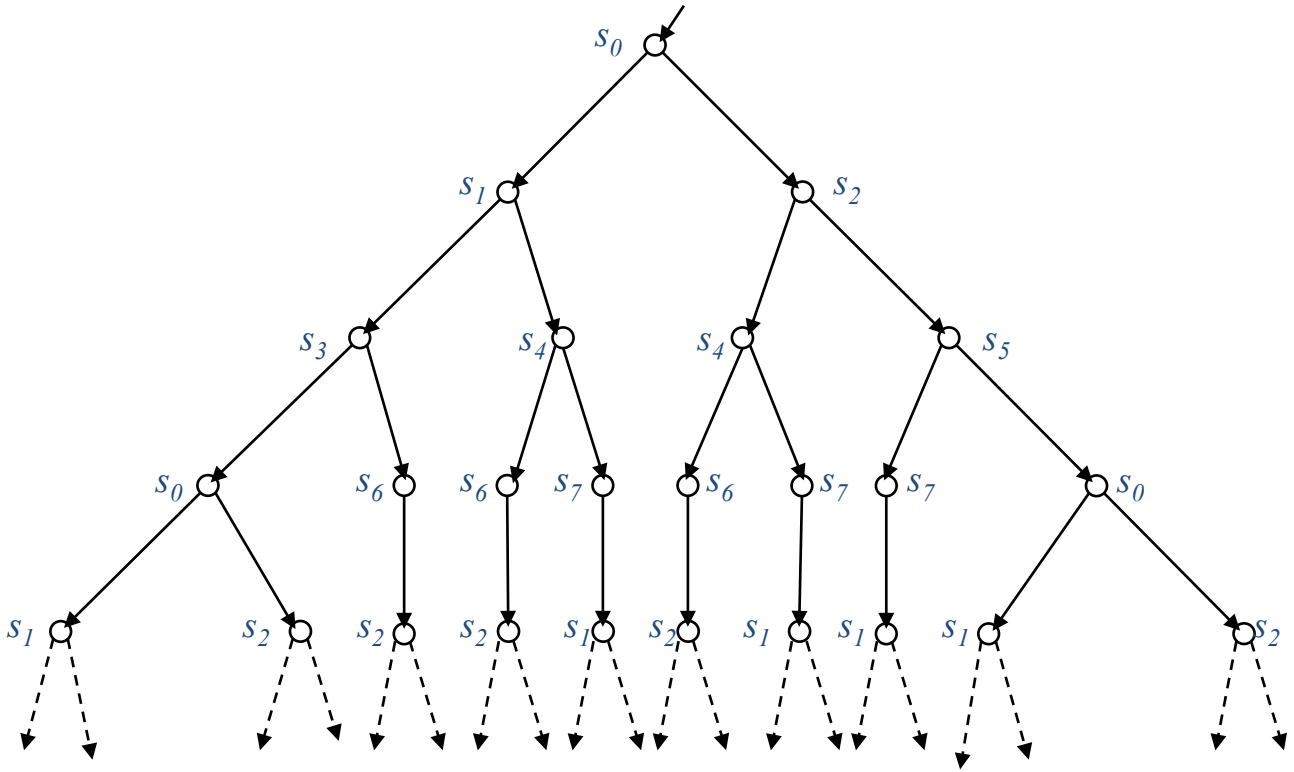
- For the branching time ones, the execution of the system is represented by the underlying infinite tree of the Kripke structure.

Linear time semantics



The linear time semantics of a Kripke structure corresponds to the (possibly infinite) set of all infinite paths

Branching time semantics



Its branching time semantics corresponds to the underlying infinite tree of the graph rooted in the initial state

Notations (1/2)

- Let $\langle S, \ell, \rightarrow, s_0 \rangle$ be a Kripke structure.
  - $(s,s') \in \rightarrow$ is noted $s \rightarrow s'$:
    - s is a predecessor of s', and s' is a successor of s
  - S^* denotes the finite words on S.
  - S^ω denotes the infinites words on S.
  - $w = s_0 \ldots s_n$ is a path of length n if we have $s_i \rightarrow s_{i+1}$ for all i, $0 \leq i < n$.

Notations (2/2)

– $s \rightarrow^{*} s'$ means a path from s *to* s'

– $s \rightarrow^{+} s'$ means a path from s *to* s' *of* positive length

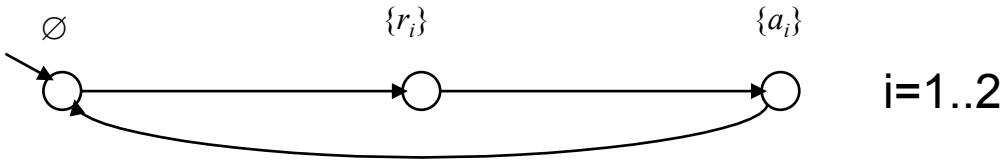
– $\rho = s_0\, s_1 \ldots$ is a run if we have $s_i \rightarrow s_{i+1}$ for all i, $0 \leq i$.

– $\rho(i)$ *the* i^{th} *state of* ρ *and,*
ρ^i *the suffix of* ρ *starting at* $\rho(i)$

– For a run $\rho = s_0\, s_1 \ldots$,
$\ell(\rho)$ is the ω-word $\ell(s_0)\,\ell(s_1)\ldots$ over 2^{AP}

Kripke structure generators

- A Kripke structure can be generated from many kinds of formalisms.
  - A program,
  - An algebraic composition of processes,
  - A synchronized product of automata,



$$\forall s \in S, \neg(a_1 \wedge a_2)$$

  - **A bounded Petri net (i.e. with a finite number of markings)**
  - Etc.

Outlines

- Modeling the system
  - Petri Net Formalism
- Specifying of the system
  - Linear Time Logic (LTL)
  - Computation Time Logic (CTL)

Modeling the system:

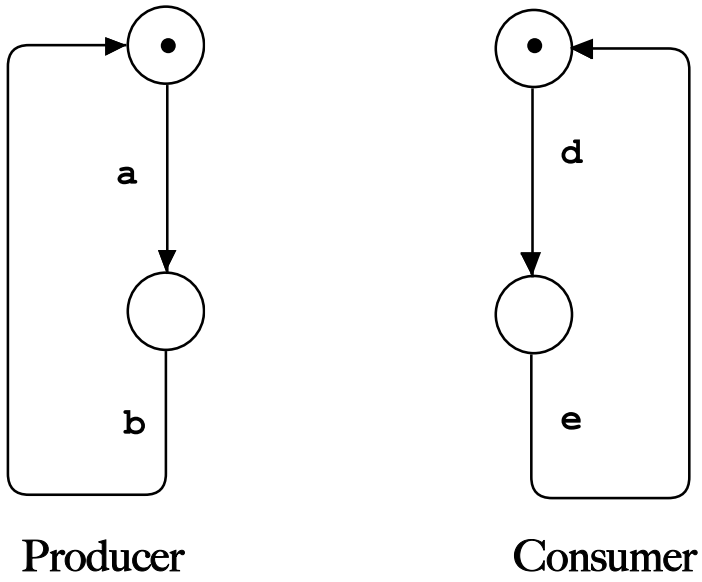
The case of Finite and Discrete Event Systems

From Automata to Petri Nets

- ***Problem*: modelling a producer/consumer asynchronous system with infinite buffer.**
  - If the buffer is not empty, the consumer can consume independently from the state of the producer.

a : produce a message

b : put a message



d : consume a message

e : treat a message

Semantics of the model

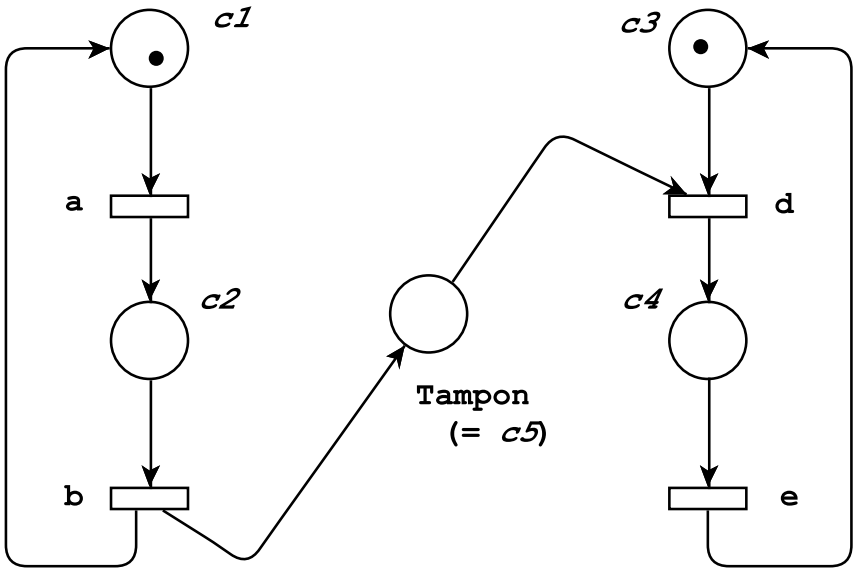
- Possible Sequences:
  - The sequence (de)* is acceptable…!
  - The consumer can execute indefinitely…!

- Need of conditions on certain actions:
  - add of rectangles on arcs, to synchronise actions.
  - Represent environment elements: the buffer.

Condition/Event Model

- Two types of nodes :
  - Conditions are represented by cercles (*Boolean holders*).
  - Events are represented by rectangles.

- The Producer/Consumer model



Notations

- We note :
  - **•n** the set of predecessors of a node n
  - **n•** the set of successors of a node n

- If n is an event, we call
  - **•n** the set of pre-conditions of n
  - **n•** the set of post-conditions of n

- **Examples**:
  - •d = {c3, c5} the pre-conditions of d are c_3 et c_5
  - b• = {c1, c5} the post-conditions of b are c_1 et c_5

Semantics of a C/E Model (1/2)

- A step in C/E model is defined by:

$$C_1 \xrightarrow{\lambda} C_2$$

where, C_1 and C_2 are condition subsets and λ is an event e, such that:

1) $\bullet\lambda \subseteq C_1$
2) $C_2 = (C_1 \setminus \bullet\lambda) \cup \lambda\bullet$

- **Example**: the Producer/Consumer model

$C_0 = \{ c_1, c_3 \}$ $\quad \bullet a = \{ c_1 \}$ $\quad a\bullet = \{ c_2 \}$

$$\bullet a \subseteq C_0 \Rightarrow C_0 \xrightarrow{a} C_1 \text{ With } C_1 = (C_0 \setminus \bullet a) \cup a\bullet$$

$$C_1 = (\{ c_1, c_3 \} \setminus \{ c_1 \}) \cup \{ c_2 \} = \{ c_2, c_3 \}$$

Semantics of a C/E model (2/2)

- A sequence of a C/E model is defined by:

$$w = \lambda_1 \lambda_2 \ldots \lambda_n$$

such that for each $i \in \{1,\ldots,n\}$,

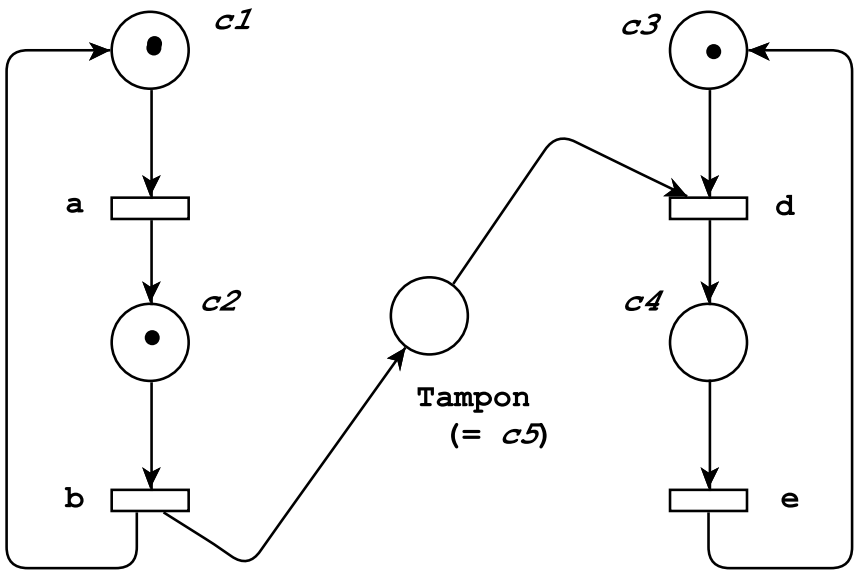
$$c_{i-1} \xrightarrow{\lambda_i} c_i$$

is a step in the C/E system.

- C_0 is the subset of conditions representing the initial state.

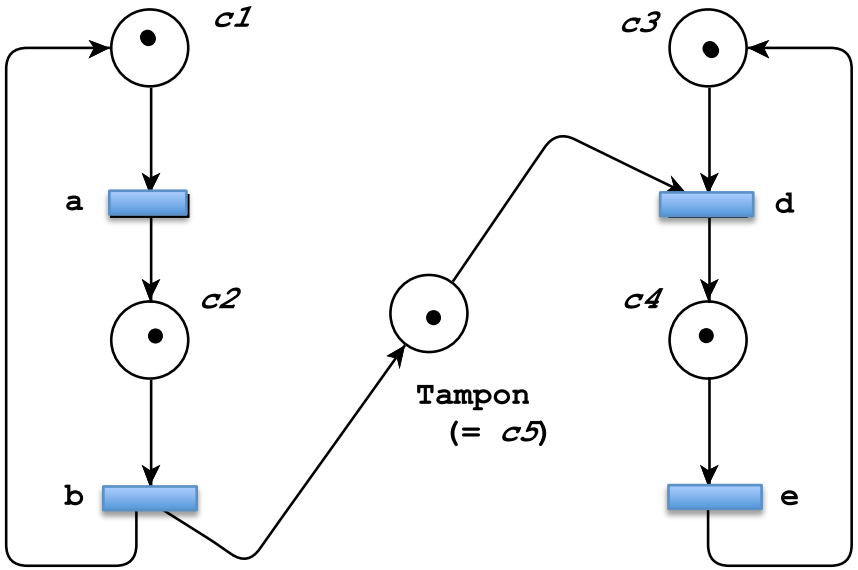
Example de sequence

- **ab** is a sequence of the C/E model of the Producer/Consumer.



Limits of the C/E model

- Producer / Consumer:



$C_o \xrightarrow{ab} C_2$

avec $C_2 = \{c_1 , c_3 , c_5 \}$

$C_2 \xrightarrow{ab} C_3$

avec $C_3 = \{c_1 , c_3 , c_5 \}$

$C_3 \xrightarrow{de} C_4$

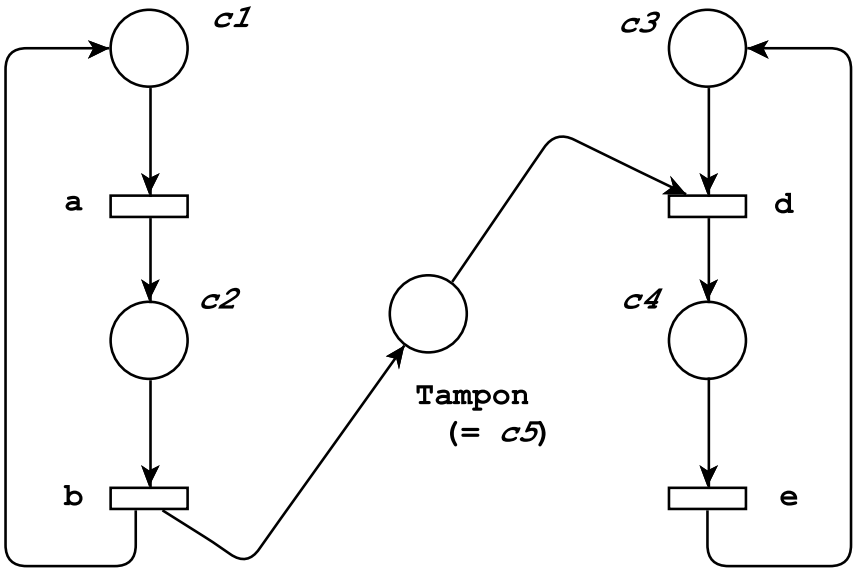
mais $C_4 = \{c_1 , c_3\}$, donc, $C_4 \not\xrightarrow{d}$

2 productions *vs.* 1 consumption!

Must use a non Boolean model : Petri Nets (PN)

PN model of the Producer/Consumer

- Representation:



Conditions → Places

Events → Transitions

- State of the PN = Marking $M = \sum_{p \in P} M(p).p$

where, $M(p)$ is the number of tokens in place p.

Example : after the execution of sequence ***abab***, $\mathbf{M = 1.c_1 + 1.c_3 + 2.\ c_5}$

Formal definition of a PN

- A PN **R** is tuple <P, T, Pre, Post> such that:
  - **P**: is a finite set of places ($P \neq \emptyset$)
  - **T**: is a finite set of transitions ($T \neq \emptyset$, $P \cap T = \emptyset$)

  - **Pre** : $P \times T \rightarrow IN$

    $Pre(p, t) = n \ \ (n > 0) \ \ \Leftrightarrow$

    the firing of a transition t is conditioned by the presence of n resources in p

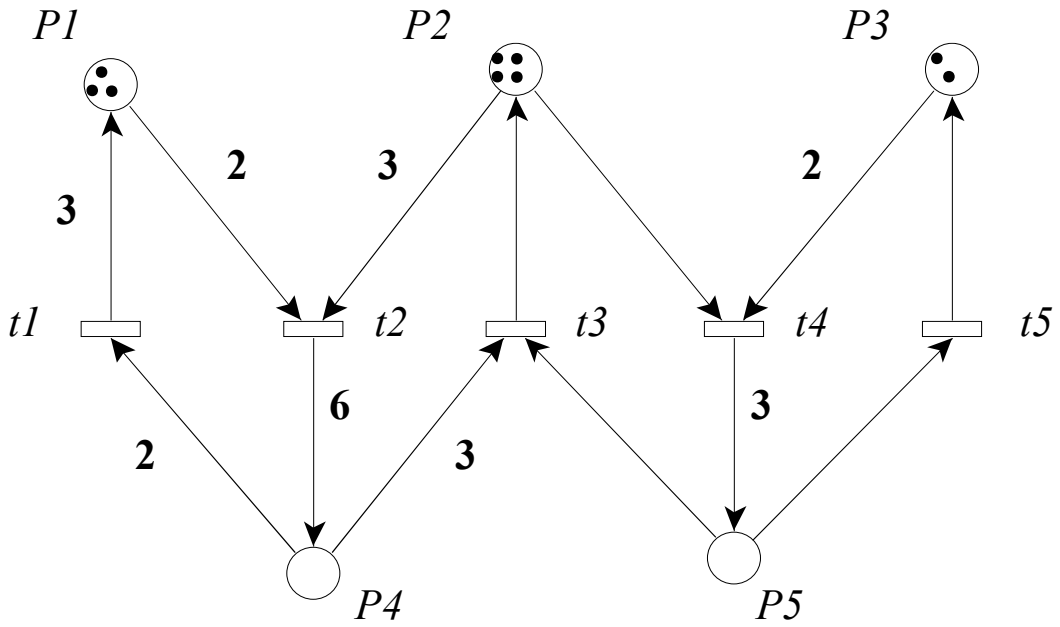
  - **Post** : $P \times T \rightarrow IN$

    $Post(p, t) = n \ \ (n > 0) \ \ \Leftrightarrow$

    the firing of t produce n resources in p

- A marked net <R, M_0> has an initial marking $M_0 \in IN^P$

A Petri Net - Example



The initial marking is M_0

$$M_0 = \begin{bmatrix} 3 \\ 4 \\ 2 \\ 0 \\ 0 \end{bmatrix}$$

Firing a transition

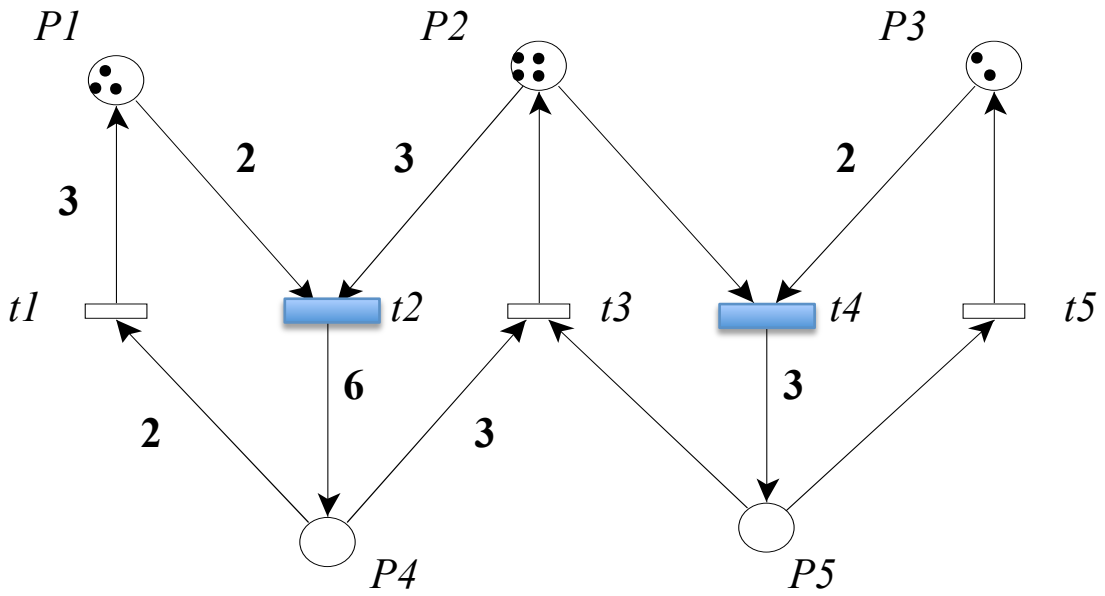
◆**The firing rule** :

- t ∈ T is **enabled** in M iff: ∀ p ∈ P, M(p) ≥ Pre(p,t).

- if t is enabled, then its firing leads to the marking M' :
  ∀ p ∈ P, M'(p) = M(p) - Pre(p,t) + Post(p,t)

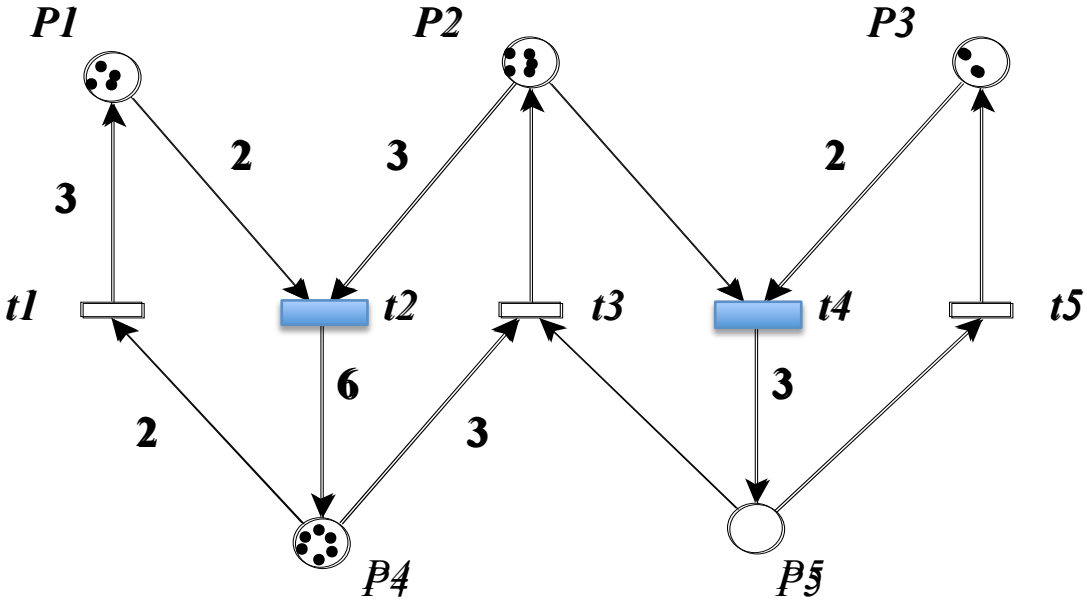
◆**The firing of t is noted : M [t> M'.**

Enabling- example



t_2 and t_4 are enabled in M_0, we note $\mathbf{M_0}\ [t_2>$ and $\mathbf{M_0}\ [t_4>$

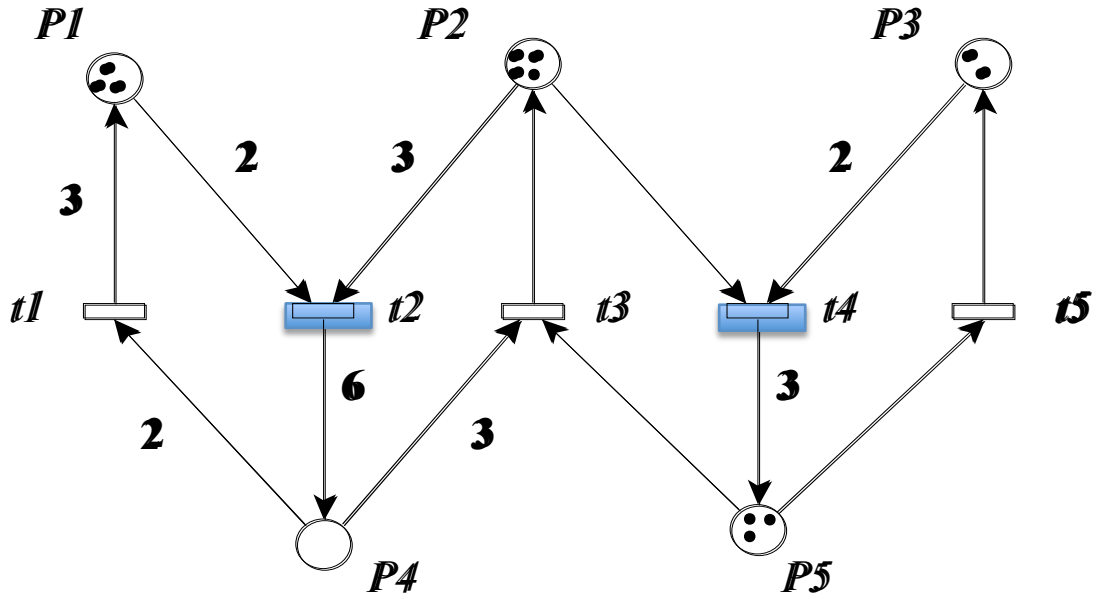
Firing – exemple (1/2)



The firing of t_2 from M_0 produces the marking $M_1 = \begin{bmatrix} 1 \\ 1 \\ 2 \\ 6 \\ 0 \end{bmatrix}$

We note, $\mathbf{M_0}\ [\mathbf{t_2} > \mathbf{M_1}$

Firing – exemple (2/2)

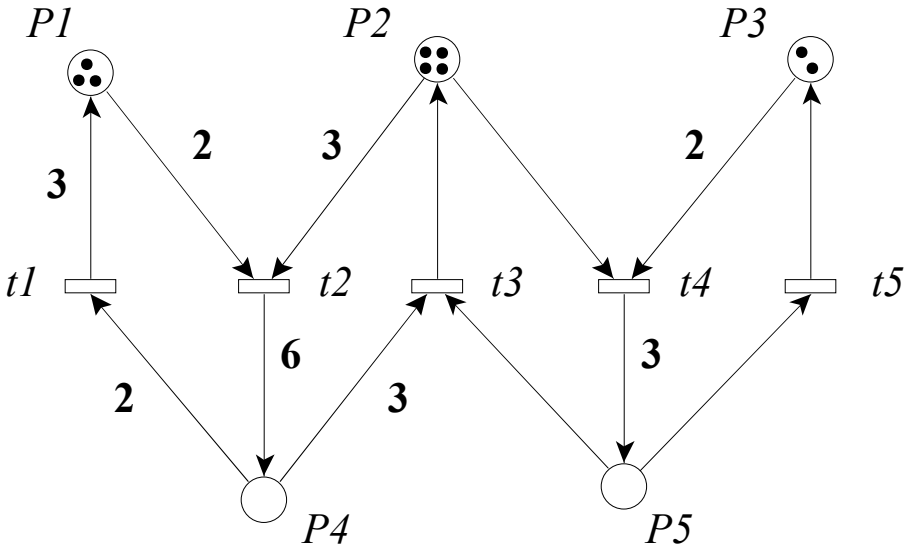


The firing of t_4 from M_0 produces the marking $M_2 = \begin{bmatrix} 3 \\ 3 \\ 0 \\ 0 \\ 3 \end{bmatrix}$

We note $\mathbf{M_0\ [t_4 > M_2}$

Firing Sequence

A firing sequence from M_0 to M_n is a word $t_0 \ldots t_{n-1}$ such that it exists a set of markings $M_1, \ldots, M_{n-1}$ where : $M_0\ [t_0> M_1 \ \ldots \ M_{n-1}\ [t_{n-1}> M_n$



$t_2\ t_4\ t_3$ is a firing sequence starting from M_0:

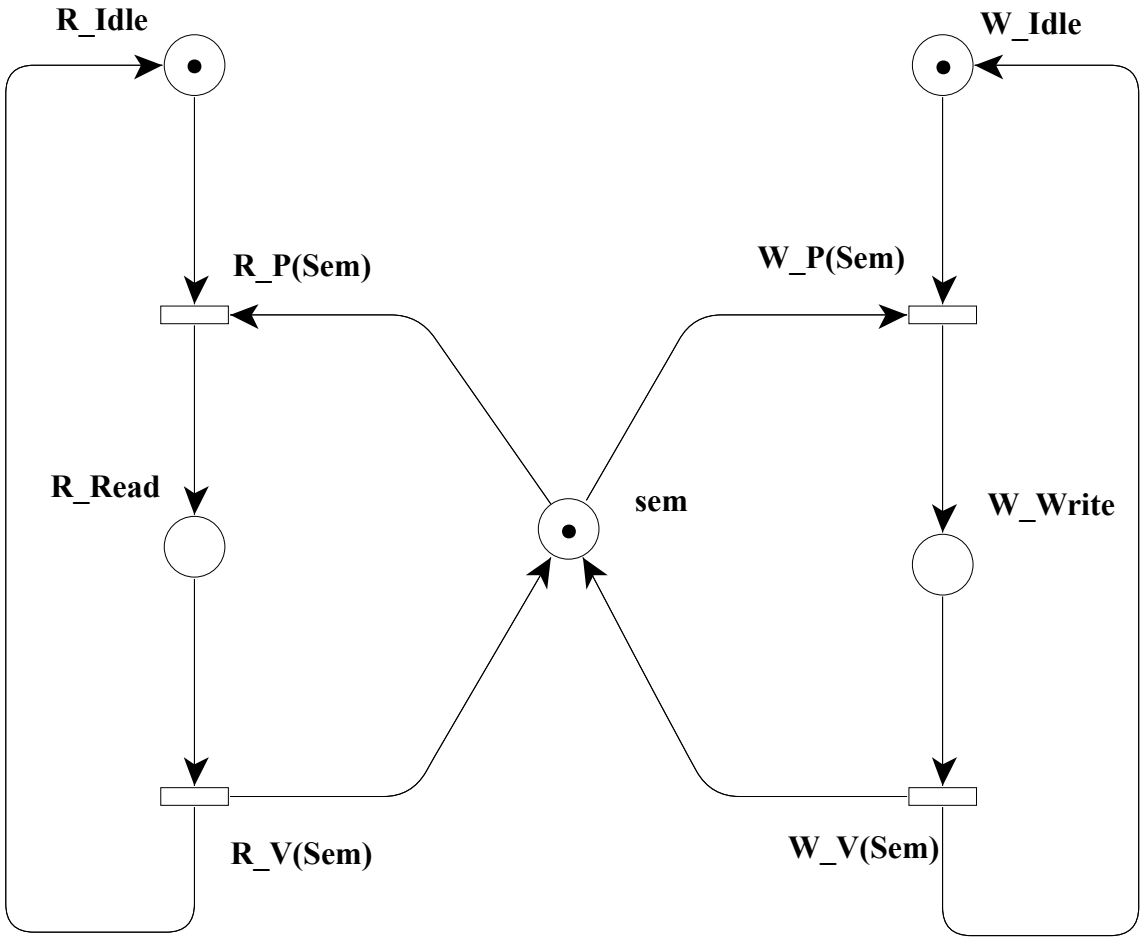
$M_0\ [t_2 > M_1\ [t_4 > M_3\ [t_3 > M_4$

$$M_3 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 6 \\ 3 \end{bmatrix} \qquad M_4 = \begin{bmatrix} 1 \\ 1 \\ 0 \\ 3 \\ 2 \end{bmatrix}$$

The Reader/Writer: example

```
Proc R{
      P(sem)
      Read in file
      V(sem)
}
```

```
Proc W{
      P(sem)
      Wrtite to file
      V(sem)
}
```



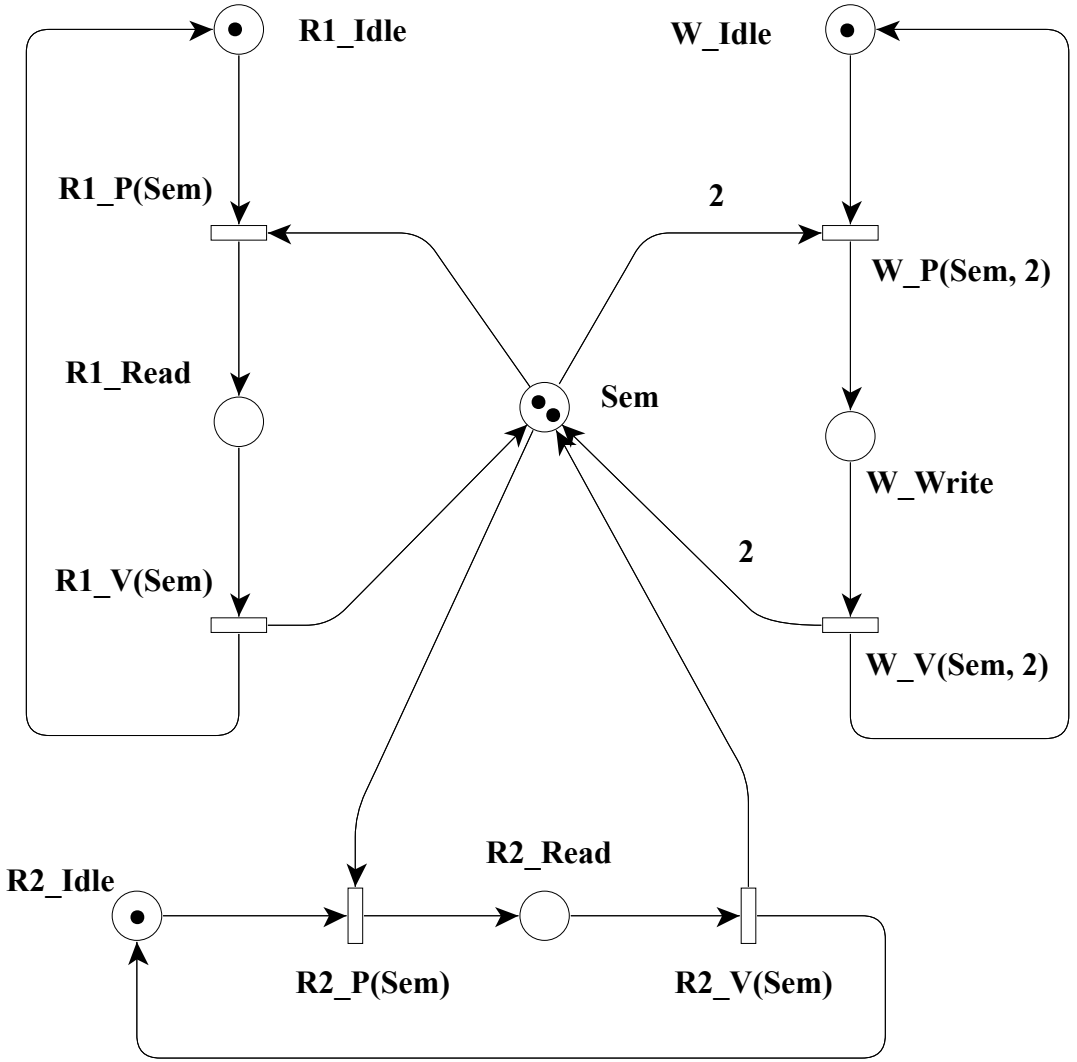
2 Readers / 1 Writer: example (1/2)

```
Proc R1{
      P(sem)
      Read in file
      V(sem)
}
```

```
Proc R2{
      P(sem)
      Read in file
      V(sem)
}
```

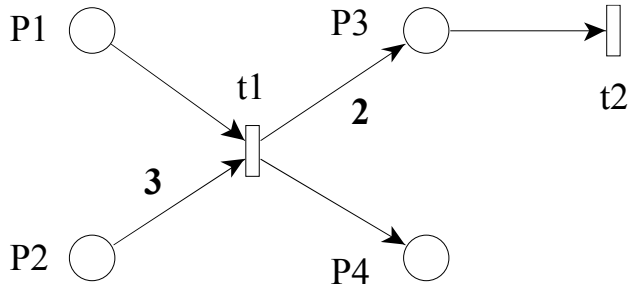
```
Proc W{
      P(sem*2)
      Wrtite to file
      V(sem*2)
}
```

2 Readers / 1 Writer: example (2/2)



PN Matrix Representation

- Graphical Representation



- Matrix Representation

$$\text{Pre} = \begin{array}{c} \\ \text{P1} \\ \text{P2} \\ \text{P3} \\ \text{P4} \end{array} \begin{array}{c} \begin{array}{cc} \text{t1} & \text{t2} \end{array} \\ \begin{bmatrix} 1 & 0 \\ 3 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} \end{array} \qquad \text{Post} = \begin{array}{c} \\ \text{P1} \\ \text{P2} \\ \text{P3} \\ \text{P4} \end{array} \begin{array}{c} \begin{array}{cc} \text{t1} & \text{t2} \end{array} \\ \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 2 & 0 \\ 1 & 0 \end{bmatrix} \end{array}$$

Incidence Matrix

◆ **Definition :**

Let R be a PN.

We define C, the Incidence Matrix of R by:

$$\mathbf{C = Post - Pre}$$

◆ **Firing**:

Let M[t>M'. We have:

$$\mathbf{M'(p)} = M(p) + Post(p, t) - Pre(p, t) = \mathbf{M(p) + C(p, t)}$$

Characteristic vector and equation

◆ **Definition :** *Parikh vector (Characteristic vector)*

Let s be a firing sequence. The *Parikh vector* $\underline{\mathbf{s}}$ of s is an integer vector, indexed by transitions. The t entry represents the number of occurrences of t in s.

$$s = t_1\ t_2\ t_1\ t_4 \quad \Rightarrow \quad \underline{\mathbf{s}} = \begin{bmatrix} 2 \\ 1 \\ 0 \\ 1 \end{bmatrix}$$

◆ **Definition :** *Characteristic equation*

Let **M [s> M'**, then M' an be deduced from M by applying the characteristic equation:

$$\mathbf{M' = M + C.\underline{s}}$$

Reachability Graph

◆ **Definition** : The Reachability Graph (RG) of marked PN $<R, M_0>$, noted $RG(R, M_0)$, is a transitions system $< Q, \Delta, \lambda, q_0 >$ such that:

- **Q** is the set of reachable markings in from M_0

  $Q = \{ M \mid M \in IN^P \wedge \exists \sigma \in T^* , M_0 [\sigma> M \}$

- **Δ** is the set of arcs connecting two markings reachable from M_0

  $\Delta = \{(q_1, q_2) \in Q \times Q \mid t \in T, q_1[t> q_2\}$

- **λ** is a label function, that associates to each arcs in Δ, the name of the transition that have been fired.

  $\lambda : \Delta \rightarrow T$

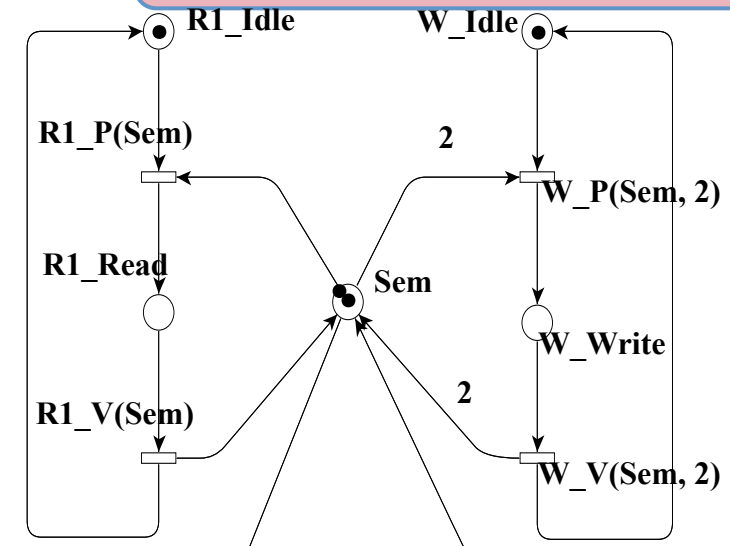
- $\mathbf{q_0} = M_0$

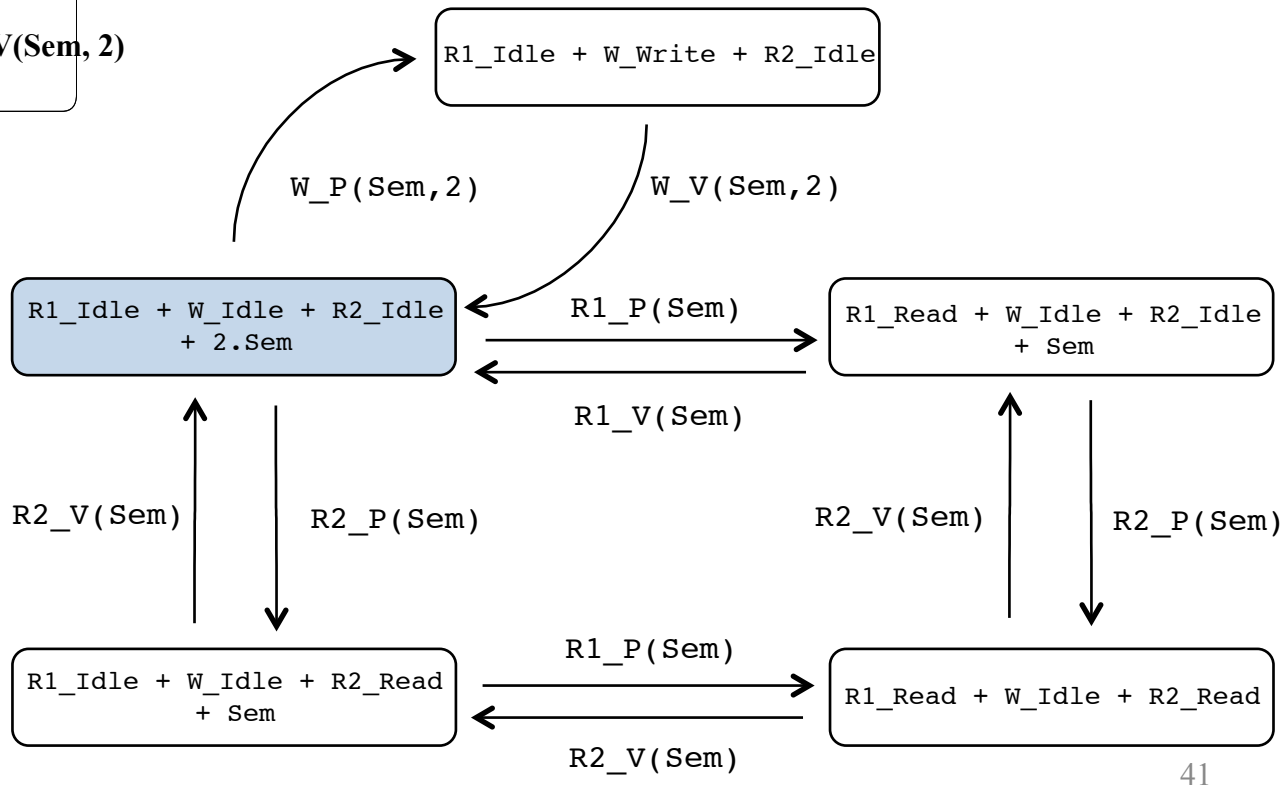
Reachability Graph Construction Algorithm

```
RG.Q = {Mo}; RG.Δ = ∅; RG.q0 = Mo ;
States = {Mo};

While (States <> ∅) {
   s = pick a state in States  ;
   States  = States \ {s};
   for each  t ∈ T {
      if (s[t>) {
         s [t> ns ;
         if (ns ∉ RG.Q) {
            RG.Q = RG.Q ∪ {ns} ;
            States = States ∪ {ns} ;
         }
         RG.Δ = RG.Δ ∪ {(s, ns)} ;
         RG.λ(s, ns) = t ;
      }
   }
}
Return RG;
```

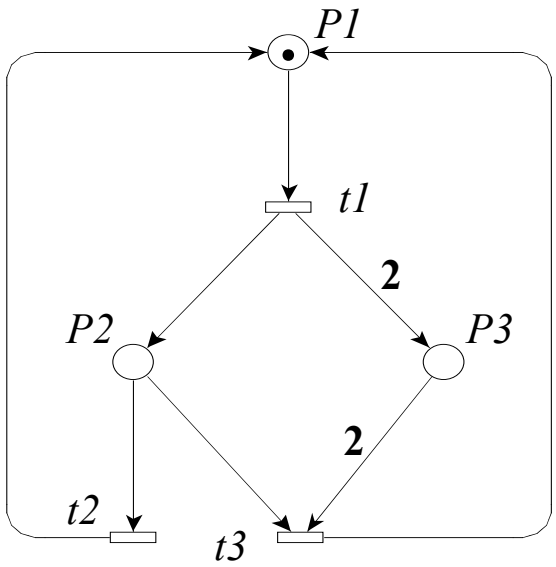
Reachability Graph – Example





Reachability Graph – Remarks

- The RG depends on both **R and M_0**.
- A finite RG can contain *infinite sequences*.
  - Existence of cycles.
- The RG can be infinite!



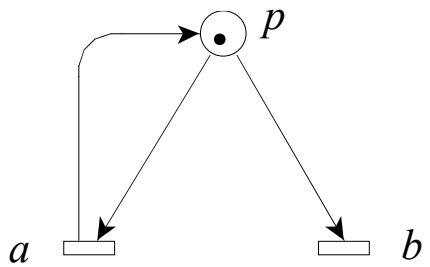
Properties : infinite sequence

Let <R, M_0> be a marked PN. s =$\mathbf{t_1.t_2...t_n...}$, where $\mathbf{t_i} \in$T is an infinite sequence, *iff,* for each finite prefix s' of s, s' is a firing sequence of <R, M_0>, i.e.,

if s = $t_1.t_2. \ldots .t_n. \ldots$, then for all i,

if $s_i = t_1.....t_i$, then $M_0[s_i>$

<R, M_0> :

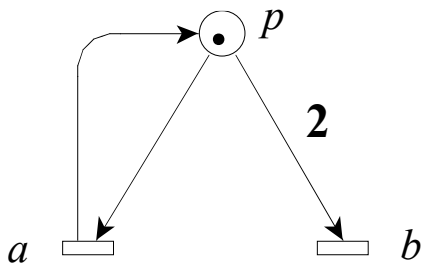


abababab…..

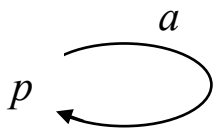
Properties : the pseudo-aliveness

A PN $<R, M_0>$ is pseudo-alive if
$\forall\ M \in RG(R, M_0), \exists\ t \in T : M[t>$

$<R, M_0>$:



$RG(R, M_0)$:

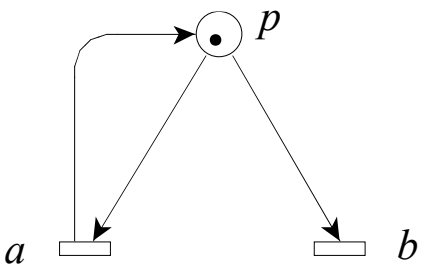


Properties : the quasi-aliveness

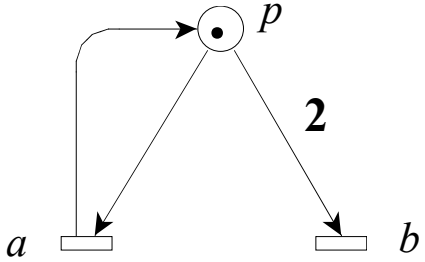
A PN <R, M_0> is quasi-alive if

$\forall t \in T, \exists M \in RG(R, M_0) : M[t>$

<R, M_0>



<R', M_0>

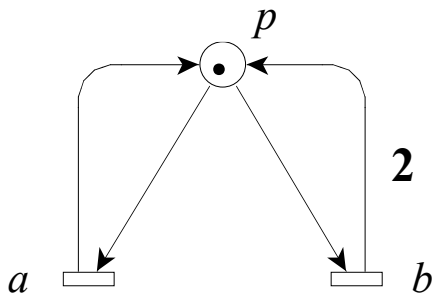


Properties : the aliveness

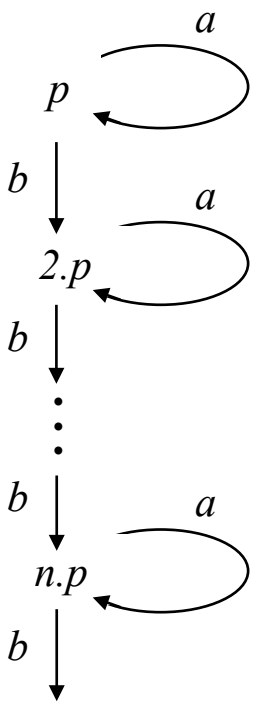
A PN <R, M_0> is alive if :

$\forall$ M $\in$ RG(R, M_0), <R, M> is quasi-alive

<R, M_0> :



RG(R, M_0) :

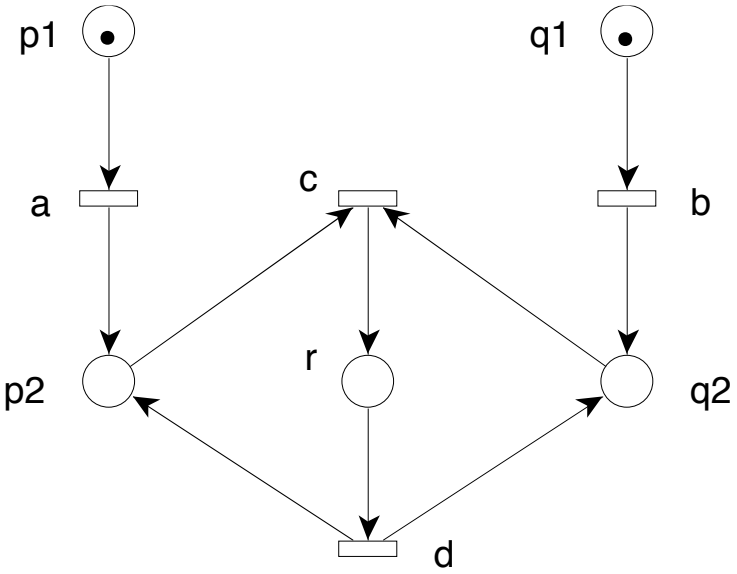


Properties : the home state

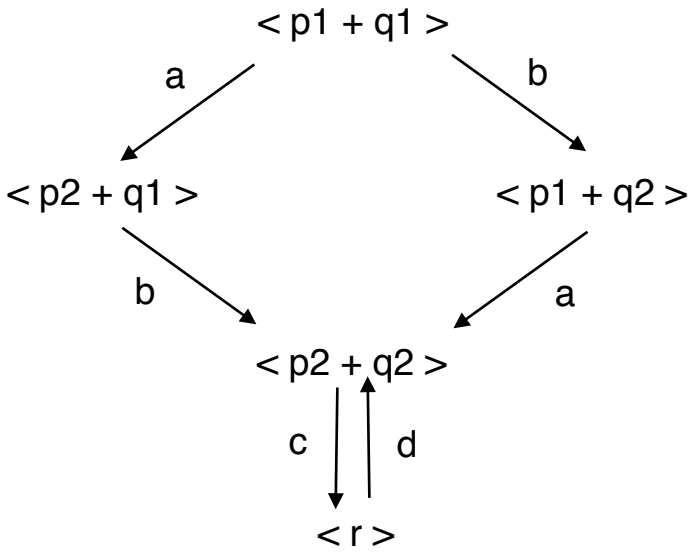
M_a is a Home State of $<R, M_0>$ if:

$\forall\ M \in RG(R, M_0), \exists$ a sequence **s** : $M[s>M_a$.

< R, M_0 >



RG(R, M_0)



Properties : the boundedness

A PN $<R, M_0>$ is not bounded if :

$\forall n \in N, \exists M \in RG(R, M_0), \exists p \in P$ s.t. : $M(p) > n$

The RG is infinite

Relations between properties

- If $<R, M_0>$ is pseudo-alive or not bounded, then $<R, M_0>$ admits an infinite sequence.

- If $<R, M_0>$ is alive, then it is quasi-alive and pseudo-alive.

- If $<R, M_0>$ is quasi-live and admits M_0 as a home state, then$<R, M_0>$ is alive.

Peterson Algorithm Model (PAM)

- Peterson algorithm : mutual exclusion of two processes.
  - The two processes are symmetrical.
  - A shared memory contains variables: `turn`, dem_p and dem_q.

- Code of process p:

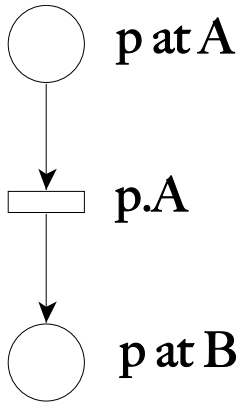
```
A :    dem_p = true
B :    turn = q
C :    wait (turn == p || dem_q  == false)
D :    < Section critique >
E :    dem_p = false; goto A
```

  - Initially :
    - $dem_p = dem_q = false$

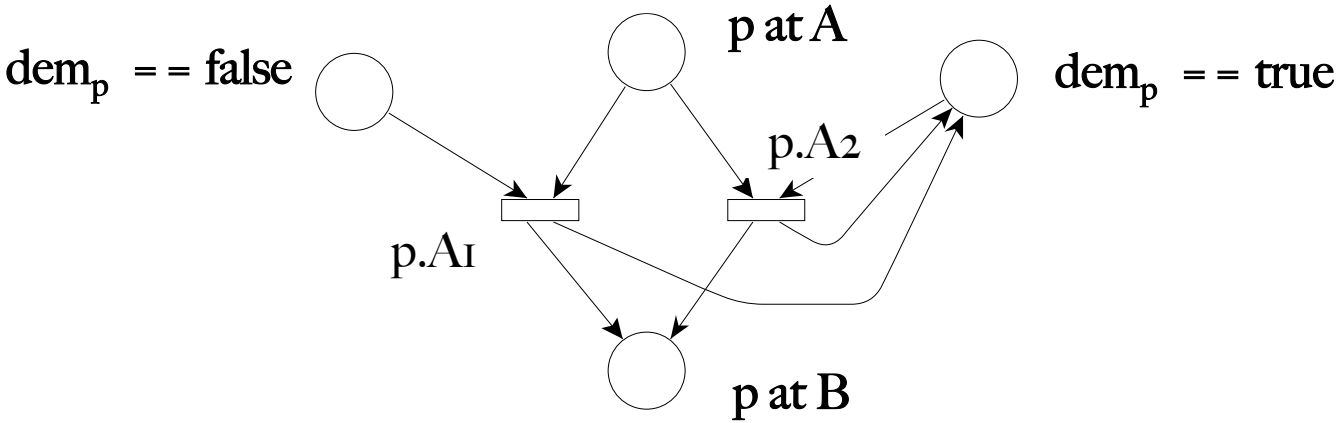
PAM : execution of the first instruction

A : dem_p = true

Move p from A to B:

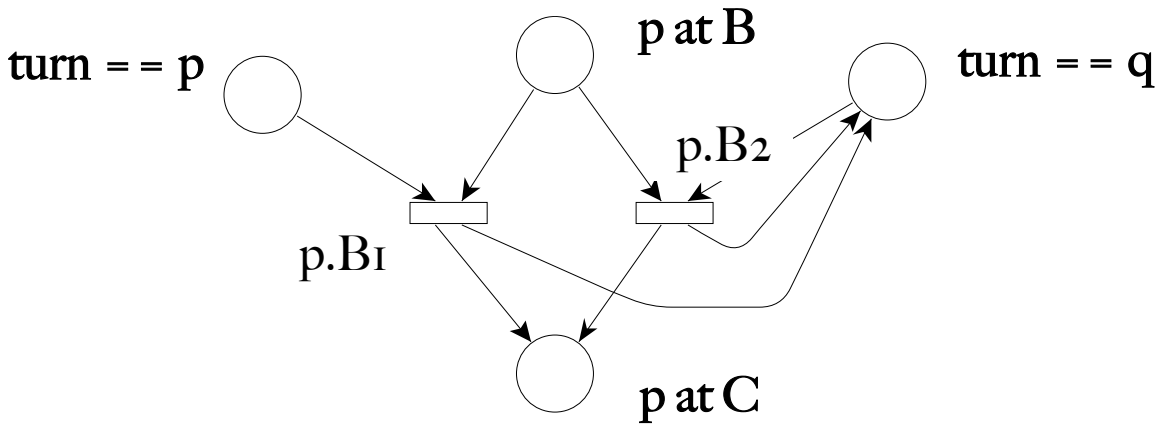


Potentially, modifies the value of dem_p:



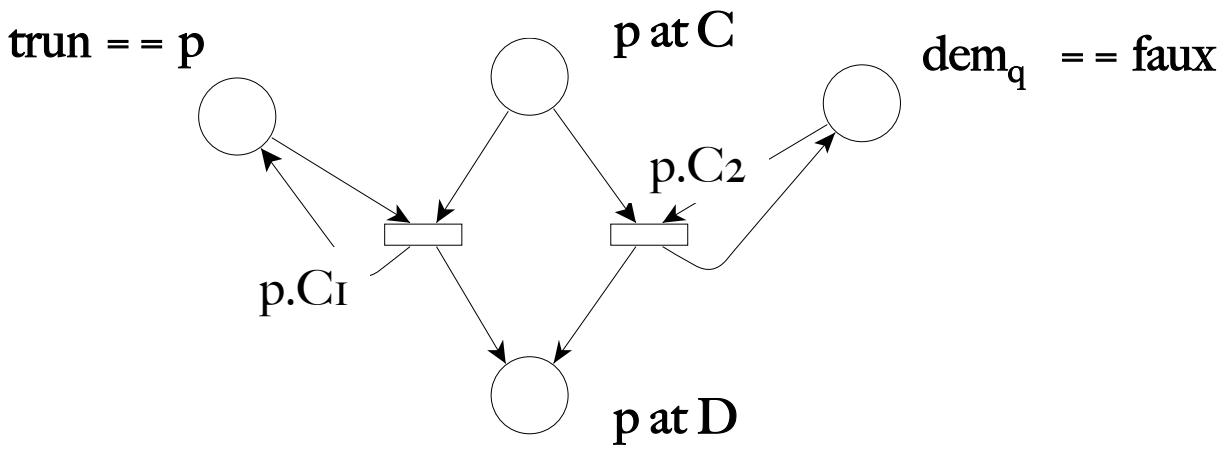
PAM : execution of the second instruction

`B : turn = q`



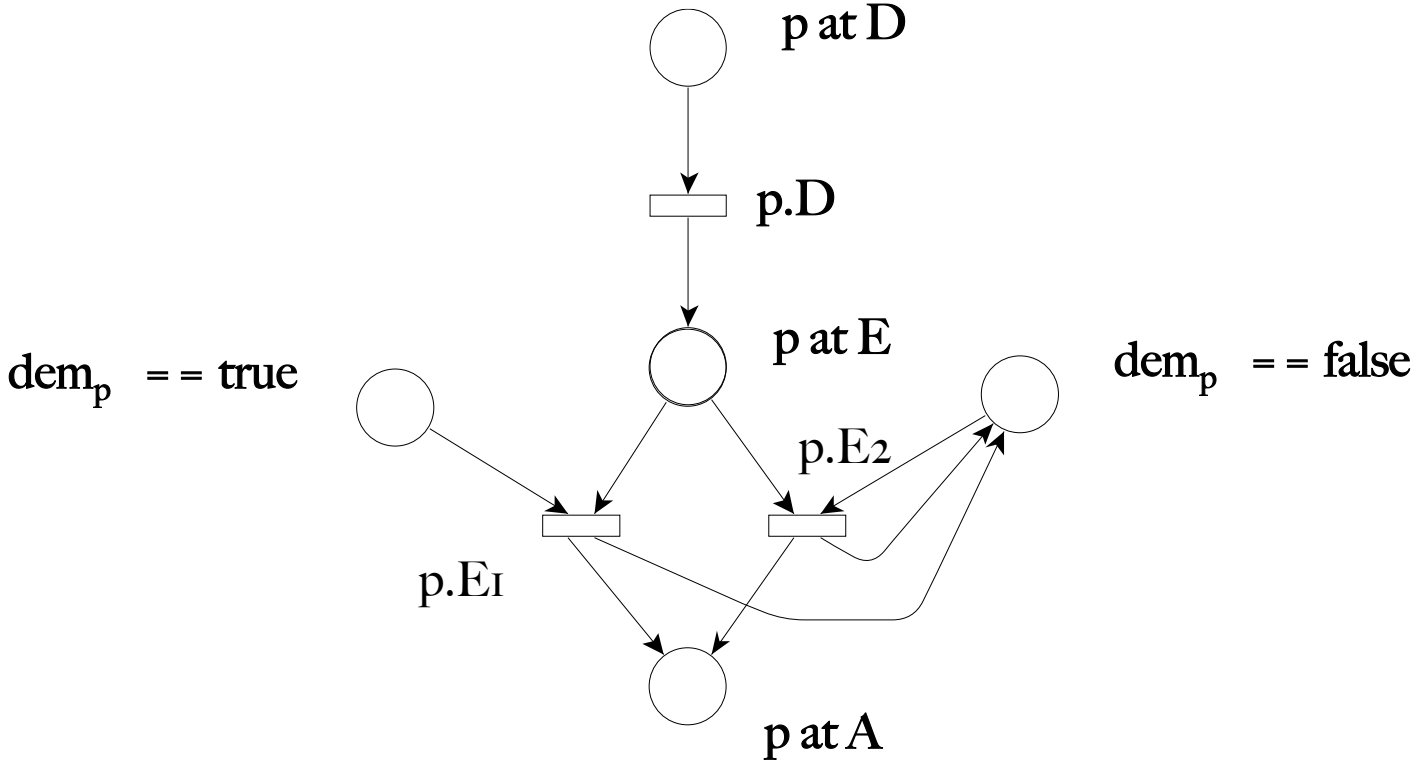
PAM : the waiting model

C : wait (turn == p || dem_q == faux)

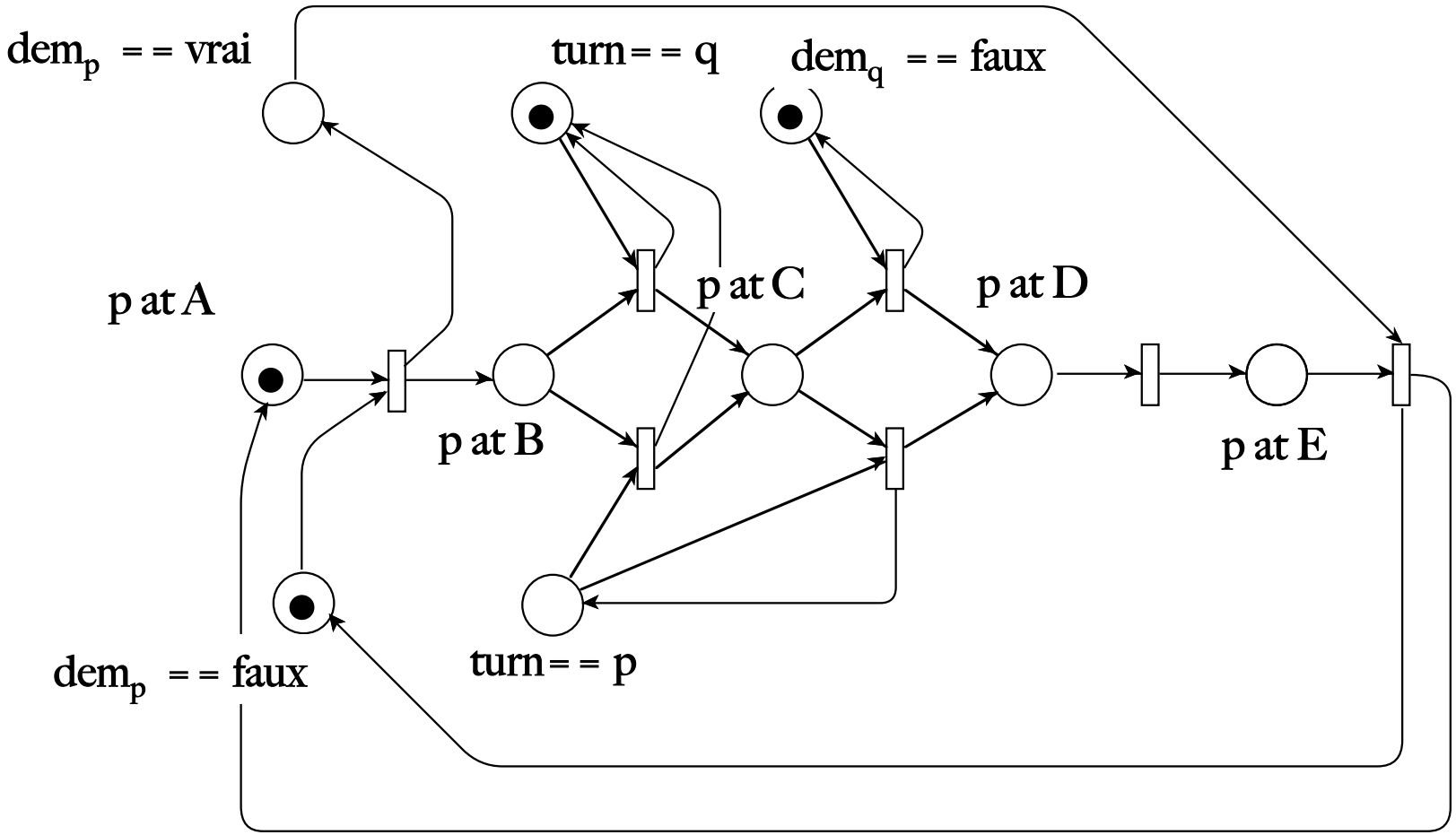


PAM : the critical section exit

```
D :   < Section critique >
E :   dem_p = faux; goto A
```



PAM : putting all together



First Practice.

Specification of the system:

the temporal logic.

What kinds of properties ?

- Safety : No unwanted situation is reached
- Liveness : Wanted situation are eventually reached
- Fairness : Particular liveness properties

Any property is the conjunction of safety and liveness (*Lamport 77*)

Why a temporal logic ?

- Propositional logic is not sufficient!
  - It addresses the properties which are local to some state.

- A formalism to express properties,
  - on a sequence of states (program states, system states) or,
  - on a sequence of actions (instructions)

A mutual exclusion algorithm

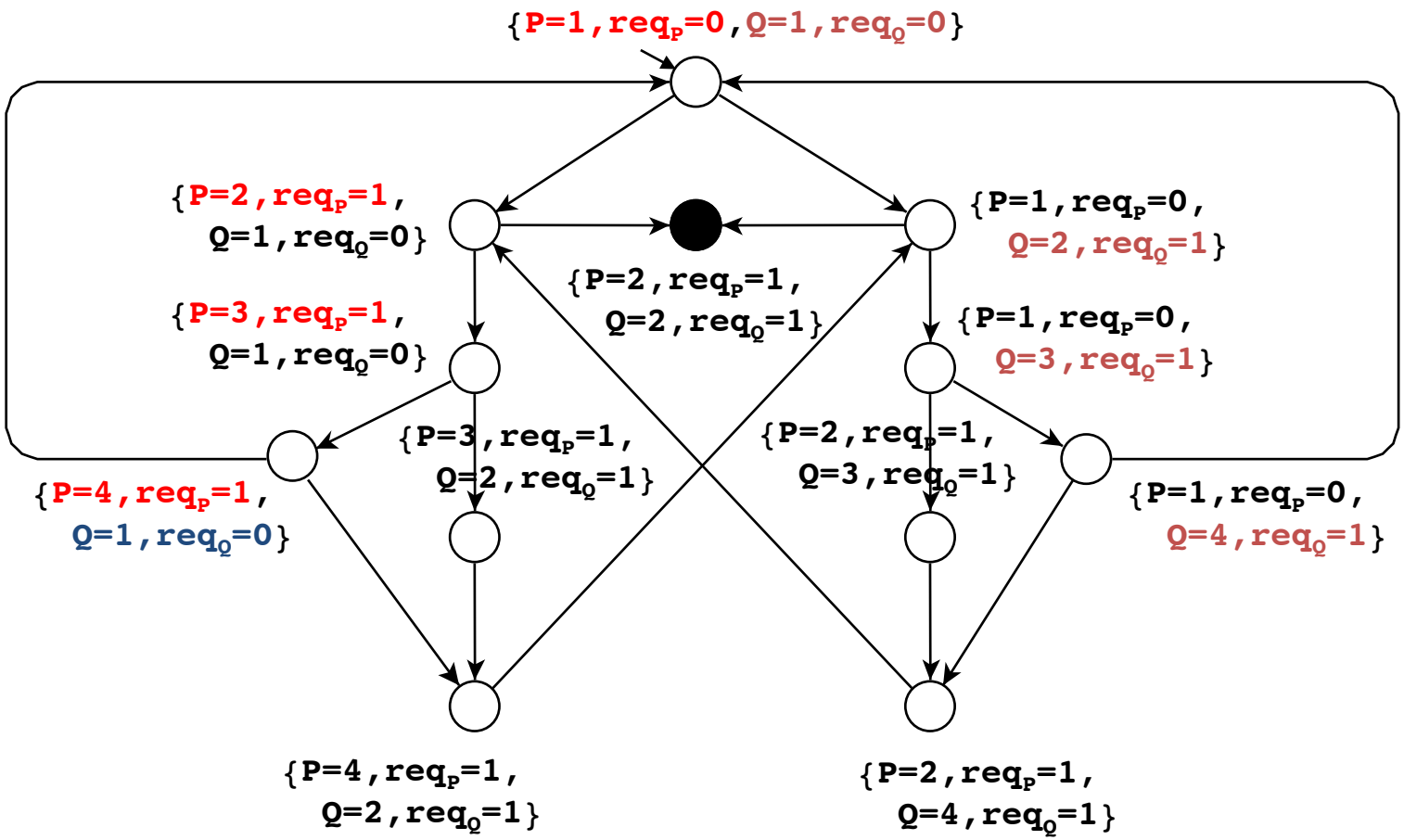
Initial state : P=1 ; Q=1 ; req_P = 0 ; req_Q = 0 (2 global variables)

P:
1- req_P := 1
2- wait (req_Q = 0)
3- *critical sect.*
4- req_P := 0; goto 1

Q:
1- req_Q := 1
2- wait (req_P = 0)
3- *critical sect.*
4- req_Q := 0; goto 1

- Prop1: In any case, there is at most one process in the critical section
- Prop2: Any process requiring the critical section will eventually reaches it
- Prop3: The order of entrances in the critical section must respect the order of the requests.

Kripke Structure of the previous algorithm



Property Prop1

For each reachable state,
(P = 3 and Q = 3) does not hold

- The property is satisfied!
- It can be checked by using the set of reachable states.

Property Prop2

For each path and for each state visited by this path
if (P = 2) holds,
then for each path starting from this state,
there is a state satisfying (P = 3).

- The property is not satisfied!

  from {P = 2, req_P = 1, Q = 2, req_Q = 1}
  there is no more reachable state (deadlock)
  in particular those satisfying (P=3).

- *Its verification requires the reachability graph (states are not enough).*

Property Prop3

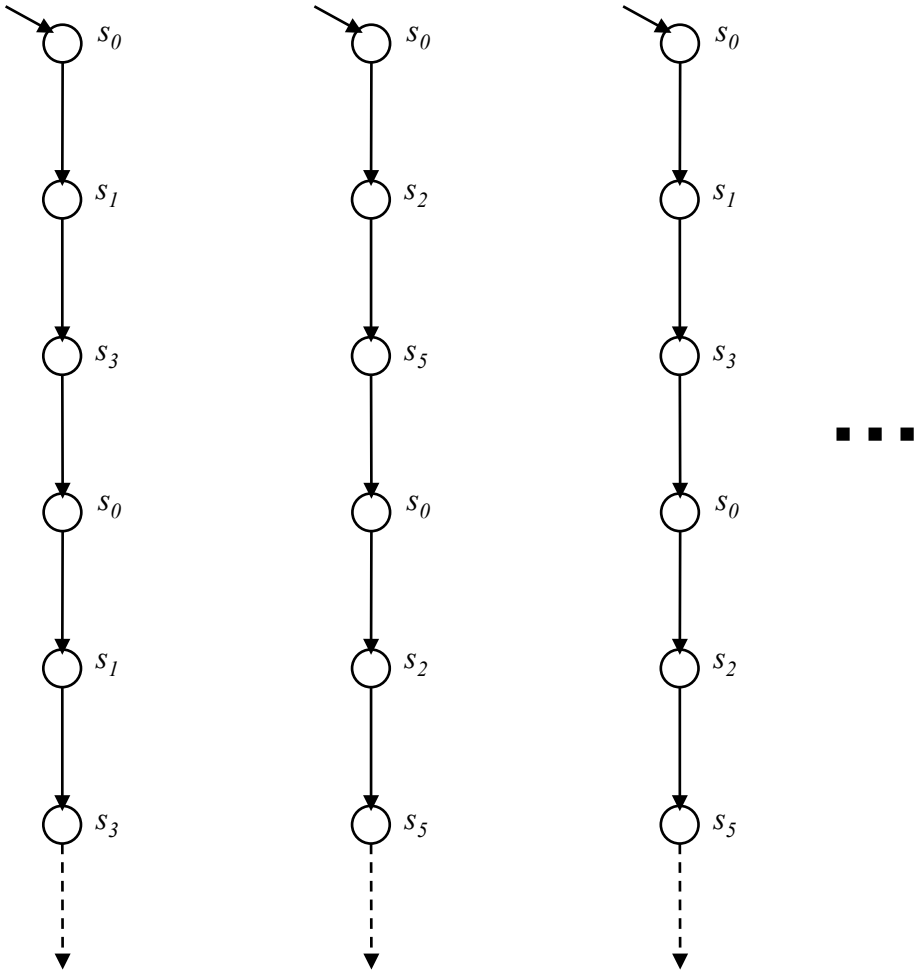
For each path and for each state of this path
if [(P=2) and (Q=1)] holds,
then for each path starting from this state,
(Q=3) does not hold until (P=3) holds.

- The property is not satisfied !

  from the state {P=2, req_P=1, Q=1, req_Q=0}
  there exists a path in which (P=3) is never satisfied

Linear time properties

Recall : Linear time semantics



The linear time semantics of a Kripke structure corresponds to the (possibly infinite) set of all infinite paths

Logics to express linear time properties

- LTL = Linear-time Temporal Logic

- *Syntax:*

  Let AP be a set of atomic propositions.

  – $a \in AP$ is an LTL formula

  – If ϕ_1 and ϕ_2 are LTL formulae then so are

$$\neg\ \phi_1 \qquad \phi_1 \wedge \phi_2 \qquad X\ \phi_2 \qquad \phi_2\ U\ \phi_2$$

where X stands for « next » and U for « until »

Semantics of LTL

- To each LTL formula ϕ, we associate a language $\mathcal{L}(\phi)$ of ω-words over 2^{AP} (i.e. we have $\mathcal{L}(\phi) \subseteq (2^{AP})^{\omega}$).
- Let $\sigma \in (2^{AP})^{\omega}$.

$$\sigma \in \mathcal{L}(a) \quad \Leftrightarrow \quad a \in \sigma(0)$$

$$\sigma \in \mathcal{L}(\neg\ \phi) \quad \Leftrightarrow \quad \neg\, \sigma \in \mathcal{L}(\phi)$$

$$\sigma \in \mathcal{L}(\phi_1 \wedge \phi_2) \quad \Leftrightarrow \quad \sigma \in \mathcal{L}(\phi_1) \cap \mathcal{L}(\phi_2)$$

$$\sigma \in \mathcal{L}(X\ \phi) \quad \Leftrightarrow \quad \sigma^1 \in \mathcal{L}(\phi)$$

$$\sigma \in \mathcal{L}(\phi_1\ U\ \phi_2) \quad \Leftrightarrow \quad \exists\, i : \sigma^i \in \mathcal{L}(\phi_2) \wedge \forall\, k < i,\ \sigma^k \in \mathcal{L}(\phi_1)$$

Syntactic complements

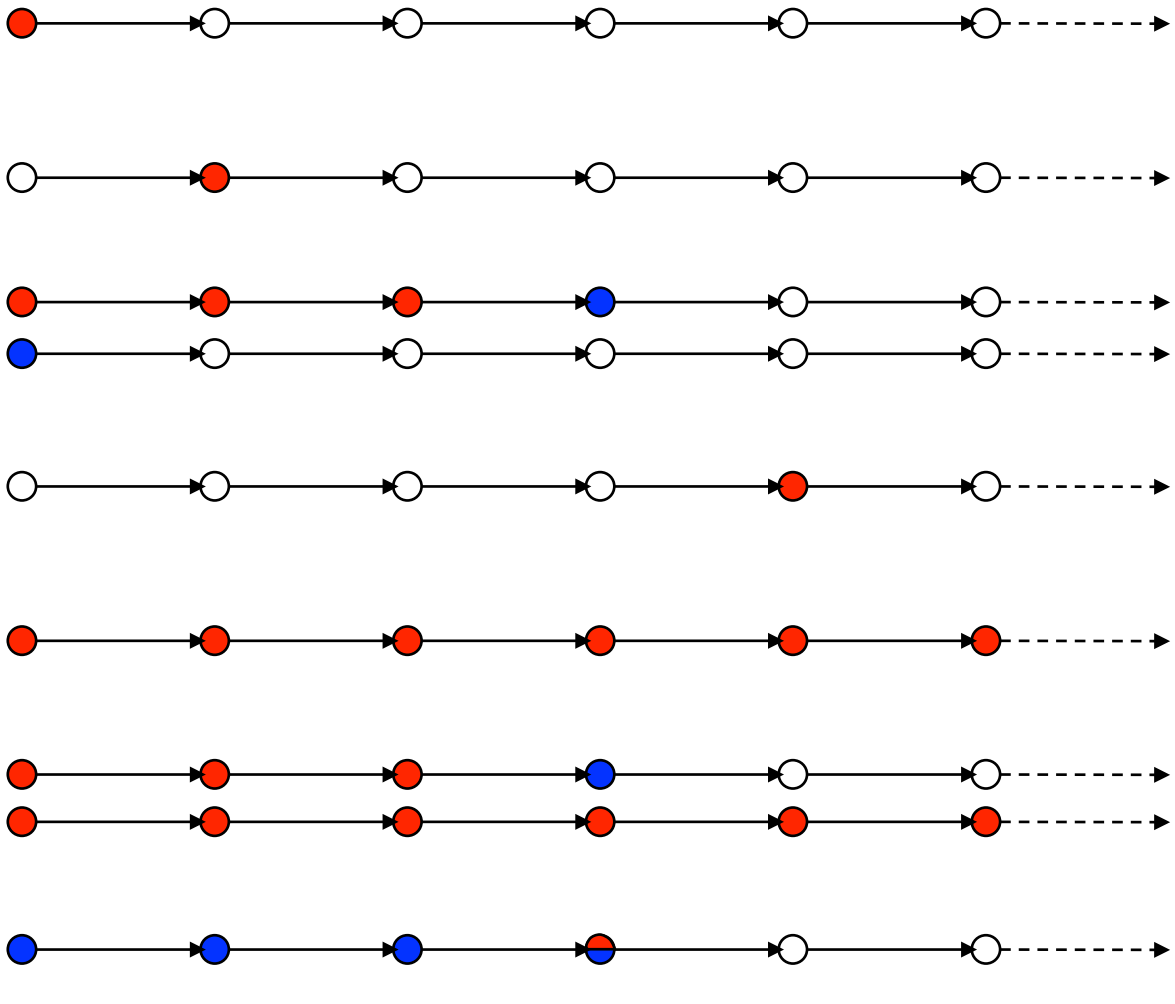
- The previous slides defined only a minimal version of the syntax.
- In practice, we will make use of the following abbreviations:

- $\phi 1 \vee \phi 2 \equiv \neg(\neg \phi 1 \wedge \neg \phi 2)$
- $\phi 1 \Rightarrow \phi 2 \equiv \neg \phi 1 \vee \phi 2$
- $true \equiv \neg a \vee a$
- $false \equiv \neg true$
- $F \phi \equiv true\ U\ \phi$
- $G \phi \equiv \neg F \neg \phi$
- $\phi_1\ W\ \phi_2 \equiv (\phi_1\ U\ \phi_2) \vee G\ \phi_1$
- $\phi_1\ R\ \phi_2 \equiv \neg (\neg \phi_1\ U \neg \phi_2)$

where *F* stands for « finally », *G* for « globally », *W* for « weak until » and *R* for « release »

Illustration of the semantics

- a
- $X\ \phi$
- $\phi_1\ U\ \phi_2$
- $F\ \phi$
- $G\ \phi$
- $\phi_1\ W\ \phi_2$
- $\phi_1\ R\ \phi_2$



Examples of LTL formulae (1/3)

- *Reachability*
  $G\ \neg(a_1 \wedge a_2)$: It always holds that a_1 and a_2 do not appear together. Assuming an appropriate valuation, this expresses the mutex property: Two processes never enter their critical sections at the same time.
- *Safety*
  $(\neg x)\ W\ y$: x does not occur before the first occurrence of y. Note: y may not occur at all, in which case x also does not occur.
- *Liveness*
  $(\neg x)\ U\ y$: x does not occur before the first occurrence of y, and y does eventually occur.

Examples of LTL formulae (2/3)

- $GF\ p$
  p appears infinitely often.

- $G(r_1 \Rightarrow F\ a_1)$
  When interpreted on a mutex algorithm:
  Whenever process 1 requests to enter its critical section, it will eventually succeed.

Examples of LTL formulae (3/3)

- Prop1

  $G \neg (P = 3 \wedge Q = 3)$

- Prop2

  $G ((P = 2) \Rightarrow F (P = 3))$

- Prop3

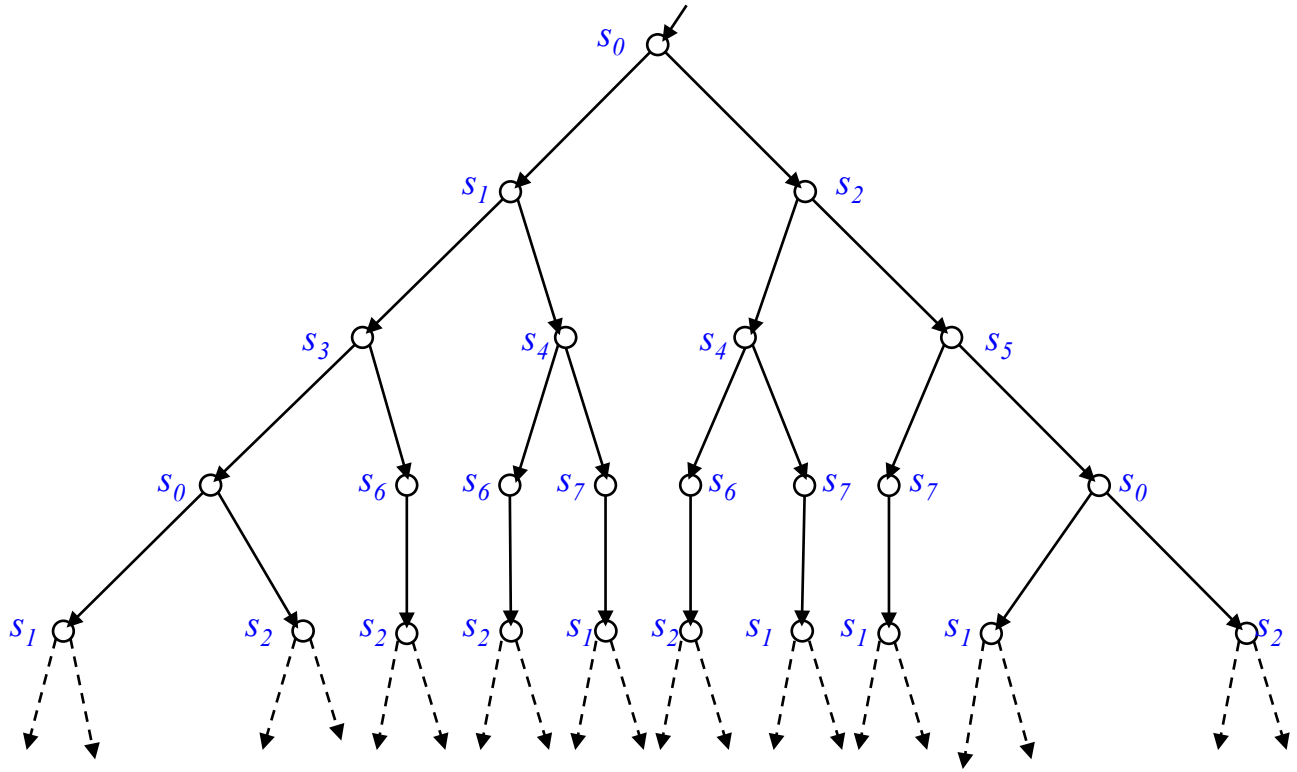
  $G ((P = 2 \wedge Q = 1) \Rightarrow (\neg Q = 3) U (P = 3))$

Interpretation of LTL on Kripke structures

- Let $K = \langle S, \ell, \rightarrow, s_0 \rangle$ be a Kripke structure
- A run ρ of K satisfies an LTL formula ϕ (written $\rho \models \phi$) iff $\ell(\rho) \in \mathcal{L}(\phi)$.
- By extension, we say that K satisfies ϕ (written $K \models \phi$) iff *for each run* ρ starting at s_0 (i.e. those with $\rho(0) = s_0$) we have $\rho \models \phi$.

Branching time properties

Recall : Branching time semantics



The branching time semantics of a Kripke structure corresponds to the underlying infinite tree

Logics to express the branching time properties

- CTL = Computational Tree Logic
- CTL versus LTL
  - LTL describes properties of individual executions.
  - Its semantics is defined as a set of executions.
  - CTL describes properties of a *computation tree*: formulas can reason about many executions at once. (CTL belongs to the family of *branching-time logics*.)
  - Its semantics is defined in terms of states.

Syntax of CTL

- CTL Combines temporal operators with quantification over runs.

- Syntax:

  Let AP be a set of atomic propositions.

  – $a \in AP$ is a CTL formula

  – If ϕ_1 and ϕ_2 are CTL formulae then so are

  $\neg\ \phi_1 \quad \phi_1 \wedge \phi_2 \quad EX\ \phi_1 \quad EG\ \phi_1 \quad \phi_1\ EU\ \phi_2$

  where *X* stands for « next », *G* for « globally », and *U* for « until »

Semantics of CTL

- Let $K = \langle S, \ell, \rightarrow, s_0 \rangle$ be a Kripke structure.
- To each CTL formula ϕ, we associate a set $\mathcal{S}_K(\phi)$ of states w.r.t. K (i.e. we have $\mathcal{S}_K(\phi) \subseteq S$)

$$s \in \mathcal{S}_K(a) \quad \Leftrightarrow \quad a \in \ell(s)$$

$$s \in \mathcal{S}_K(\neg\ \phi) \quad \Leftrightarrow \quad s \notin \mathcal{S}_K(\phi)$$

$$s \in \mathcal{S}_K(\phi_1 \wedge \phi_2) \quad \Leftrightarrow \quad s \in \mathcal{S}_K(\phi_1) \cap \mathcal{S}_K(\phi_2)$$

$$s \in \mathcal{S}_K(EX\ \phi) \quad \Leftrightarrow \quad \exists\, s' : s \rightarrow s' \wedge s' \in \mathcal{S}_K(\phi)$$

$$s \in \mathcal{S}_K(EG\ \phi) \quad \Leftrightarrow \quad \exists \text{ a run } \rho \text{ of } K \text{ s.t. } \rho(0) = s$$
$$\wedge \qquad\qquad\qquad\qquad\qquad\qquad \forall\, i \geq 0,\ \rho(i) \in \mathcal{S}_K(\phi)$$

$$s \in \mathcal{S}_K(\phi_1\ EU\ \phi_2) \Leftrightarrow \exists \text{ a run } \rho \text{ of } K \text{ s.t. } \rho(0) = s \wedge$$
$$\exists\, i \text{ s.t. } \rho(i) \in \mathcal{S}_K(\phi_2) \wedge \forall\, k < i,\ \rho(k) \in \mathcal{S}_K(\phi_1)$$

- K satisfies a CTL formula ϕ iff $s_0 \in S_K(\phi)$

Syntactic complements

- In practice, we will make use of the following abbreviations:

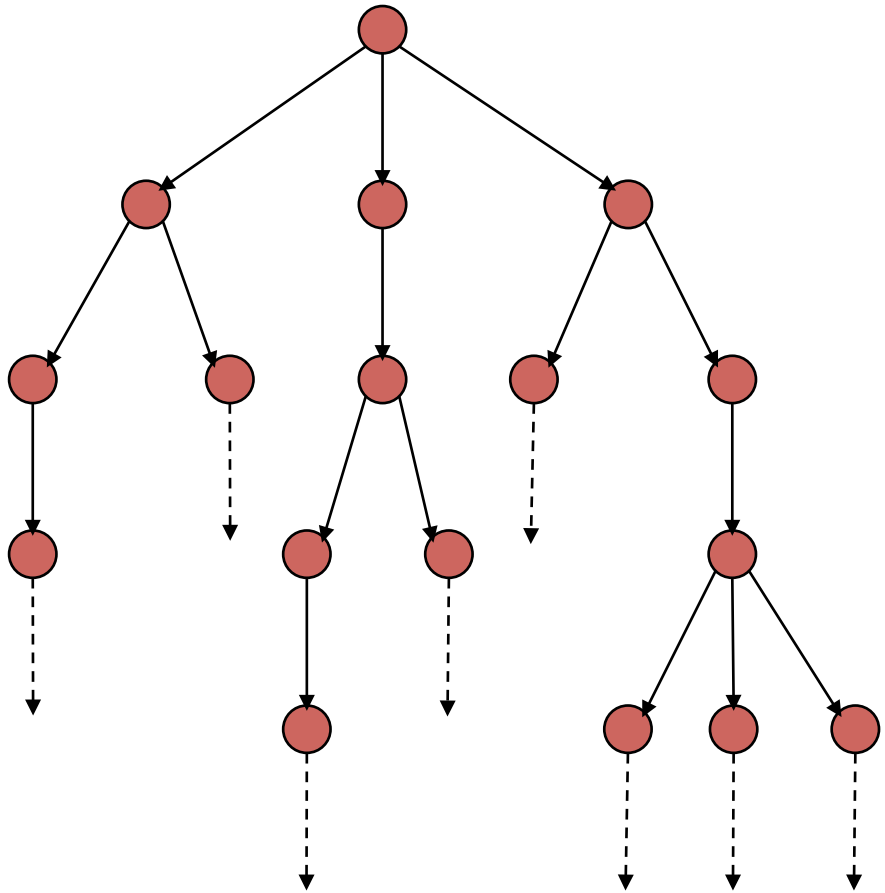
- $\phi 1 \vee \phi 2 \equiv \neg(\neg \phi 1 \wedge \neg \phi 2)$
- $\phi 1 \Rightarrow \phi 2 \equiv \neg \phi 1 \vee \phi 2$
- $true \equiv \neg a \vee a$
- $false \equiv \neg true$

- $EF\ \phi \equiv true\ EU\ \phi$
- $\phi_1\ EW\ \phi_2 \equiv (\phi_1\ EU\ \phi_2) \vee EG\ \phi_1$
- $\phi_1\ R\ \phi_2 \equiv \neg(\neg \phi_1\ U \neg \phi_2)$

- $AX\ \phi \equiv \neg\ EX \neg \phi$
- $AG\ \phi \equiv \neg\ EF \neg \phi$
- $AF\ \phi \equiv \neg\ EG \neg f$
- $\phi_1\ AW\ \phi_2 \equiv \neg(\neg \phi_2\ EU \neg(\phi_1 \vee \phi_2))$
- $\phi_1\ AU\ \phi_2 \equiv AF\phi_2 \wedge (\phi_1\ AW\ \phi_2)$

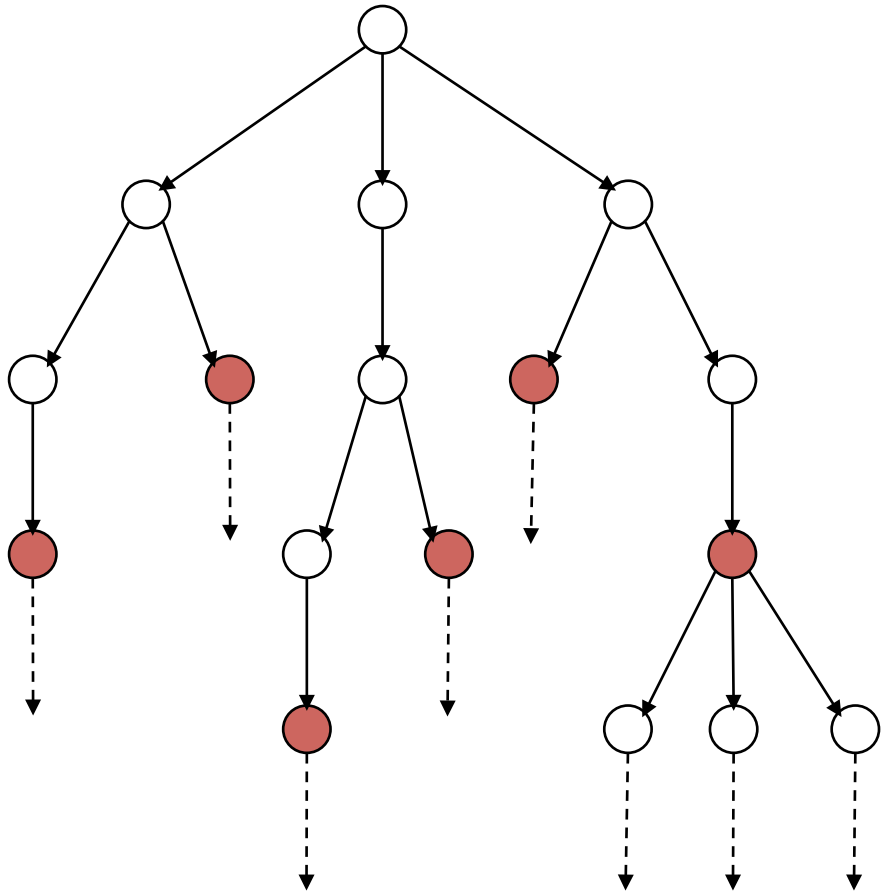
Semantics illustration (1/8)

$AG\ p$



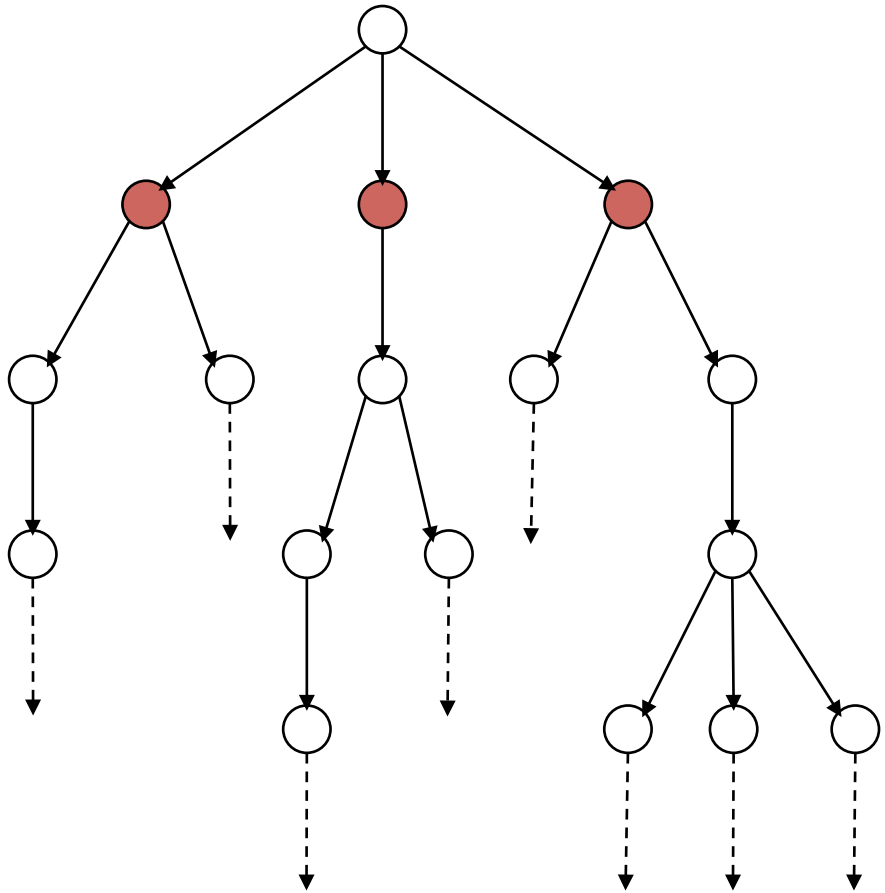
Semantics illustration (2/8)

$AF\ p$



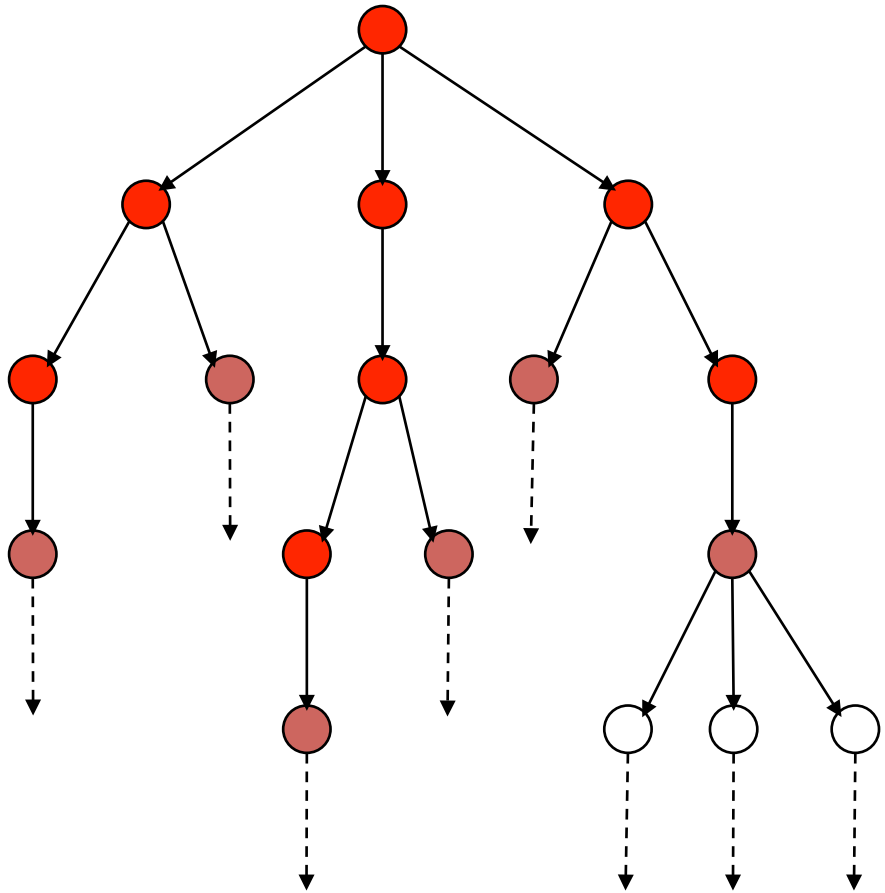
Semantics illustration (3/8)

$AX\ p$



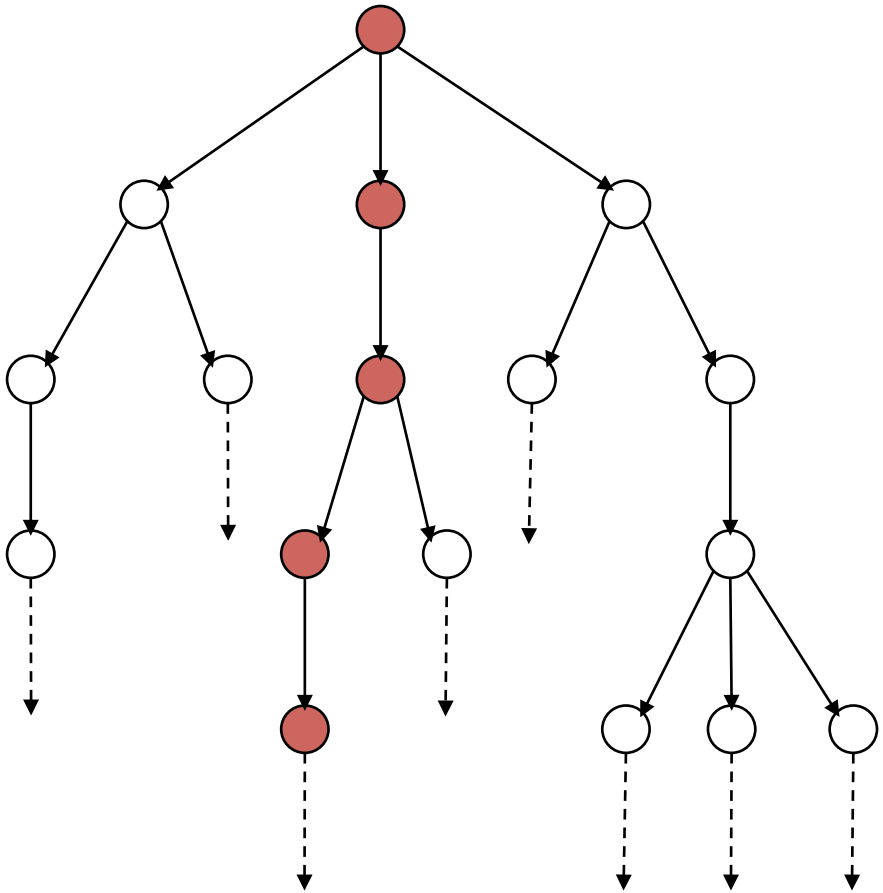
Semantics illustration (4/8)

$q\ AU\ p$



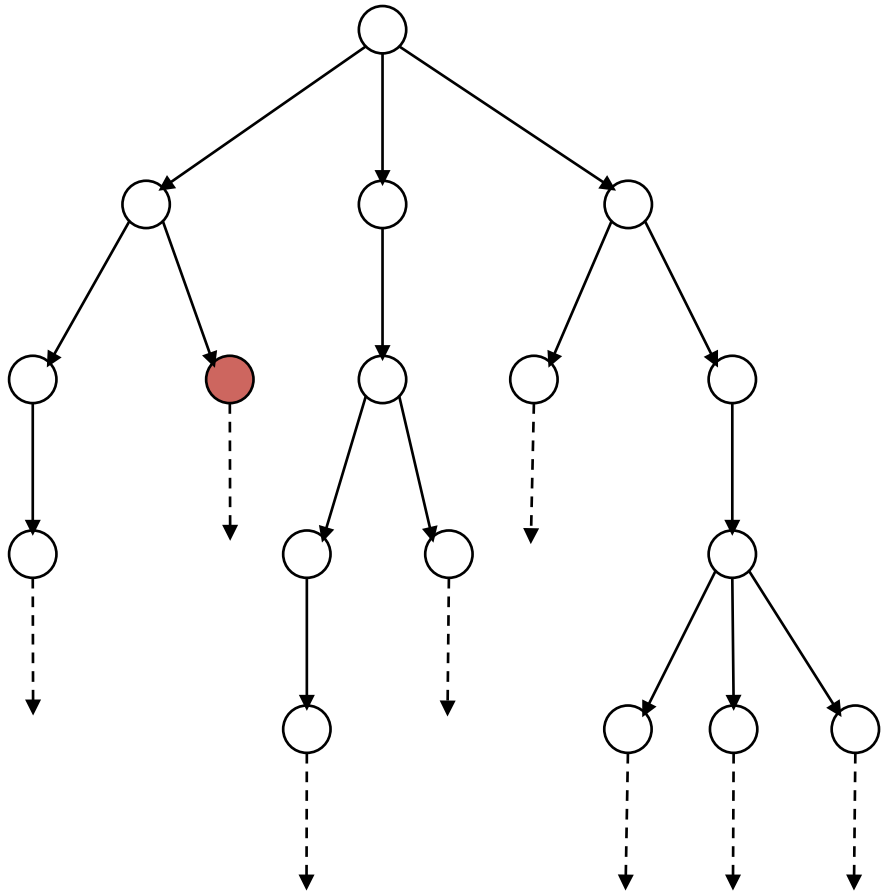
Semantics illustration (5/8)

EG p



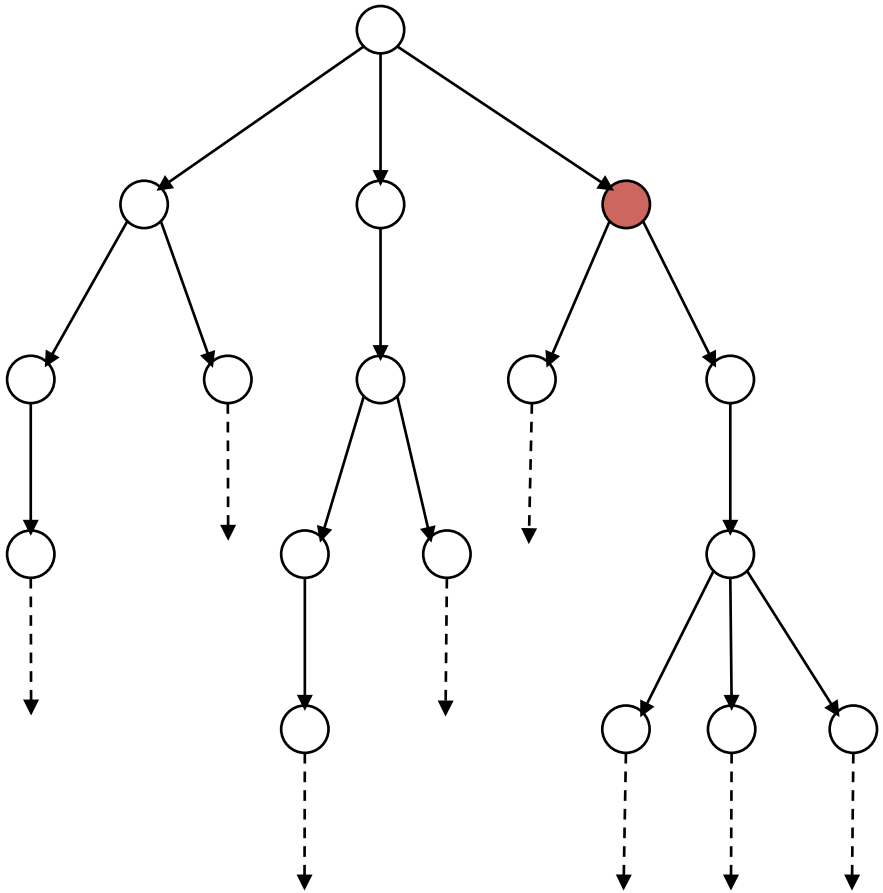
Semantics illustration (6/8)

EF p



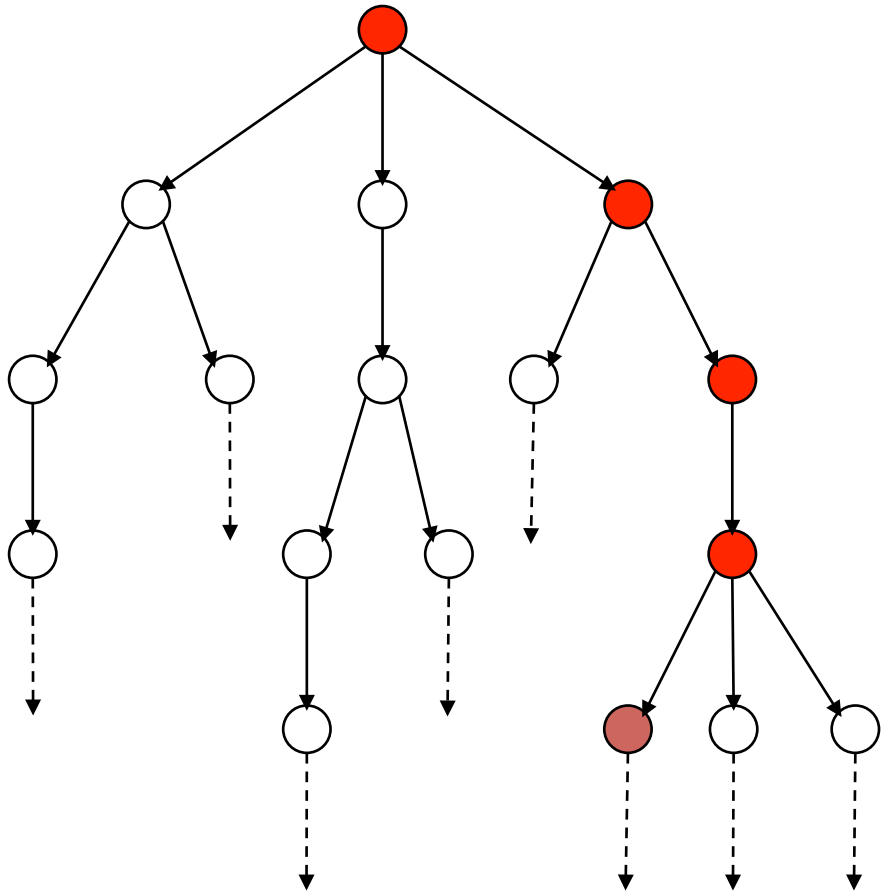
Semantics illustration (7/8)

$EX\ p$



Semantics illustration (8/8)

q EU p



Examples of CTL formulae (1/2)

- *Reachability*
  $AG\ \neg(a_1 \wedge a_2)$: It always holds that a_1 and a_2 do not appear together. Assuming an appropriate valuation, this expresses the mutex property: Two processes never enter their critical sections at the same time.
- *Safety*
  $(\neg x)\ AW\ y$: x does not occur before the first occurrence of y. Note: y may not occur at all, in which case x also does not occur.
- *Liveness*
  $(\neg x)\ AU\ y$: x does not occur before the first occurrence of y, and y does eventually occur.

Examples of CTL formulae (2/2)

- $AG\,AF\,p$
  p appears infinitely often.

- $AG(r_1 \Rightarrow AF\,a_1)$
  When interpreted on a mutex algorithm:
  Whenever process 1 requests to enter its critical section, it will eventually succeed.

Expressiveness of CTL and LTL (1/3)

- CTL and LTL have a large overlap, i.e. properties expressible in both logics.

Examples:

- Invariants (e.g., "*p* never holds.")

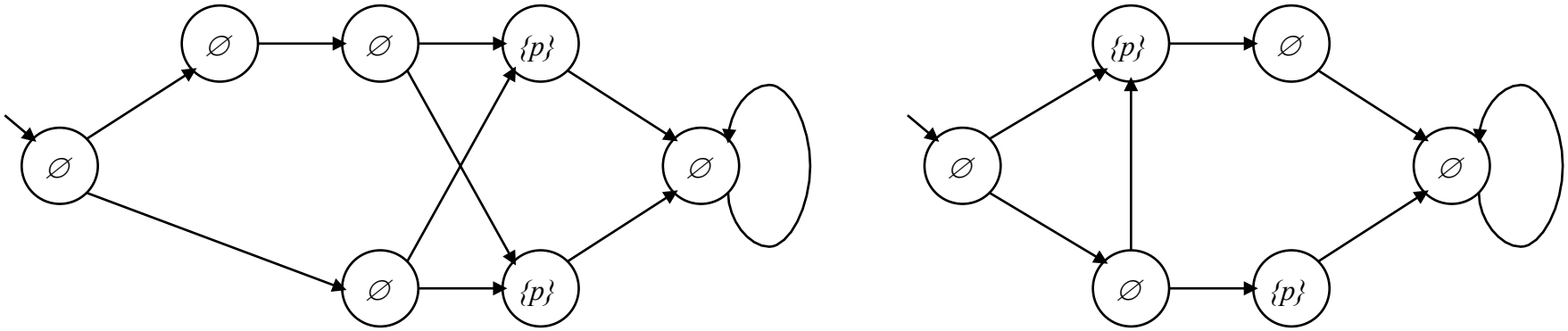
$$AG\ \neg p \quad \text{or} \quad G\ \neg p$$

- Reactivity (e.g. "Whenever *p* happens, eventually *q* will happen.")

$$AG\ (p \Rightarrow AF\ q) \quad \text{or} \quad G(p \Rightarrow Fq)$$

Expressiveness of CTL and LTL (2/3)

- CTL considers the whole computation tree whereas LTL only considers individual runs. Thus CTL allows to reason about the *branching behavior*, considering multiple possible runs at once.
- Examples:
  - The CTL property $AG\ EF\ p$ (“reset property”) is not expressible in LTL.
  - The CTL property $AF\ AX\ p$ distinguishes the following two systems, but the LTL property $FX\ p$ does not:

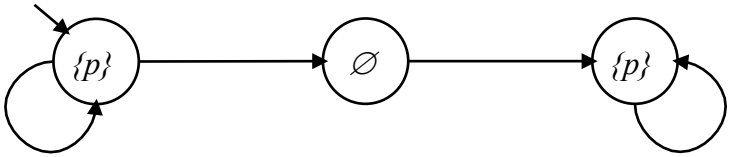


Expressiveness of CTL and LTL (3/3)

- Even though CTL considers the whole computation tree, its state-based semantics is subtly different from LTL. Thus, there are also properties expressible in LTL but not in CTL.

- Example:

  The LTL property $FG\ p$ is not expressible in CTL:



$K \models FG\ p$ but $K \not\models AF\ AG\ p$

Conclusion

- The expressiveness of CTL and LTL is incomparable; there is an overlap, and each logic can express properties that the other cannot.

- Remark: There is a logic, called CTL*, that combines the expressiveness of CTL and LTL. However, we will not deal with it in this course.

Second Practice.