!!!!

Ces slides ne sont qu’une adaptation des slides du livre :

Concurrency: State Models & Java Programs
Jeff Magee & Jeff Kramer

Concurrence

Les origines de la programmation concurrente sont liés à l'évolution technologique des systèmes d'exploitation.

Un style de programmation dont le but est la collaboration des processus.

Trois types de concurrence :

- -Disjointe : pas de communication ou interaction
- -Compétitive : pour l'accès à certaines ressources partagés
- -Coopérative : pour atteindre un objectif commun

Pourquoi la concurrence ?

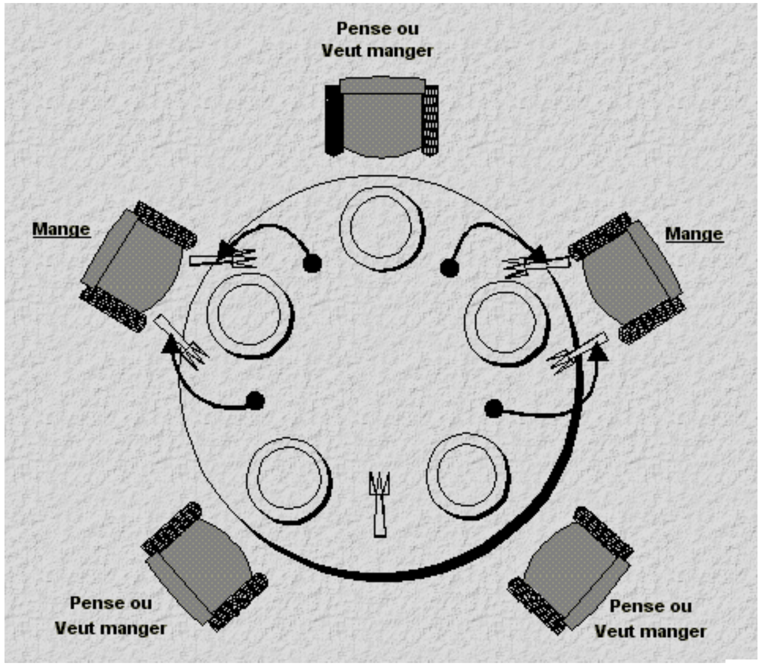
-Exécuter plusieurs tâches sur un seul processeur
-Répartir une application sur plusieurs processeurs
-Transmettre de l'information entre composants
-Connecter des ordinateurs ou équipements proches
-Connecter des équipements lointains
-Partager de l'information à grande échelle
-Chercher de l'information à grande échelle
-Contrôler des systèmes compacts
-Coordonner des grands systèmes
-Rendre les systèmes robustes aux pannes
-…

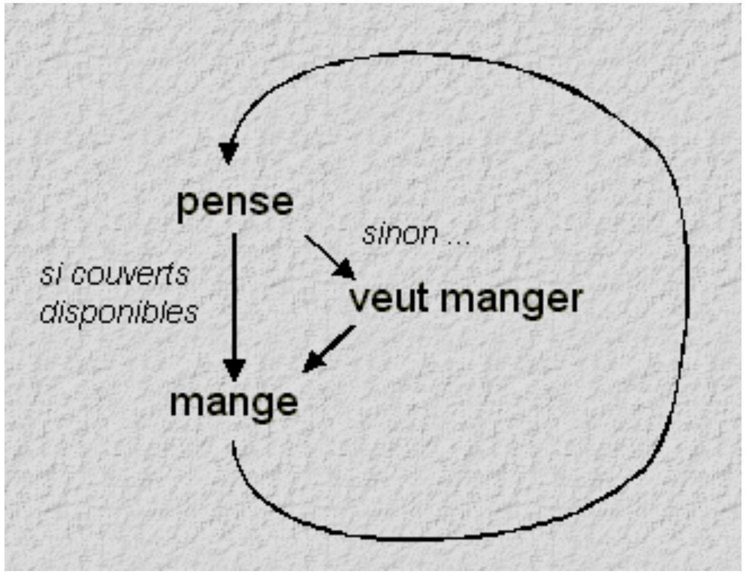
Le problème du ou parallèle

- En C, trois disjonctions booléennes sont possibles
  - OuS (e, e') = e | e' : boucle si e ou e' boucle
  - OuG (e, e') = e || e' : si e alors vrai sinon e'
    rend vrai si e rend vrai,
    même si e' boucle
  - OuD (e, e') = e' || e : symétrique
- Le parallélisme est-il possible ?
  - existe-t-il OuP (e, e') qui rend vrai si l'un de e ou e' rend vrai, même si l'autre boucle ?

Réponse : non !

Un exemple historique





L'interblocage

La famine

Nécessité d'un formalisme pour faire de preuves

Questions de base

- C’est quoi un processus ?

- Comment se synchronisent les processus ?

  Retarder l'exécution d’un processus pour satisfaire les contraintes sur l’ordre des événements d’action et perception.

  Synchrone ou Asynchrone

- Comment communiquent les processus ?

  L'exécution d’un processus influence celle des autres

  Shared variables (Monitors)

  Message passing (Réseaux de Kahn)

  HandShake (CSP)

Sections critiques

$$P = \{x + 1;\} \quad Q = \{\text{var } y = x;\ x = 2y\};$$

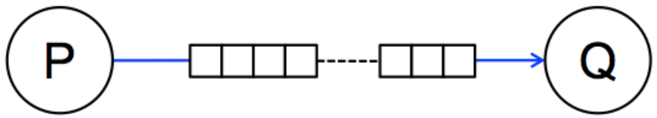
$$P \parallel Q \quad ?$$

Verrous

Sémaphores

Monitors

Réseaux de Kahn



Découplage de l'émetteur et du récepteur

files FIFO non bornées

get bloquant, **send** non-bloquant

Communicating Sequential Processes (Hoare)

P and Q sont asynchrones, mais à un moment donnée ils communiquent de façon localement synchrone.

Processus

Evénements

$(e \longrightarrow P)$

CSP

αP ={up,rigth}

P = rigth→(up → (rigth→ (rigth→STOP)))

$\alpha CLOCK$ ={tick}

CLOCK =(tick →CLOCK)

CSP

$$
\begin{aligned}
P ::= \quad & STOP \mid SKIP \mid \\
& (e \rightarrow P) \mid \\
& P \parallel P \mid \\
& (c!e \rightarrow P) \mid (c?x \rightarrow P(x))
\end{aligned}
$$

$$
\frac{(c!v \rightarrow P) \parallel (c?x \rightarrow Q(x))}{c!v \rightarrow (P \parallel Q(v))}
$$

Modélisation de systèmes concurrents

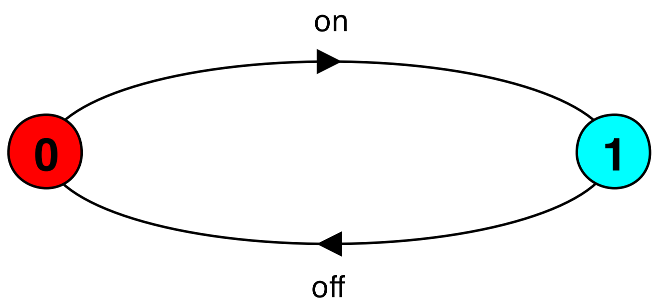
Les modelés sont décrits en utilisant des machines à états,

LTS = Labelled Transition System (forme graphique)

FSP = Finite State Processes (forme algébrique)

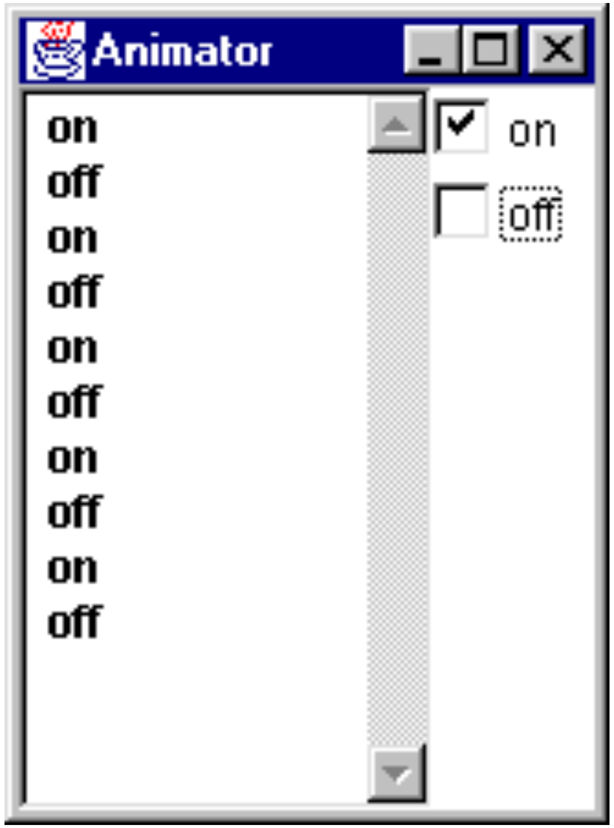
LTSA = Labelled Transition System Analizer (test)

Exemple



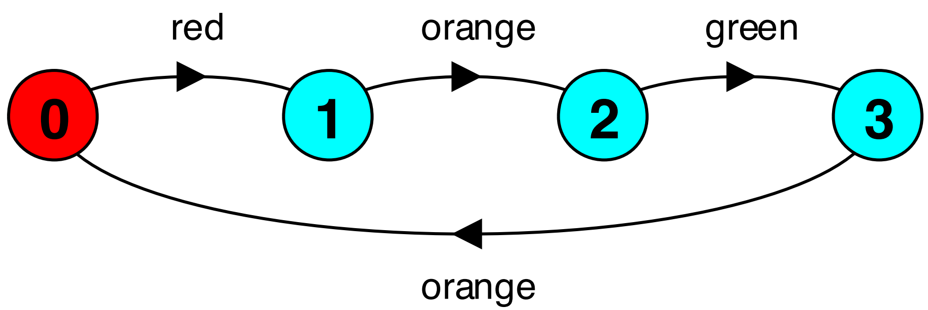
on→off→on→off→...

```
SWITCH = OFF,
OFF    = (on -> ON),
ON     = (off-> OFF).
```



Un feu rouge (action prefix)

(x-> P)



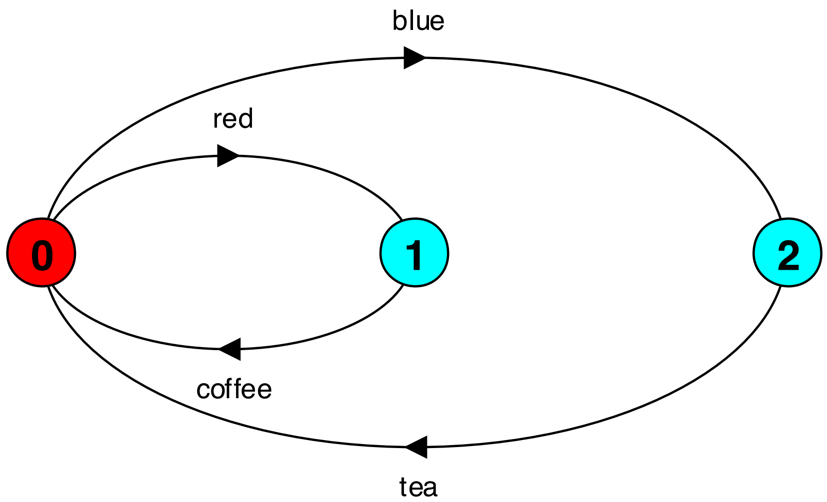
red→orange→green→orange→red→orange→green …

```
TRAFFICLIGHT = (red->orange->green->orange
                   -> TRAFFICLIGHT).
```

Coffee or tea (le choix)

(x-> P | y-> Q)

```
DRINKS = (red->coffee->DRINKS
         |blue->tea->DRINKS
         ).
```

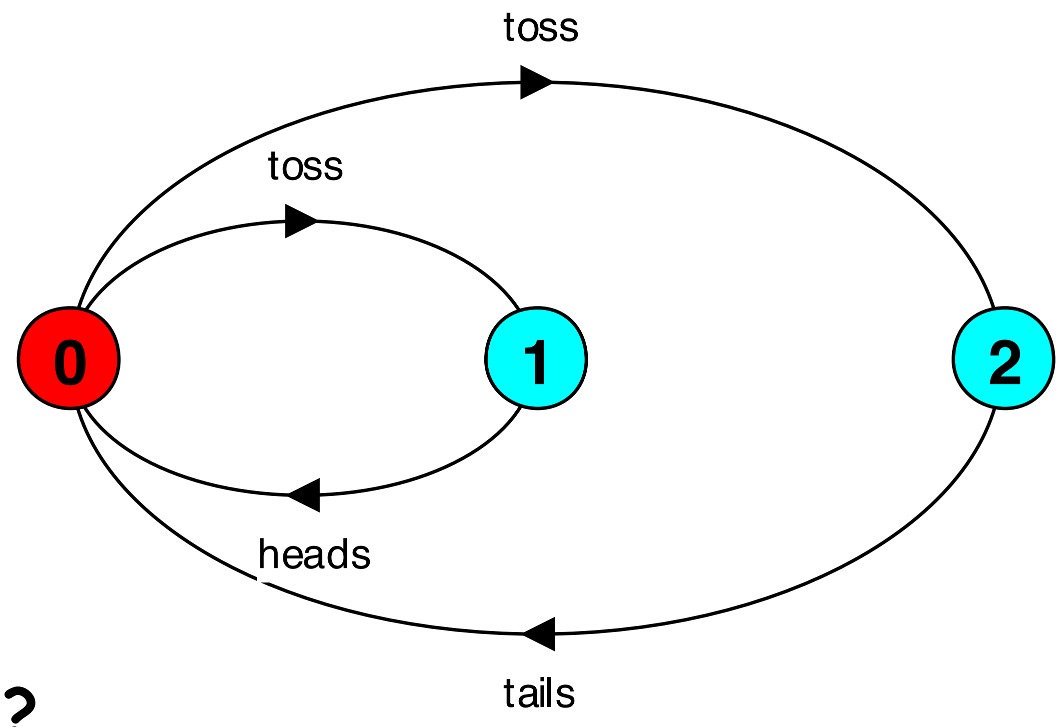


Traces ?

Pil ou face (le choix ND)

(x-> P | x-> Q)

```
COIN = (toss->HEADS|toss->TAILS),
HEADS= (heads->COIN),
TAILS= (tails->COIN).
```



Traces ?

Indexation des processus et actions

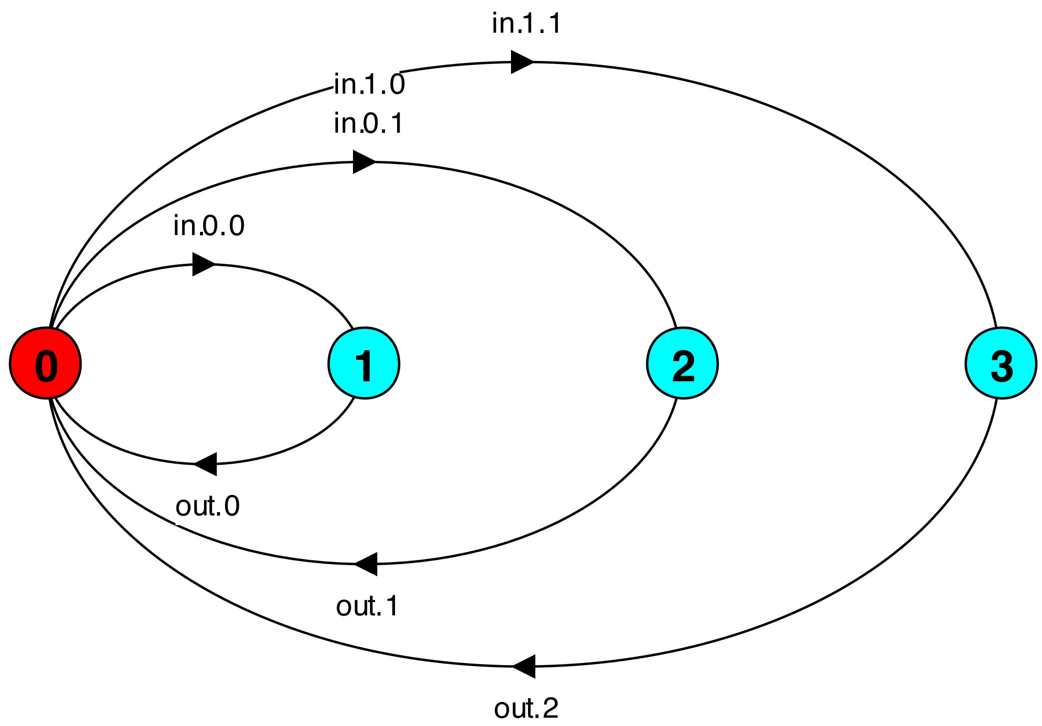
```
BUFF = (in[0]->out[0]->BUFF
       |in[1]->out[1]->BUFF
       |in[2]->out[2]->BUFF
       |in[3]->out[3]->BUFF
       ).
```

```
BUFF = (in[i:0..3]->out[i]-> BUFF).
```

```
BUFF(N=3) = (in[i:0..N]->out[i]-> BUFF).
```

LST ?

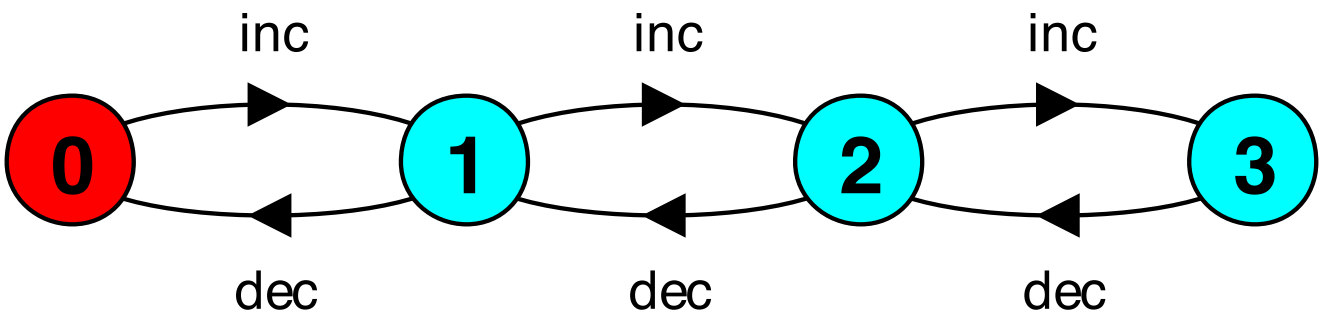
Constant and range declaration




```
const N = 1
range T = 0..N
range R = 0..2*N

SUM          = (in[a:T][b:T]->TOTAL[a+b]),
TOTAL[s:R]   = (out[s]->SUM).
```

Un compteur (action avec condition)

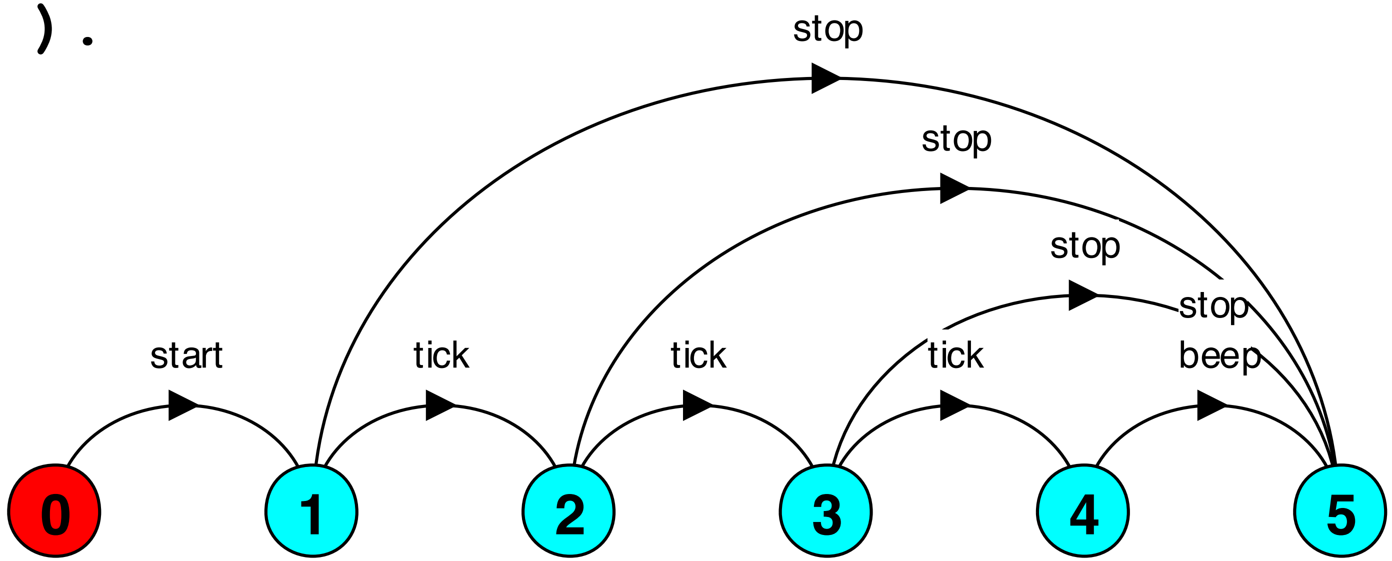
(when B x -> P | y -> Q)




```
COUNTER (N=3)          = COUNT[0],
COUNT[i:0..N] = (when(i<N) inc->COUNT[i+1]
                |when(i>0) dec->COUNT[i-1]
                ).
```

Un exemple + complexe

```
const N = 3
COUNTDOWNBEEP    = (start->COUNTDOWN[N]),
COUNTDOWN[i:0..N] =
          (when(i>0) tick->COUNTDOWN[i-1]
          |when(i==0)beep->STOP
          |stop->STOP
          ).
```

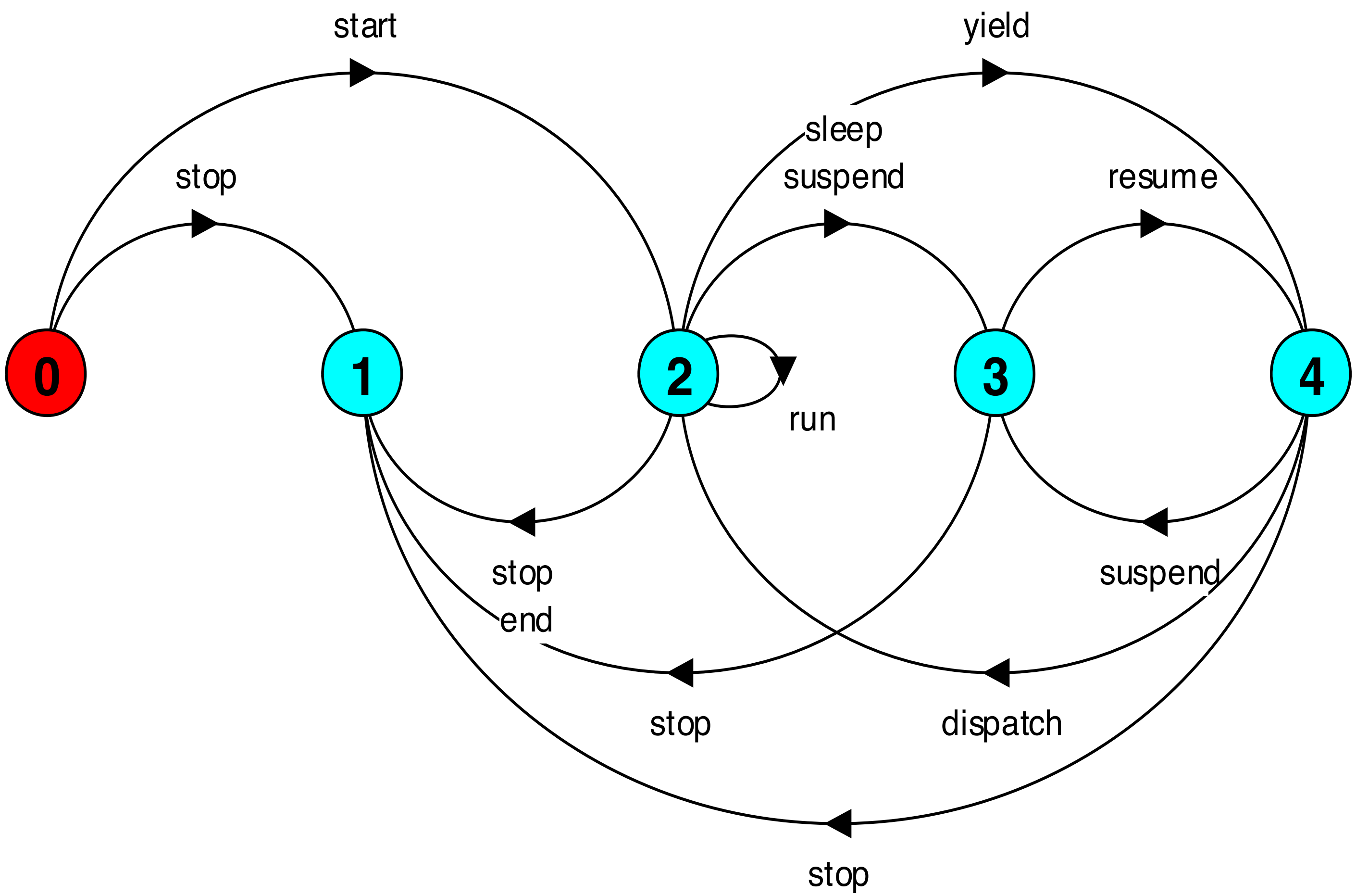


Implémentation

FSP spécification des threads

```
THREAD        = CREATED,
CREATED       = (start           ->RUNNING
                |stop            ->TERMINATED),
RUNNING       = ({suspend,sleep}->NON_RUNNABLE
                |yield           ->RUNNABLE
                |{stop,end}      ->TERMINATED
                |run             ->RUNNING),
RUNNABLE      = (suspend         ->NON_RUNNABLE
                |dispatch        ->RUNNING
                |stop            ->TERMINATED),
NON_RUNNABLE = (resume           ->RUNNABLE
                |stop            ->TERMINATED),
TERMINATED    = STOP.
```

FSP spécification

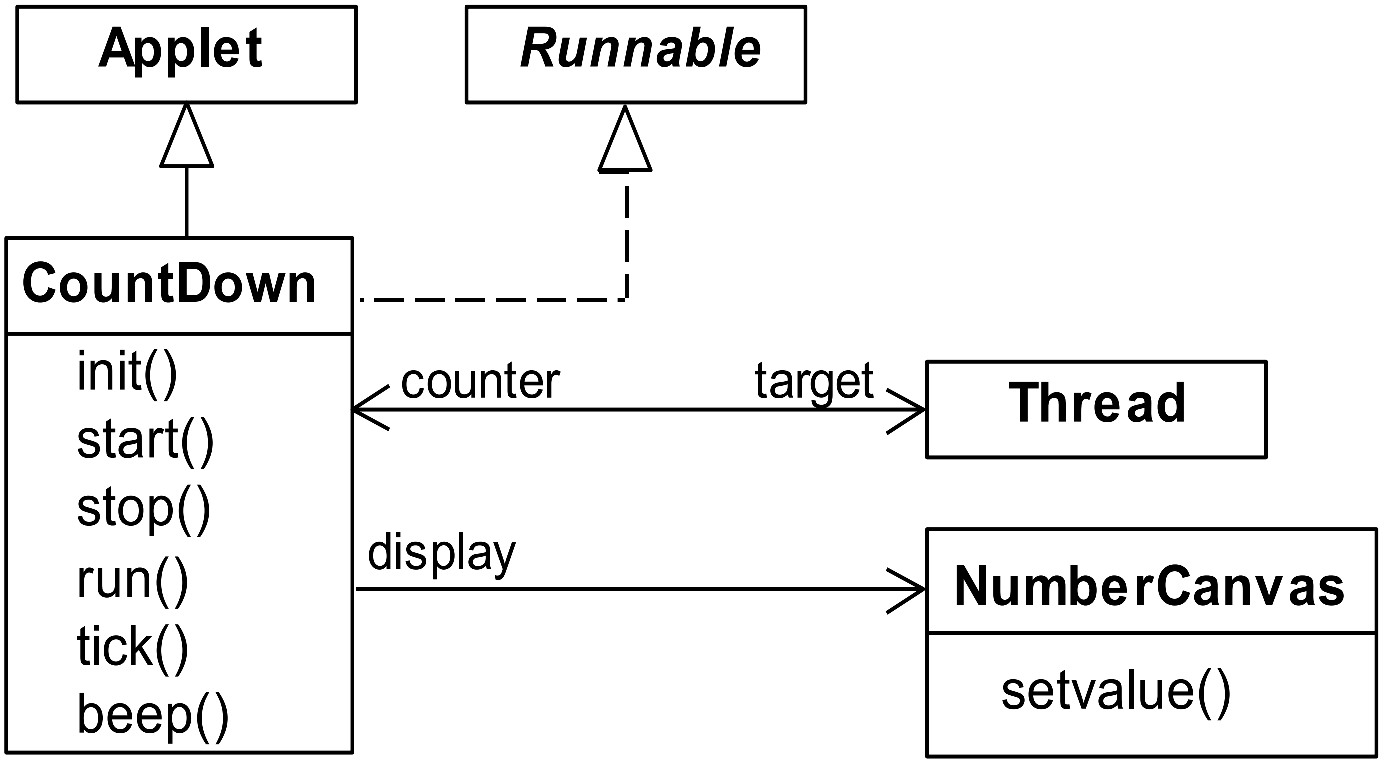


[0 … 4] = [**CREATED**, **TERMINATED**, **RUNNING**, **NON-RUNNABLE**, **RUNNABLE**]

Countdown exemple

```
COUNTDOWN (N=3)    = (start->COUNTDOWN[N]),
COUNTDOWN[i:0..N] =
          (when(i>0) tick->COUNTDOWN[i-1]
          |when(i==0)beep->STOP
          |stop->STOP
          ).
```

Diagramme


Applet
Runnable
CountDown
init()
start()
stop()
run()
tick()
beep()
counter
target
Thread
display
NumberCanvas
setvalue()

CountDown Class

```
public class CountDown
                implements Runnable {
  Thread counter; int i;
  final static int N = 10;


  public void start() {...}
  public void stop()  {...}
  private void tick() {...}
  private void beep() {...}

  public void run()   {...}

}
```

CountDown Class

```
public void start() {
   counter = new Thread(this);
   i = N; counter.start();
 }

 public void stop() {
   counter = null;
 }

 public void run() {
   while(true) {
     if (counter == null) return;
     if (i>0)  { tick(); --i; }
     if (i==0) { beep(); return;}
   }
 }
```

Composition paralèlle - interleaving

(P||Q) représente l'exécution concurrente de P et Q.

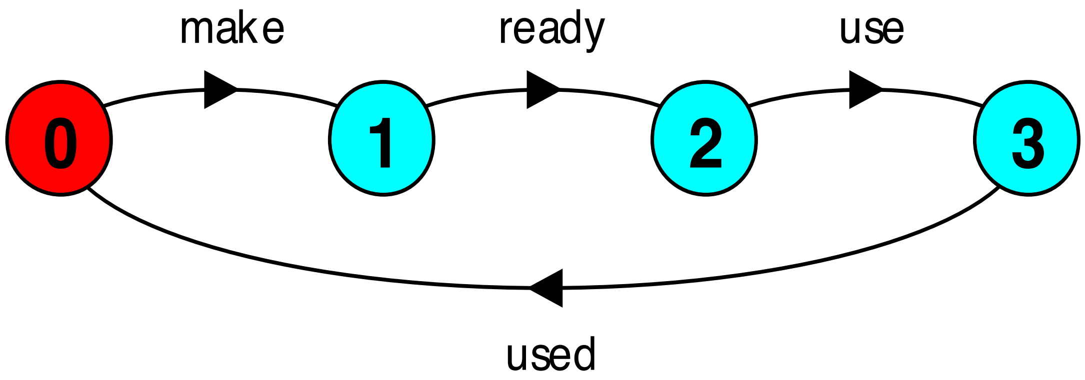
Commutative: `(P||Q) = (Q||P)`
Associative: `(P||(Q||R)) = ((P||Q)||R)`
`= (P||Q||R)`

```
PLAT = (rouge->STOP).
FIN= (blanc->armagnac->STOP).

||ABOIRE= (PLAT || FIN).
```

rouge→blanc→armagnac
blanc→rouge→armagnac
blanc→armagnac→rouge

Handshake

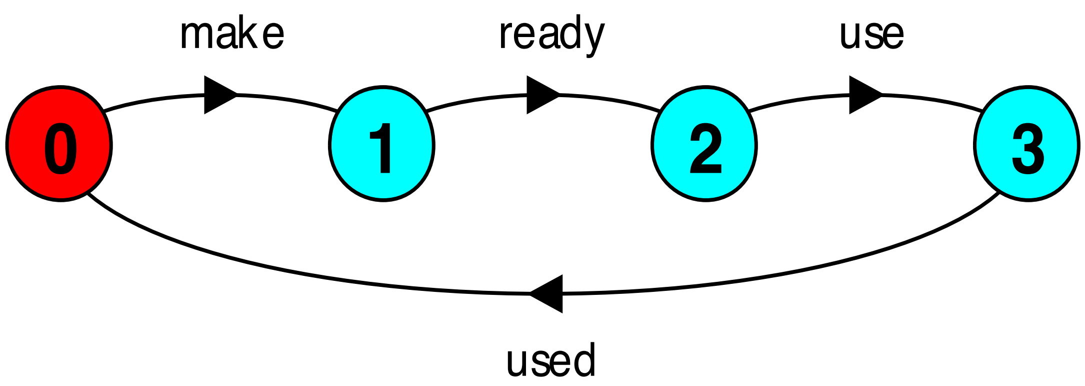
```
MAKERv2 = (make->ready->used->MAKERv2).
USERv2  = (ready->use->used ->USERv2).

||MAKER_USERv2 = (MAKERv2 || USERv2).
```



4 states

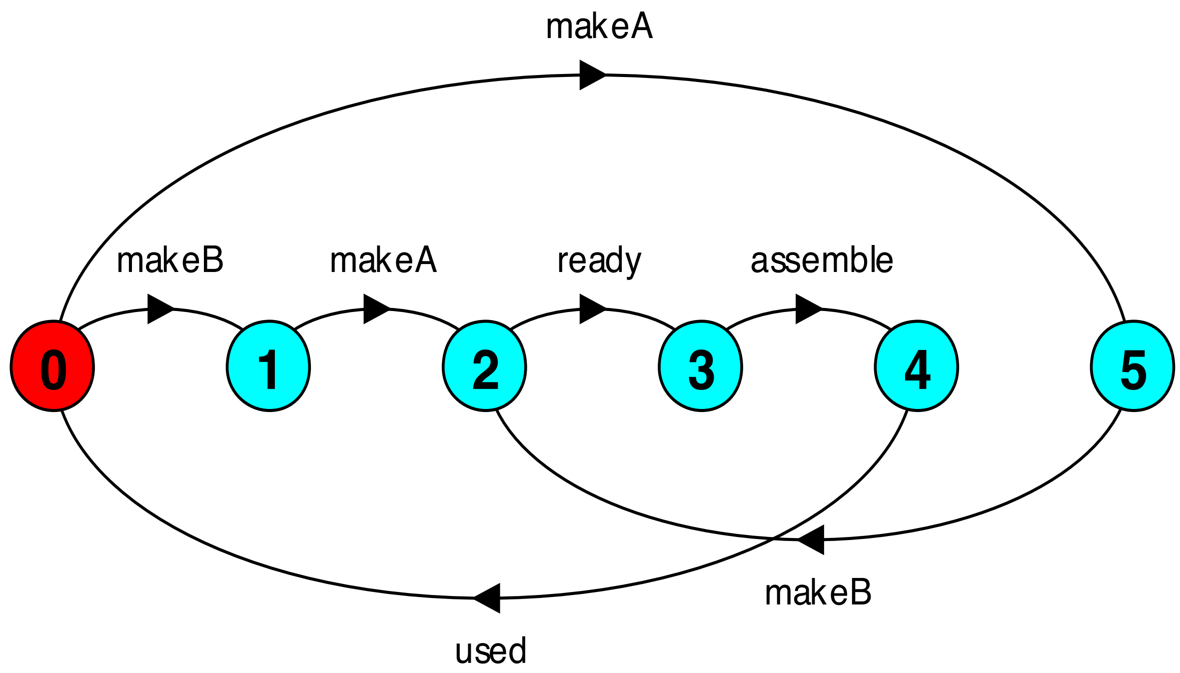
handshake

```
MAKERv2 = (make->ready->used->MAKERv2).
USERv2  = (ready->use->used ->USERv2).

||MAKER_USERv2 = (MAKERv2 || USERv2).
```



multiple processes

```
MAKE_A    = (makeA->ready->used->MAKE_A).
MAKE_B    = (makeB->ready->used->MAKE_B).
ASSEMBLE = (ready->assemble->used->ASSEMBLE).

||FACTORY = (MAKE_A || MAKE_B || ASSEMBLE).
```



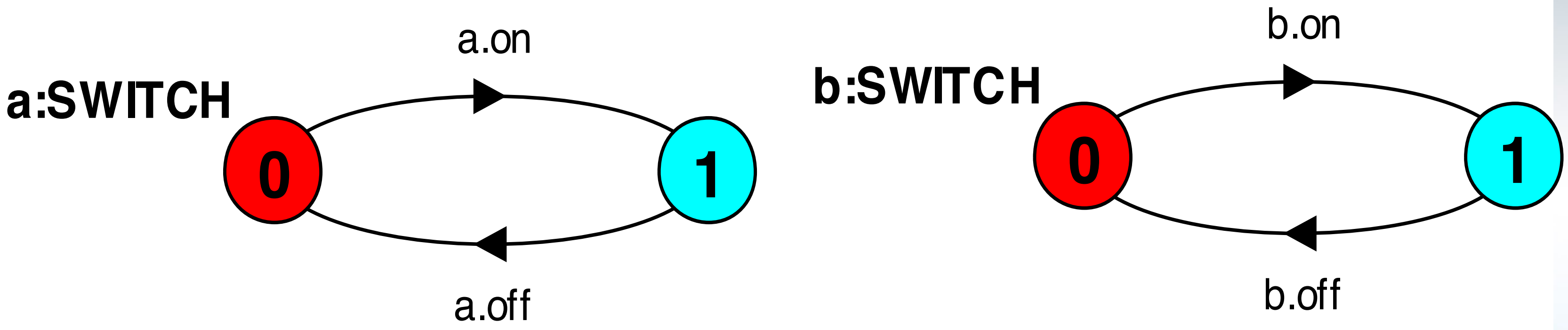
composite processes

```
||MAKERS = (MAKE_A || MAKE_B).

||FACTORY = (MAKERS || ASSEMBLE).
```

process labeling a:P

Deux **instances** d'un switch

```
SWITCH =  (on->off->SWITCH).
```




```
||TWO_SWITCH = (a:SWITCH || b:SWITCH).
```

An array d'**instances** de switch :

```
||SWITCHES(N=3) = (s[i:1..N]:SWITCH).
```

Labeling avec en ensemble de prefixes

{a1,..,ax}::P remplace toute action n par a1.n,…,ax.n. C'est utile pour modéliser des ressources partages

```
RESOURCE = (acquire->release->RESOURCE).
USER = (acquire->use->release->USER).

 ||RESOURCE_SHARE = (a:USER || b:USER
                     || {a,b}::RESOURCE).
```

action relabeling

Changer les noms des actions:
/{*newlabel_1*/*oldlabel_1*,… *newlabel_n*/*oldlabel_n*}.

Synchronisation sur certaines actions

```
CLIENT = (call->wait->continue->CLIENT).
SERVER = (request->service->reply->SERVER).
```

```
||CLIENT_SERVER = (CLIENT || SERVER)
                  /{call/request, reply/wait}.
```

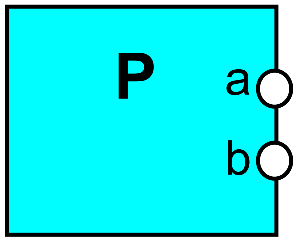
action hiding – abstraction

When applied to a process P, the hiding operator \{a1..ax} removes the action names a1..ax from the alphabet of P and makes these concealed actions "silent". These silent actions are labeled tau. Silent actions in different processes are not shared.

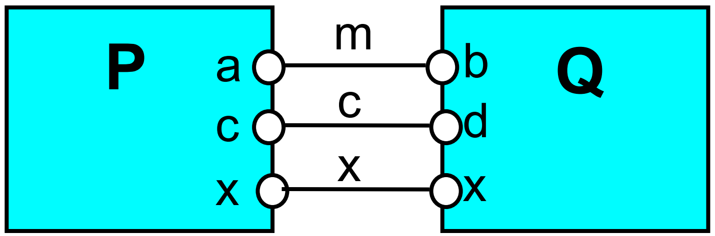
Sometimes it is more convenient to specify the set of labels to be exposed....

> When applied to a process P, the interface operator @{a1..ax} hides all actions in the alphabet of P not labeled in the set a1..ax.

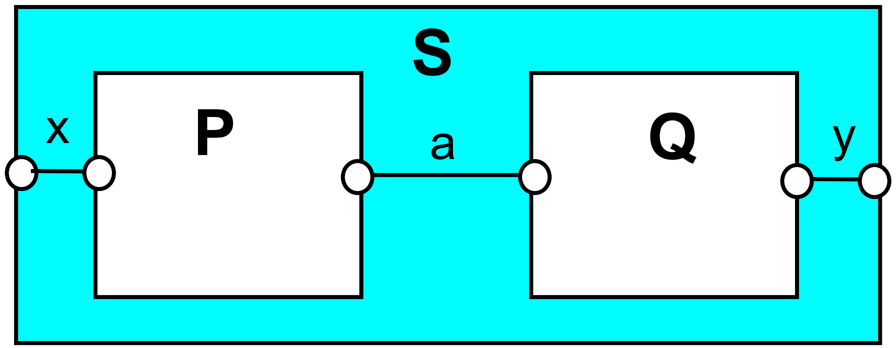
structure diagrams



Process P with alphabet {a,b}.



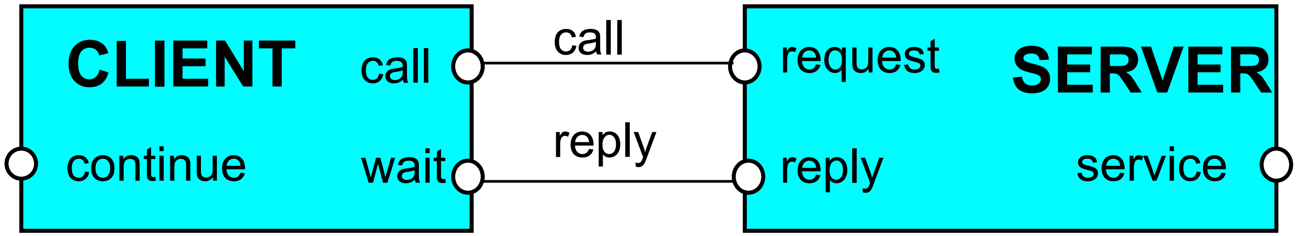
Parallel Composition (P||Q) / {m/a,m/b,c/d}



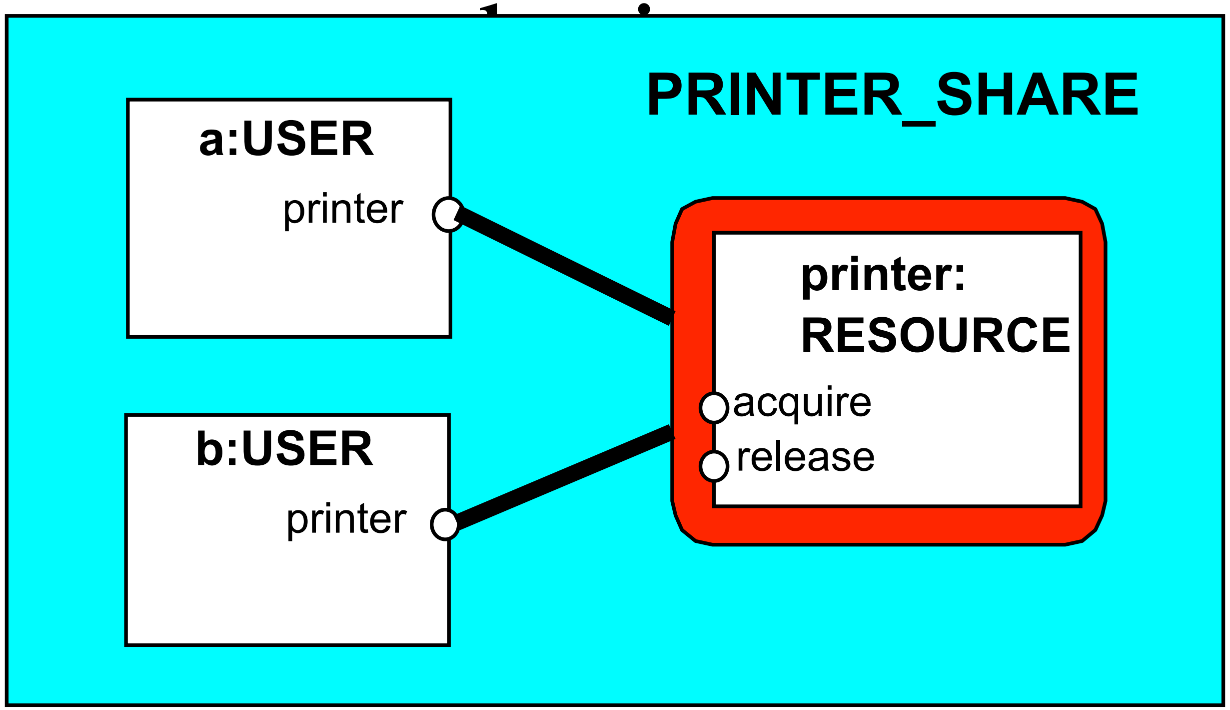
Composite process ||S = (P||Q) @ {x,y}

structure diagrams

Structure diagram for **`CLIENT_SERVER`** ?

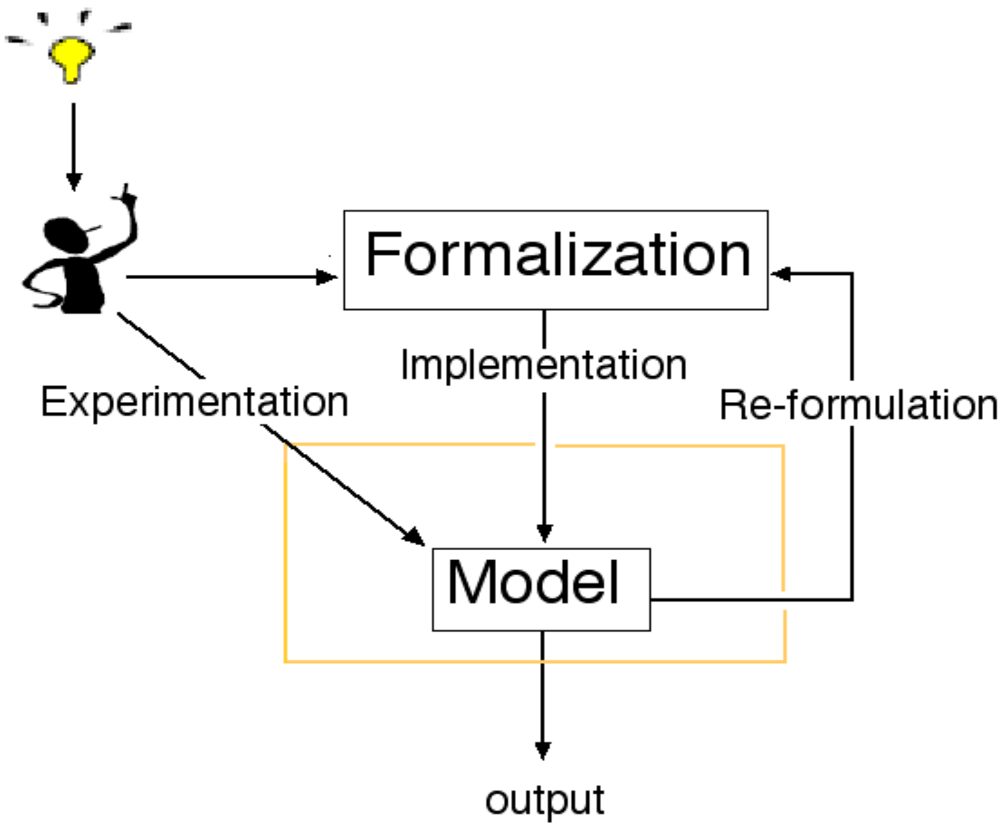


structure diagrams - resource




```
RESOURCE = (acquire->release->RESOURCE).
USER =     (printer.acquire->use
                 ->printer.release->USER).

||PRINTER_SHARE
= (a:USER||b:USER||{a,b}::printer:RESOURCE).
```

Pourquoi modéliser



Mémorisation

Condensation

Ordonnancement du savoir

Calcul

3 utilisations des LST

- Analyse de deadlock.
- Propriétés de sûreté.
- Propriétés de vivacité.

Deadlock: 4 conditions nécessaires et suffisantes

- 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.

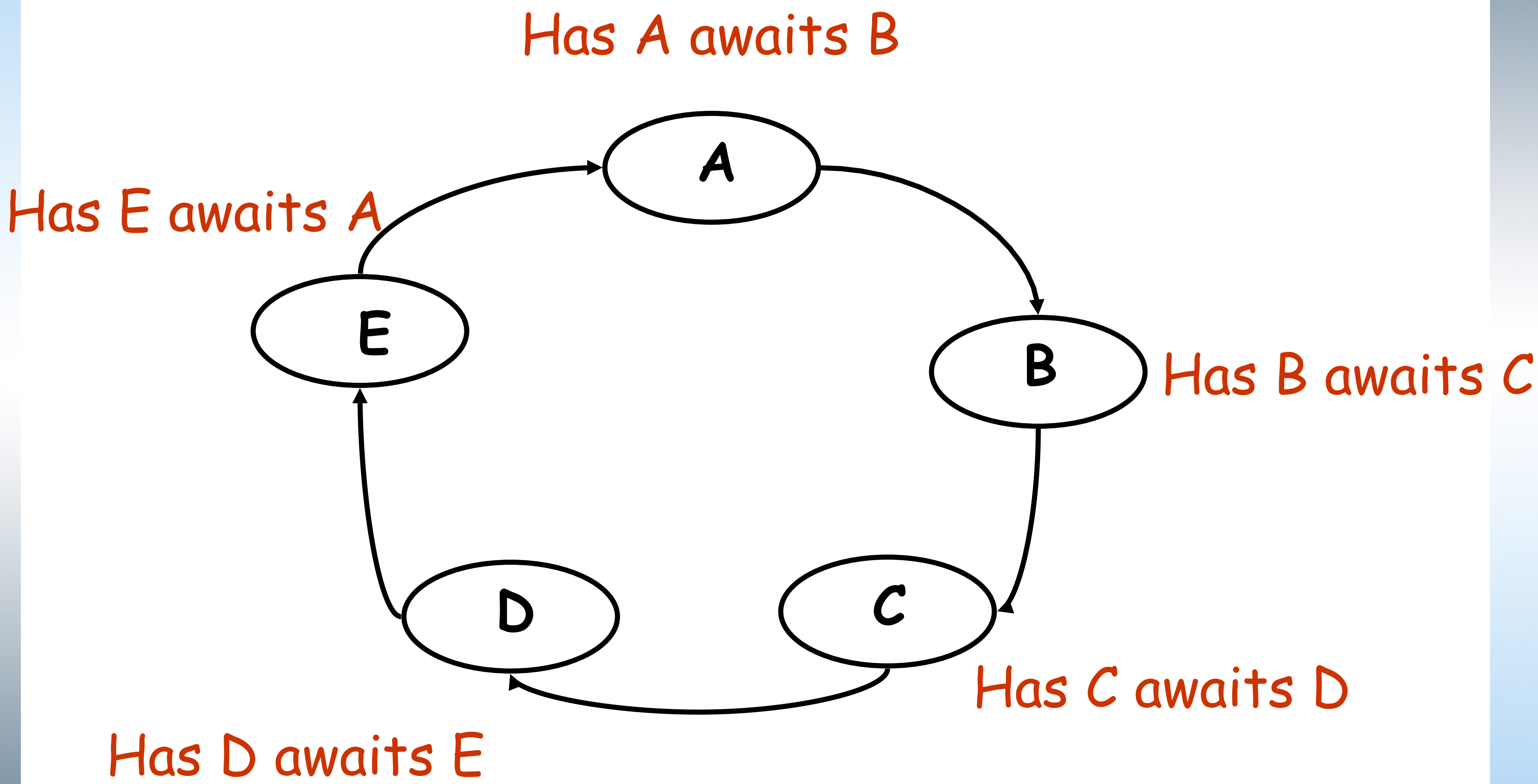
- No pre-emption:

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. {P0, .. Pn} Pi wait for r in Pi+1 and Pn wait for r in P0

Wait-for cycle



Prévention

♦Indirecte

Eliminer une des trois conditions.

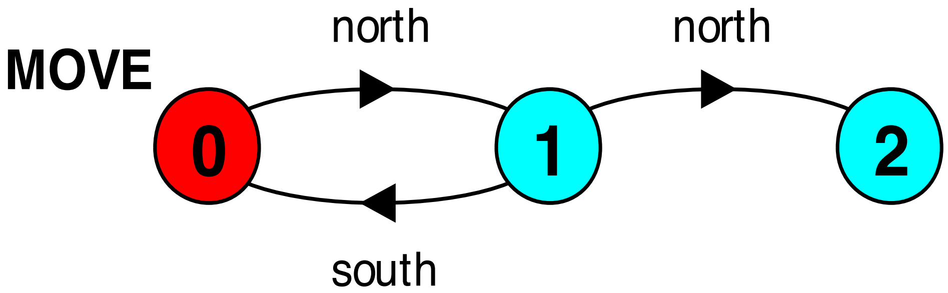
♦Directe

Eviter l'apparition du cycle (l'algorithme du banquier)

Deadlock analysis

- deadlocked state is one with no outgoing transitions
- in FSP: **STOP** process

```
MOVE = (north->(south->MOVE|north->STOP)).
```

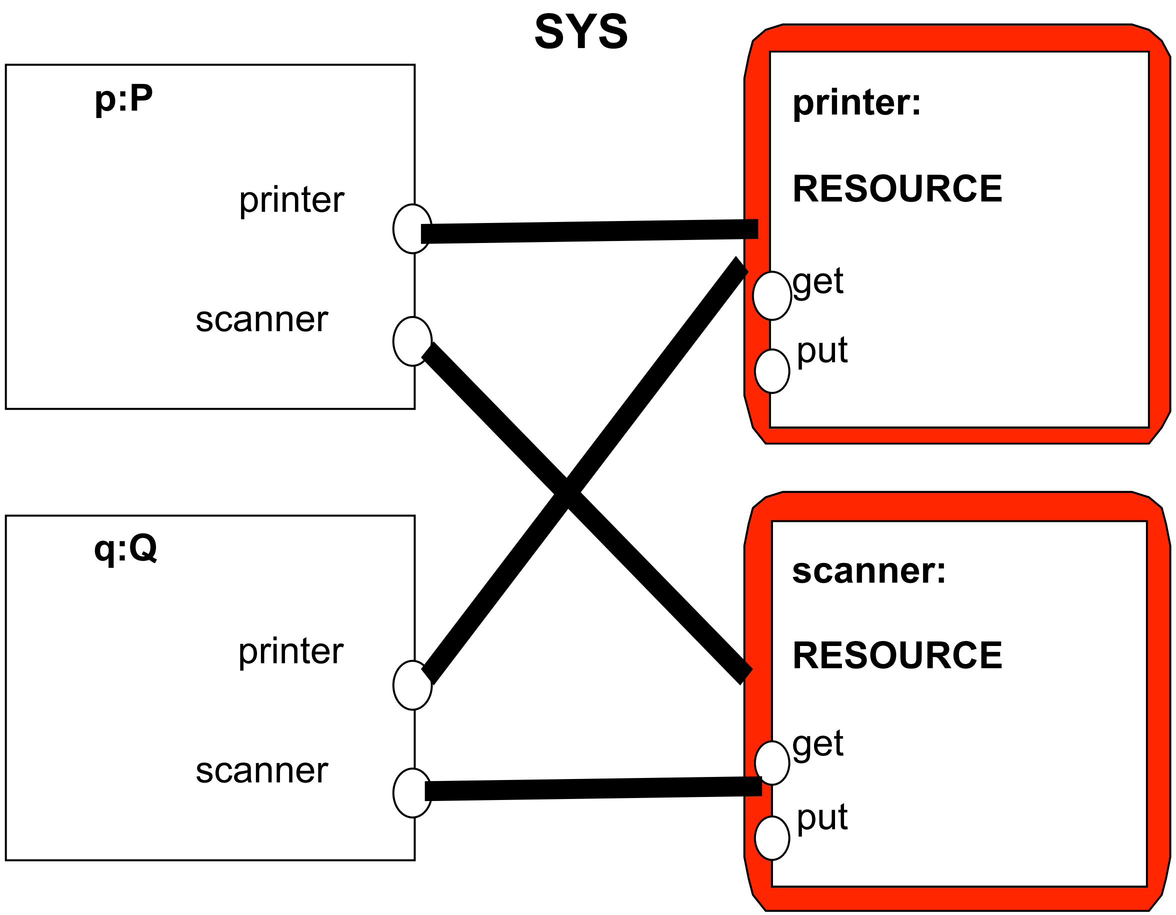


- animation to produce a trace.
- analysis using ***LTSA***:

  (shortest trace to **STOP**)

```
Trace to
DEADLOCK:
    north
    north
```

deadlock analysis - parallel composition



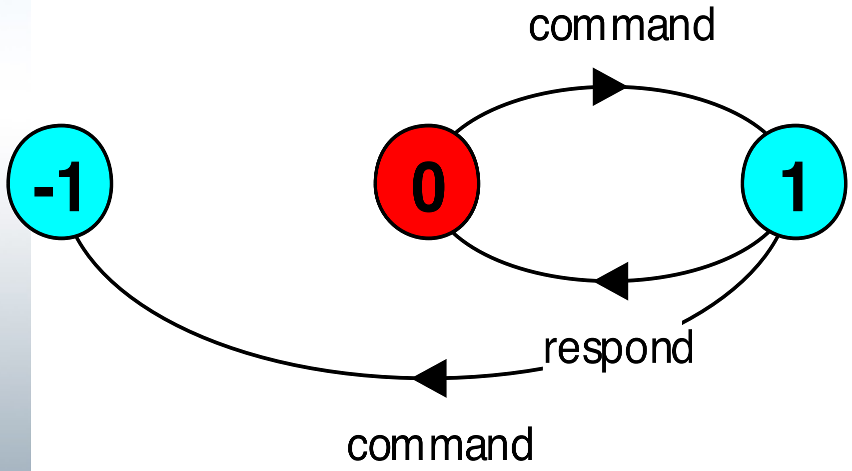
Pre-emption

```
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
             ).
```

Propriétés de sûreté.

A **safety** property asserts that nothing bad happens.

- STOP or deadlocked state (no outgoing transitions)
- ERROR process (-1) to detect erroneous behaviour




```
ACTUATOR
    =(command->ACTION),
ACTION
    =(respond->ACTUATOR
     |command->ERROR).
```

- analysis using LTSA:
  (shortest trace)

```
Trace to ERROR:
   command
   command
```

Tic tac

```
Tic = (tic->Tic).
Tac = (tac->Tac).

||TicTac1 = (Tic || Tac).


Tic2 = (tic->sync->sync->Tic2).
Tac2 = (sync->tac->sync->Tac2).

||TicTac2 = (Tic2 || Tac2).
||TicTac2 = (Tic2 || Tac2)\{sync}.
||TicTac2 = (Tic2 || Tac2)@{tic,tac}.
```

Tic tac

```
TTest1 = (tic->TTest1_1
        |tac->TTest1),
TTest1_1 = (tic->ERROR
          |tac->TTest1).

TTest2 = (tac->TTest2_1
        |tic->TTest2),
TTest2_1 = (tac->ERROR
          |tic->TTest2).

//||TicTac3 = (Tic2 || Tac2 || TTest1 || TTest2).
```

Propriétés de vivacité.

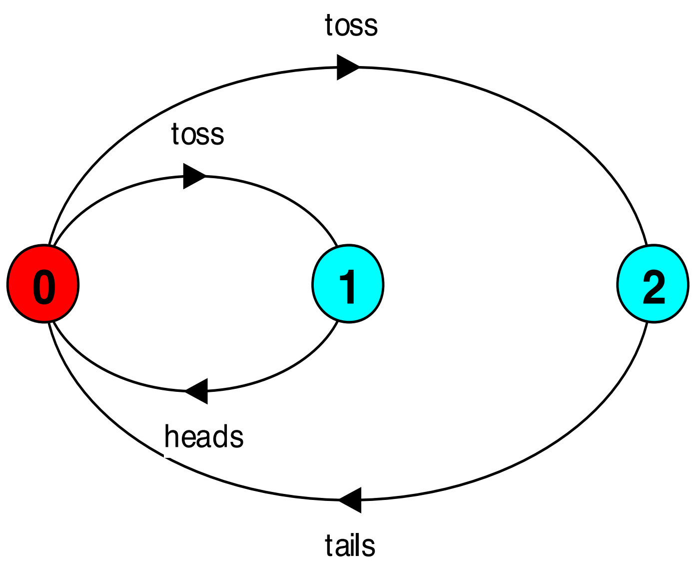
A **safety** property asserts that nothing bad happens.

A **liveness** property asserts that something good ***eventually*** happens.

A progress property asserts that it is *always* the case that an action is *eventually* executed. Progress is the opposite of *starvation*, the name given to a concurrent programming situation in which an action is never executed.

Progress properties - fair choice

Un système progresse vis-à-vis d'une action a si et seulement si, dans toute trace (ou exécution) infinie du système le nombre d'occurrences de l'action a est aussi infini.




```
COIN =(toss->heads->COIN
      |toss->tails->COIN).
```

Progress Pile = {tails}
Progress Face = {heads}

Progress properties

Tout d'abord, les états sont regroupés dans des composantes fortement connexes
(algorithme SCC : strongly connected component).
Toute transition sortante "boucle" dans la composante connexe.

Pour chaque propriété de progression, on regarde si tous les cycles contiennent
l'action considérée. Sinon l'action ne peut être déclenchée indéfiniement et donc
la propriété est invalidée.

Progress properties

```
TWOCOIN = (pick->COIN|pick->TRICK),
TRICK   = (toss->heads->TRICK),
COIN    = (toss->heads->COIN|toss->tails->COIN).
```

```
progress HEADS = {heads}  oui
progress TAILS = {tails}  non
```

Trois exemples

1) analyse de DEADLOCK (par parcours du LTS)

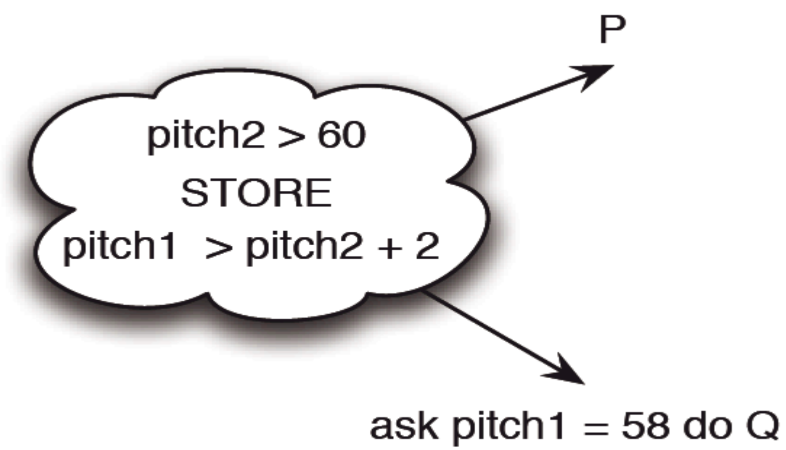
2) analyse de propriétés de sûreté (par parcours du LTS pour un système mis en parallèle avec un processus de test)

3) analyse de propriété de vivacité (par analyse des composantes fortement connexes).

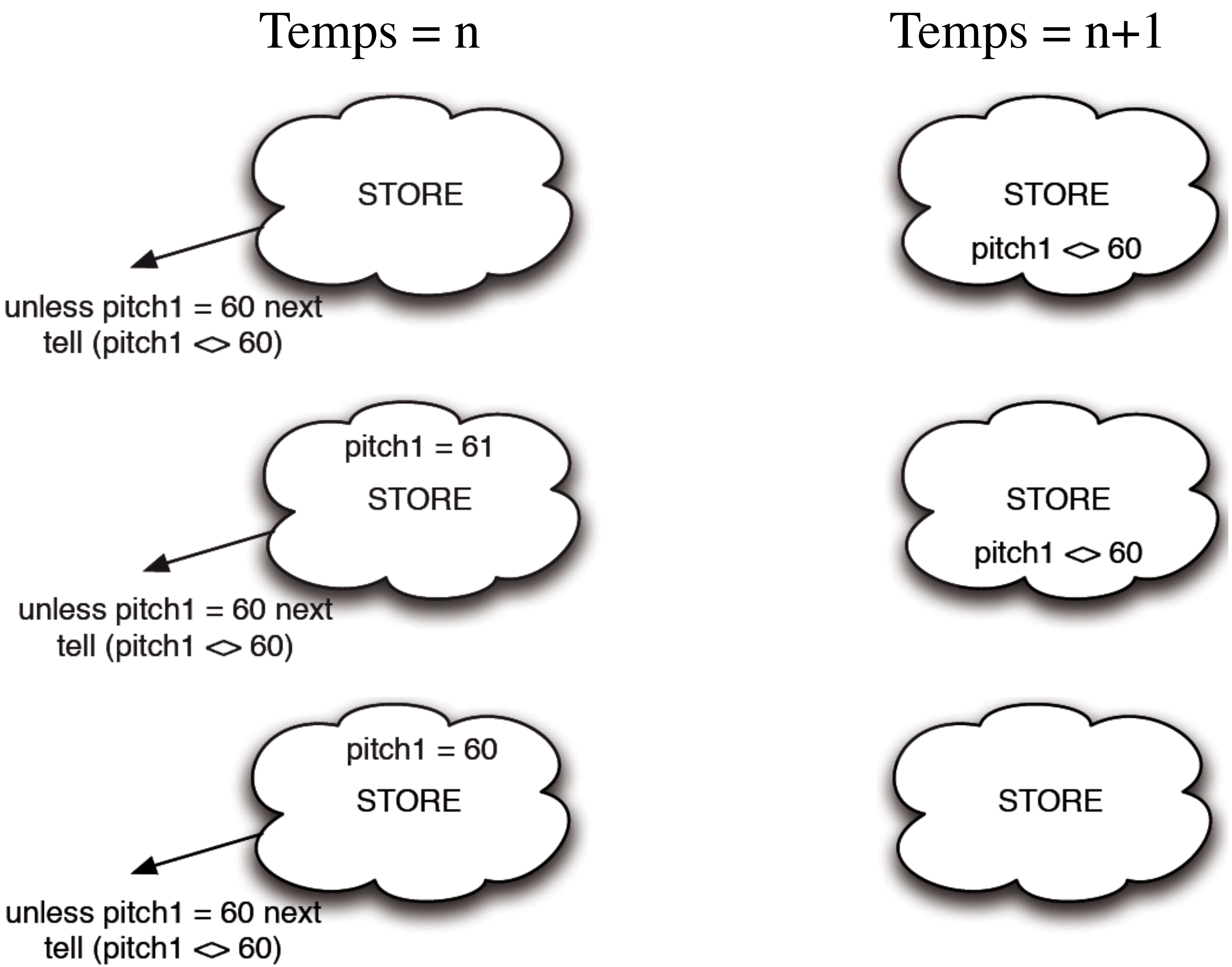
Autres paradigmes de communication
CCP
Aspect temporelles

Les agents communiquent via un *store* de contraintes

$$pitch1 > pitch2 + 2, pitch2 > 60 \models pitch1 > 58$$



TCC


Temps = n
Temps = n+1
STORE
unless pitch1 = 60 next
tell (pitch1 <> 60)
STORE
pitch1 <> 60
pitch1 = 61
STORE
unless pitch1 = 60 next
tell (pitch1 <> 60)
STORE
pitch1 <> 60
pitch1 = 60
STORE
unless pitch1 = 60 next
tell (pitch1 <> 60)
STORE