Model Checking

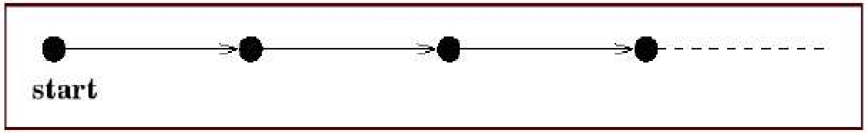
- Logiche temporali
- Automi e logiche temporali
- Modelli per sistemi concorrenti:
  - Process Algebra
  - Reti di Petri

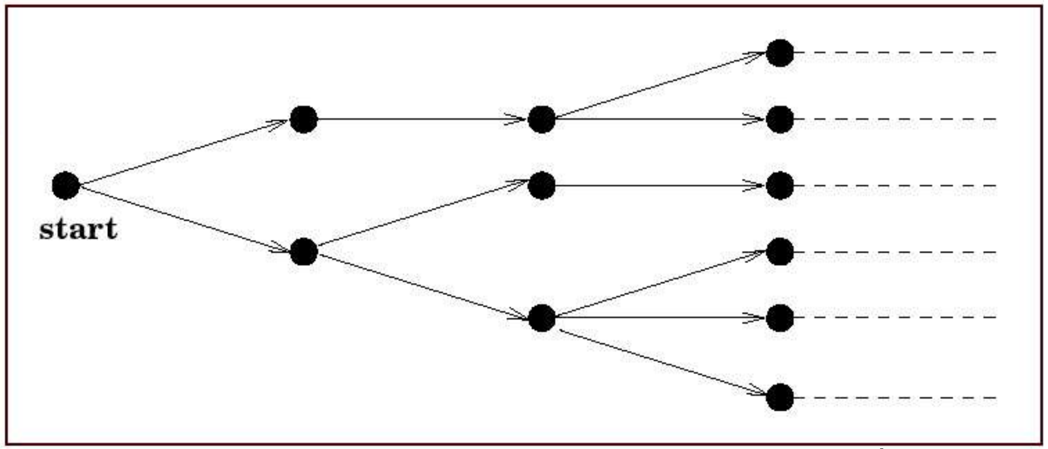
Logiche temporali

- Logiche classiche considerano un unico “mondo” sul quale esprimono proprietà che sono sempre vere o false
  - Esempio: la proposizione “è lunedì” risulta essere o sempre vera o sempre falsa
- Ma i sistemi software sono sistemi “dinamici”
  - Esempio: la proposizione “la variabile x contiene il valore 10” puo’ essere a volte vera a volte falsa
- Le logiche temporali permettono di esprimere proprietà temporali:
  - vere in alcuni “mondi”, false in altri

Logiche temporali

- I “mondi” considerati corrispondono a diversi momenti temporali
- Possono esistere diversi modi per mettere in relazione questi “mondi”
  - Linear time:
  - Branching time





Sequenze lineari di “mondi”

- Una sequenza linear time di “mondi” è quindi una sequenza di insiemi di proposizioni che indicano i fatti veri in quell’istante temporale
- Esempi:

$$\bullet \longrightarrow \bullet \longrightarrow \bullet \longrightarrow \bullet \longrightarrow \bullet \longrightarrow \bullet \ \cdots$$

{lun} {mar} {mer} {gio} {ven} {sab}

$$\bullet \longrightarrow \bullet \longrightarrow \bullet \longrightarrow \bullet \longrightarrow \bullet \longrightarrow \bullet \ \cdots$$

{s} {s,v} {p} {} {v} {s}

s=sole v=vento p=pioggia

Linear Temporal Logic (LTL)

- Estende la logica proposizionale con operatori per specificare proprietà su sequenze linear time di "mondi":

$\bigcirc\varphi$	φ is true in the *next* moment in time
$\Box\varphi$	φ is true in *all* future moments
$\Diamond\varphi$	φ is true in *some* future moment
$\varphi\, \mathcal{U}\, \psi$	φ is true *until* ψ is true

- Esempio:

$$\Box((\neg passport \vee \neg ticket) \Rightarrow \bigcirc\neg board\_flight)$$

Altri esempi

- Sistema in cui si richiede l'esecuzione di un task: tale richiesta viene ricevuta, elaborata, ed eseguita

$$\Box(requested \Rightarrow \Diamond received)$$

$$\Box(received \Rightarrow \bigcirc processed)$$

$$\Box(processed \Rightarrow \Diamond\Box done)$$

- In tale sistema, se si richiede l'esecuzione del task, questo sarà sicuramente eseguito. Formalmente la seguente proprietà, per un tale sistema, è falsa:

$$requested \wedge \Box\neg done$$

Sintassi LTL

φ, ψ	$\longrightarrow$	$p \mid$	(atomic proposition)
		$\top \mid$	(true)
		$\bot \mid$	(false)
		$\neg\varphi \mid$	(complement)
		$\varphi \wedge \psi \mid$	(conjunction)
		$\varphi \vee \psi \mid$	(disjunction)
		$\bigcirc\varphi \mid$	(next time)
		$\square\varphi \mid$	(always)
		$\diamond\varphi \mid$	(sometime)
		$\varphi\, \mathcal{U}\, \psi$	(until)

Modelli: sequenze di mondi

- Si considerino **formule di LTL** che utilizzano **proposizioni p**$\in$**P**
- Un **modello** π è una **sequenza infinita** $\pi_1, \pi_2, \pi_3, \ldots$ di "mondi" dove π_i è una funzione:

$$\pi_i : P \to \{\mathrm{T}, \mathrm{F}\}$$

- Nota: nei disegni visti si rappresenta l'insieme delle proposizioni vere

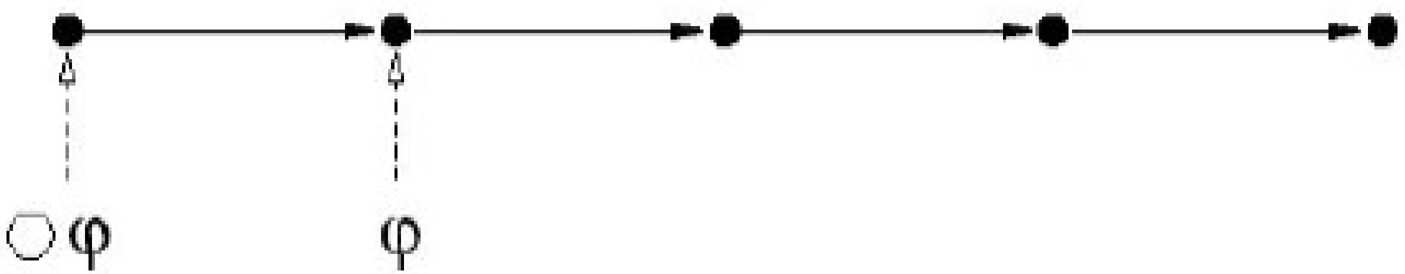
Semantica di LTL

Definizione della relazione $\pi \models \phi$

- $\pi \models p \iff \pi_1(p) = \mathtt{T}$ (unico diverso da logica proposizionale)
- $\pi \models \top$, $\pi \not\models \bot$
- $\pi \models \neg\phi \iff \textit{not}\ \pi \models \phi$
- $\pi \models \phi \wedge \phi' \iff \pi \models \phi\ \textit{and}\ \pi \models \phi'$
- $\pi \models \phi \vee \phi' \iff \pi \models \phi\ \textit{or}\ \pi \models \phi'$
- $\pi \models \phi \to \phi' \iff \pi \models \phi\ \textit{implies}\ \phi \models \phi'$
- $\ldots$

Operatore "next"

- Usato per indicare proposizioni vere nel prossimo momento temporale:

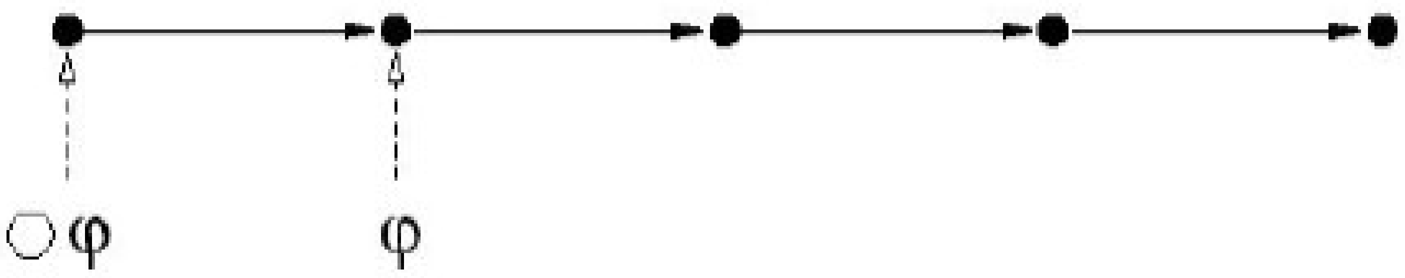


Esempi:

$$(sad \wedge \neg rich) \quad \Rightarrow \quad \bigcirc sad$$
$$((x = 0) \wedge add3) \quad \Rightarrow \quad \bigcirc (x = 3)$$

Operatore “next”

- Usato per indicare proposizioni vere nel prossimo momento temporale:



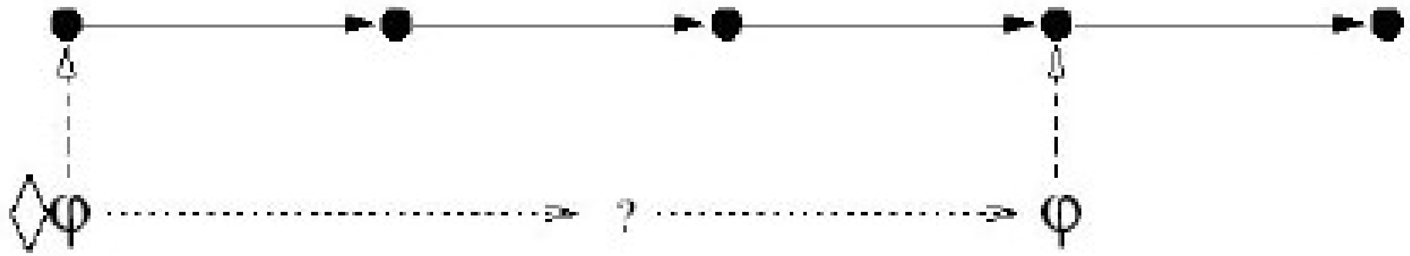
Semantica:

$$\pi \models \bigcirc\varphi \iff \pi^2 \models \varphi$$

Con π^i si rappresenta la sequenza infinita ottenuta da π partendo dal mondo i-esimo

Operatore “sometime”

- Usato per indicare proposizioni vere ora o in un qualche momento futuro:

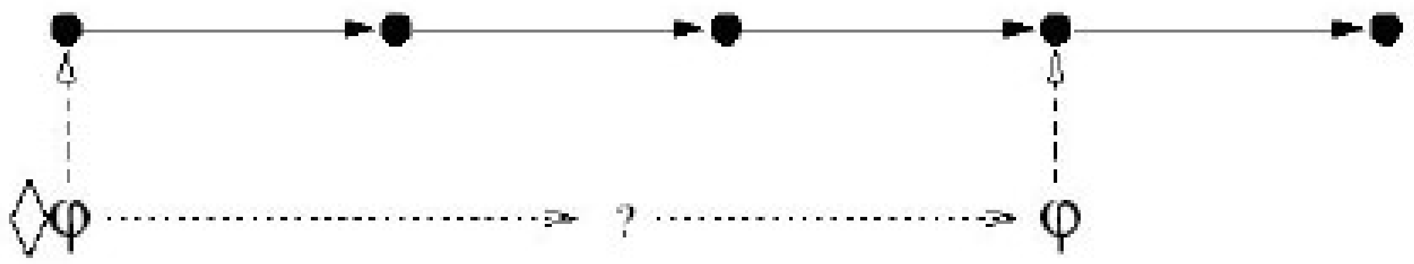


Esempi:

$$\begin{aligned} (\neg resigned \wedge sad) &\Rightarrow \Diamond famous \\ sad &\Rightarrow \Diamond happy \\ send &\Rightarrow \Diamond receive \end{aligned}$$

Operatore “sometime”

- Usato per indicare proposizioni vere ora o in un qualche momento futuro:

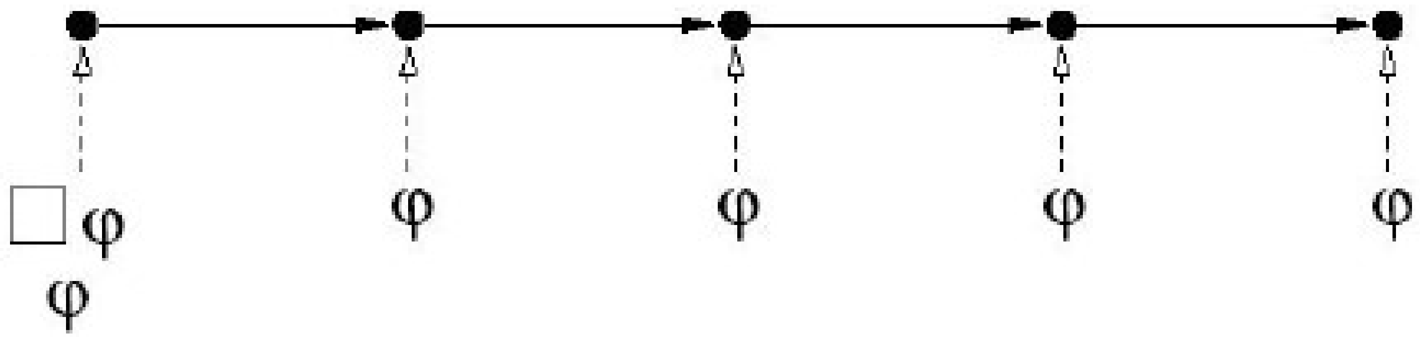


Semantica:

$$\pi \models \Diamond\varphi \Longleftrightarrow \textit{there is some } i \geq 1 \textit{ such that } \pi^i \models \varphi$$

Operatore “always”

- Usato per indicare proposizioni vere ora e in tutti i momenti futuri:

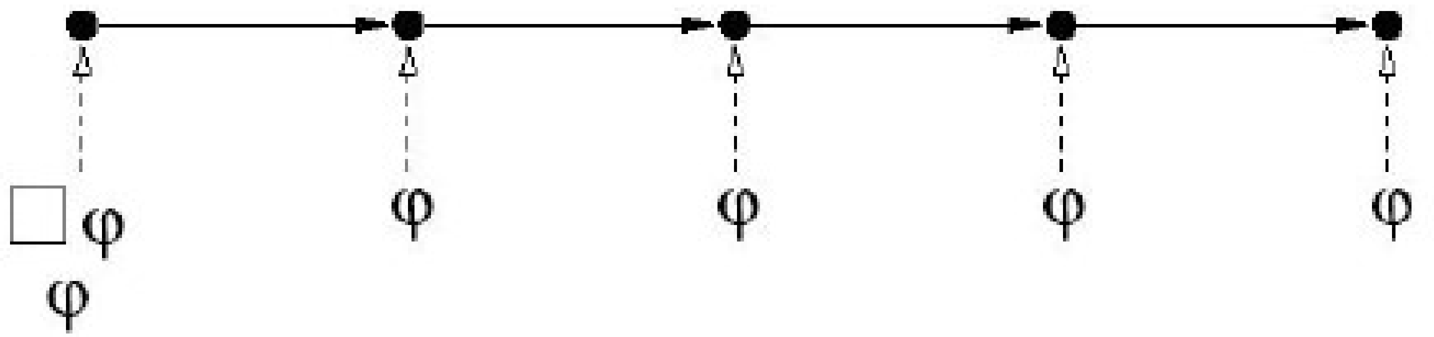


Esempi:

$$lottery\text{-}win \Rightarrow \square rich$$

Operatore “always”

- Usato per indicare proposizioni vere ora e in tutti i momenti futuri:

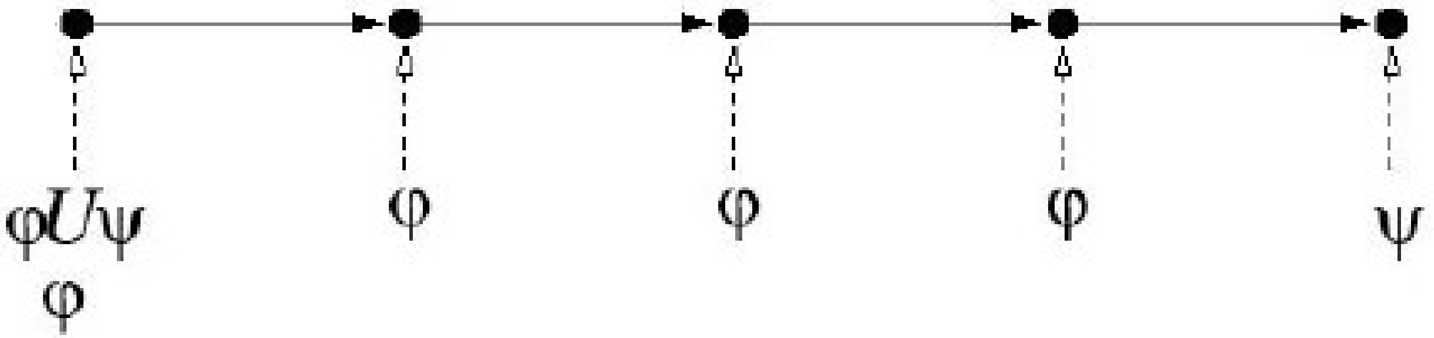


Semantica:

$$\pi \models \Box\varphi \iff \textit{for all } i \geq 1 \textit{ we have } \pi^i \models \varphi$$

Operatore "until"

- Usato per mettere in relazione due proposizioni, una vera ora ed in tutti momenti successivi fino ad un momento in cui è vera la seconda:

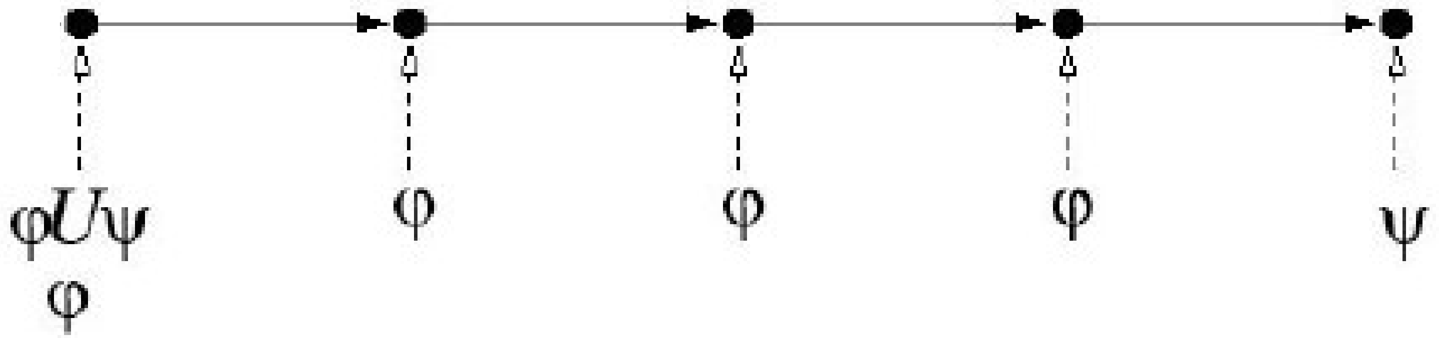


Esempi:

$$\begin{aligned} start\_lecture &\Rightarrow talk\ \mathcal{U}\ end\_lecture \\ born &\Rightarrow alive\ \mathcal{U}\ dead \\ request &\Rightarrow reply\ \mathcal{U}\ acknowledgement \end{aligned}$$

Operatore "until"

- Usato per mettere in relazione due proposizioni, una vera ora ed in tutti momenti successivi fino ad un momento in cui è vera la seconda:



Semantica:

$$\pi \models \varphi\, u\, \psi \iff \textit{there is some } i \geq 1 \textit{ such that } \pi^i \models \psi \textit{ and}$$
$$\textit{for all } j = 1 \ldots i-1 \textit{ we have } \pi^j \models \varphi$$

Operatori minimali

- "always" e "sometime" sono operatori duali:

$$\neg \Box \varphi \equiv \Diamond \neg \varphi$$

- "sometime" puo' essere espresso usando "until":

$$\Diamond \varphi \equiv \top \,\mathcal{U}\, \varphi$$

- Di conseguenza tutti gli operatori possono essere espressi usando solo "next" e "until"

Relazione con operatori classici

- "sometime" distribuisce su or, "always" su and:

$$\Diamond(\varphi \vee \psi) \equiv \Diamond\varphi \vee \Diamond\psi$$

$$\Box(\varphi \wedge \psi) \equiv \Box\varphi \wedge \Box\psi$$

- Relazione con il not:

$$\neg\bigcirc\varphi \equiv \bigcirc\neg\varphi$$

$$\neg(\varphi\, \mathcal{U}\, \psi) \equiv (\neg\psi\, \mathcal{U}\, (\neg\varphi \wedge \neg\psi)) \vee \Box\neg\psi$$

cioè φ diviene falso prima che ψ sia vero oppure ψ non è mai vero

Notazione alternativa

- Notazione standard testuale (usata da nostro libro) per gli operatori temporali: “U” per “until” e

  $\Diamond \rightsquigarrow F$ sometime in the Future

  $\Box \rightsquigarrow G$ Globally in the future

  $\bigcirc \rightsquigarrow X$ neXtime

  Oppure semplicemente (usata dal nostro tool):

  <> sometime

  [] always

  () next

Proprietà tipiche per sistemi software

- Esistono utili concetti che tipicamente sono importanti nei sistemi informatici:
  - Proprietà di “**safety**”:
    non si raggiungono mai stati con errori
  - Proprietà di “**liveness**”:
    prima o poi si eseguira’ una certa azione
  - Proprietà di “**fairness**”:
    se si richiede una cosa infinite volte,
    questa verrà eseguita infinite volte

Safety

- "qualcosa di cattivo non accadrà mai"

$$\Box \neg (reactor\_temp > 1000)$$

- Proprietà di "safety" si esprimono tipicamente così:

$$\Box \neg \ldots$$

Liveness

- "qualcosa di buono accadra'"

$$\Diamond rich$$

$$\Diamond (x > 5)$$

$$\Box (start \Rightarrow \Diamond terminate)$$

$$\Box (Trying \Rightarrow \Diamond Critical)$$

- Proprietà di "liveness" si esprimono solitamente così:

$$\Diamond \ldots.$$

Fairness

- “se qualcosa è richiesta infinite volte, allora avverrà infinite volte”

$$\Box\Diamond ready \Rightarrow \Box\Diamond run$$

LTL come frammento decidibile della logica dei predicati

- Le proposizioni $\pi_i(p)$ per $i \geq 1$ sono codificate tramite predicati unari p(i) su numeri naturali
  - costante z associata a 0 e s(i) funzione successore

Una formula LTL φ è codificata dalla formula $[\![\varphi]\!]_{s(z)}$, dove:

$$[\![p]\!]_t = p(t)$$

$$[\![\bigcirc\varphi]\!]_t = [\![\varphi]\!]_{s(t)}$$

$$[\![\Diamond\varphi]\!]_t = \exists t'\ t' \geq t \wedge [\![\varphi]\!]_{t'}$$

$$[\![\Box\varphi]\!]_t = \forall t'\ t' \geq t \rightarrow [\![\varphi]\!]_{t'}$$

$$[\![\varphi\, u\, \psi]\!]_t = \exists t'\ t' \geq t \wedge [\![\psi]\!]_{t'} \wedge \forall t''\ (t \leq t'' \wedge s(t'') \leq t') \rightarrow [\![\varphi]\!]_{t''}$$

Labeled Transition Systems e Logiche Temporali

- Un **Labeled Transition System (LTS)** è un **NFA senza** stati di accettazione **F**
  - è una quadrupla (Q, Σ, δ, q_0) con Σ insieme di etichette
- Una **traccia** di un LTS è una sequenza di etichette

$$\bullet \xrightarrow{l_1} \bullet \xrightarrow{l_2} \bullet \xrightarrow{l_3} \bullet \xrightarrow{l_4} \bullet \xrightarrow{l_5} \bullet \ \ldots$$

- Le tracce ricordano le stringhe degli NFA: iniziano dallo stato iniziale, ma ci interessano le **massimali**
  - lunghezza infinita o terminano in uno stato di **blocco** (no transizioni in uscita)

Formalmente…

- Una **traccia** σ di un **LTS** (Q, Σ, δ, q_0) è una sequenza di etichette $l_1\ l_2 \ldots$ tale che esistono stati $q_1, q_2, \ldots$ con $q_i \in \delta(q_{i-1}, l_i)$ per ogni $i = 1 \ldots len(\sigma)$
  - $len(\sigma)$ è la lunghezza della sequenza (può valere ∞)

- Una **traccia** σ è **massimale** se $len(\sigma)=\infty$ oppure se $q_{len(\sigma)}$ è di blocco, cioè $\delta(q_{len(\sigma)}, l)=\emptyset$ per ogni $l \in \Sigma$

Esprimere proprietà di LTS con formule di logica temporale

- In ogni stato ("mondo") attraversato lungo una **traccia** vale **una sola proposizione** coincidente con l'**etichetta** della transizione successiva:

$$\begin{array}{ccccccccccc}
\bullet & \xrightarrow{l_1} & \bullet & \xrightarrow{l_2} & \bullet & \xrightarrow{l_3} & \bullet & \xrightarrow{l_4} & \bullet & \xrightarrow{l_5} & \bullet \cdots \\
\{l_1\} & & \{l_2\} & & \{l_3\} & & \{l_4\} & & \{l_5\} & &
\end{array}$$

- Un LTS soddisfa una **formula** di LTL se **tutte le sue tracce massimali** soddisfano tale formula

Formalmente…

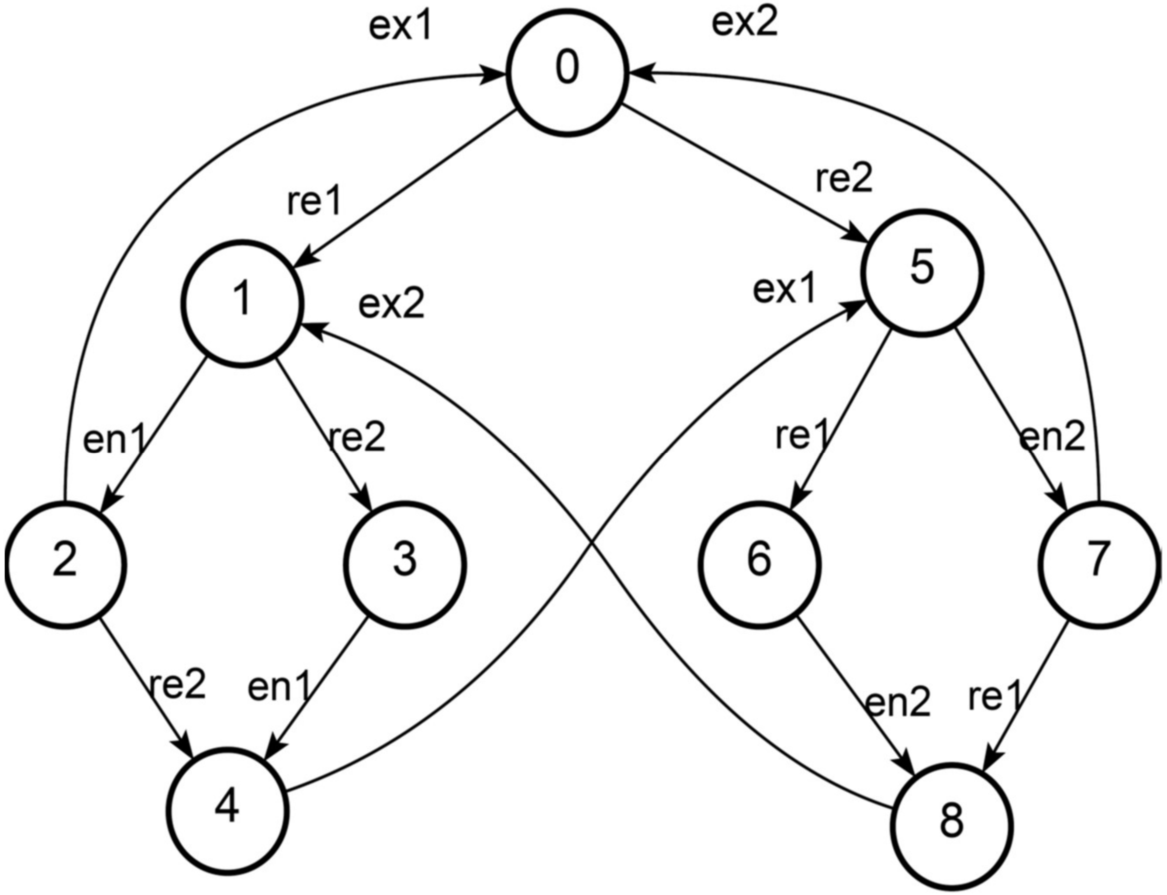
- Una **traccia** $\sigma = l_1\ l_2 \ldots$ di un LTS (Q, Σ, δ, q_0) **individua il modello** π (per formule LTL con proposizioni su Σ) definito da:

$$\pi_i(p) = \mathrm{T} \iff i \leq len(\sigma) \wedge p = l_i$$

  dove $i \geq 1$ e $p \in \Sigma$

- Un LTS soddisfa una formula ϕ di LTL se i modelli π individuati da **tutte** le sue tracce **massimali** sono tali che: $\pi \models \phi$

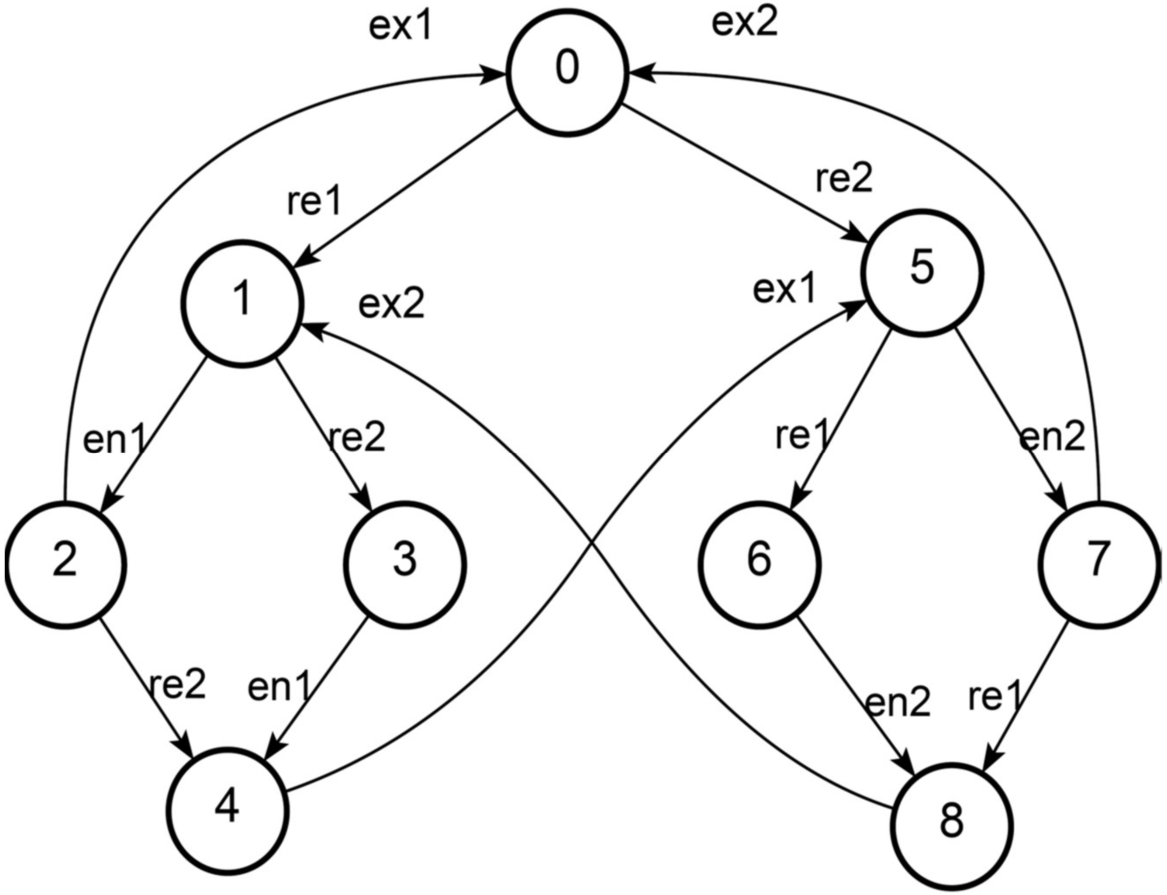
Esempio: mutua esclusione



- re1/re2: utente1/utente2 richiedono di entrare in sez. critica
- en1/en2: utente1/utente2 entrano in sezione critica
- ex1/ex2: utente1/utente2 escono da sez. critica

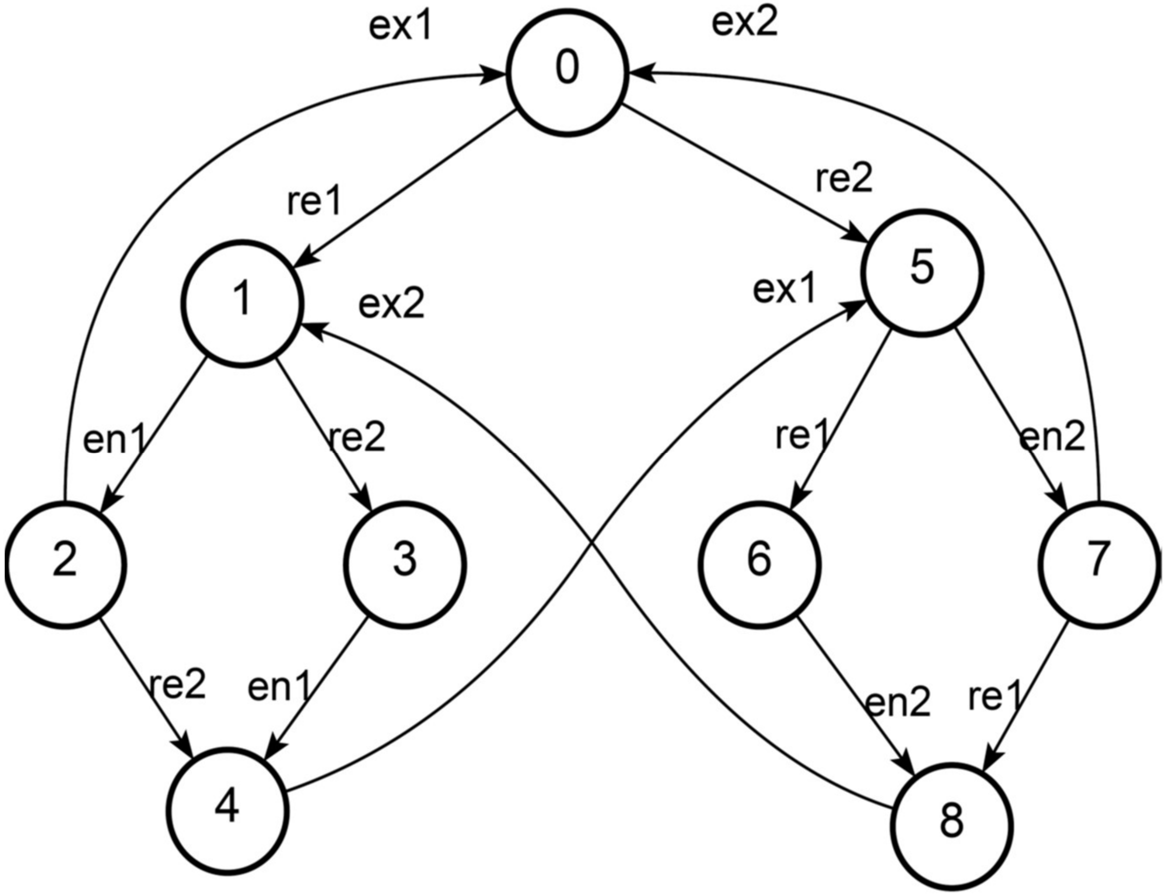
Esempio: mutua esclusione

re = request
en = enter
ex = exit



- Proprietà interessanti (safety):
  no contemporanea presenza dei due utenti in sezione critica

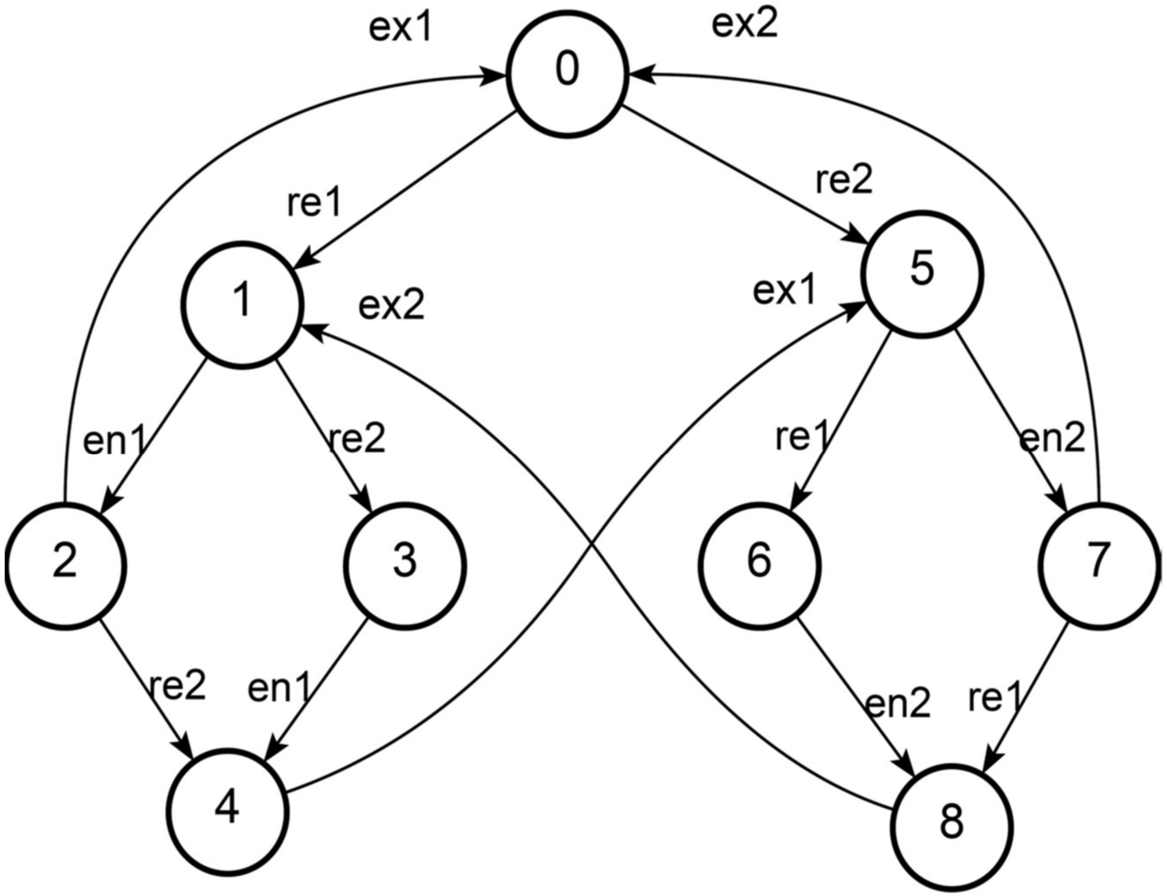
Esempio: mutua esclusione



- Proprietà interessanti (safety):
  [](en1 => (-en2 U ex1)) /\ [](en2 => (-en1 U ex2))

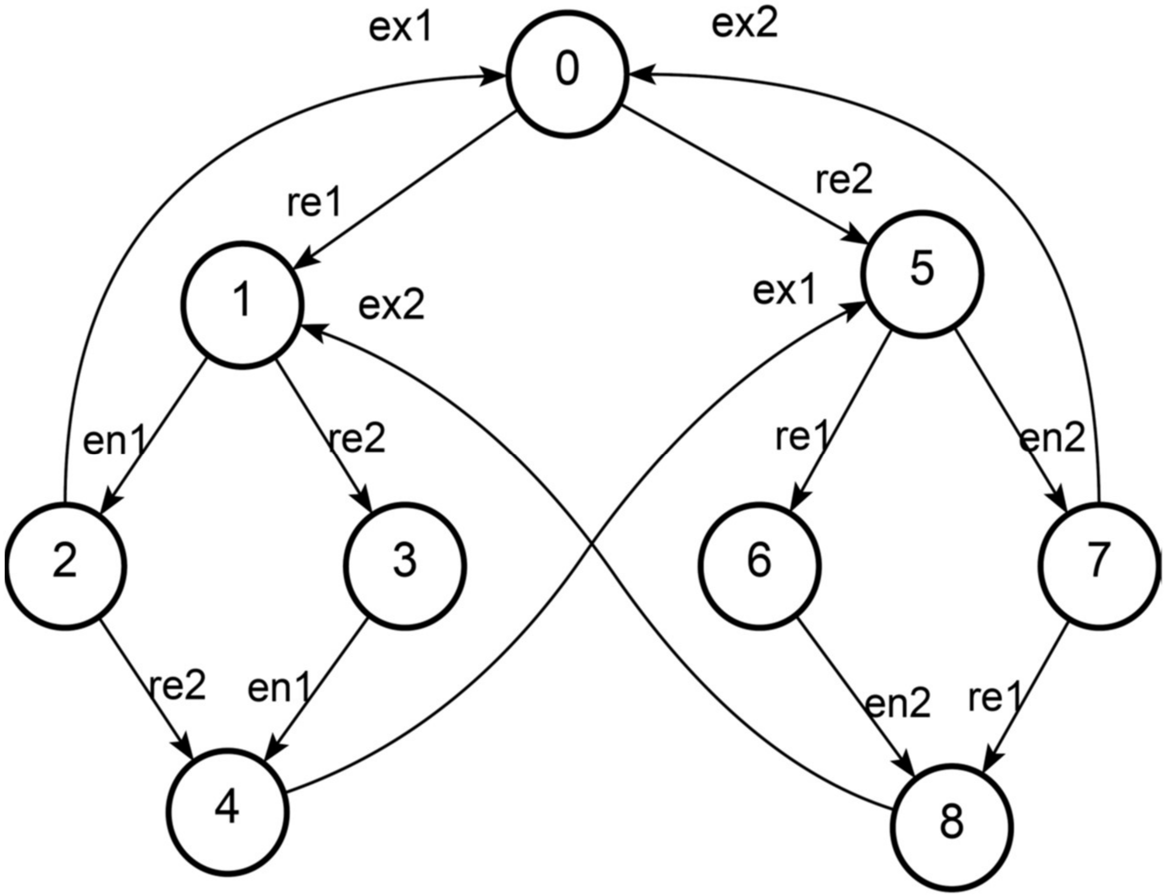
  always se en1 en2 è falsa finché ex1 e viceversa

Esempio: mutua esclusione



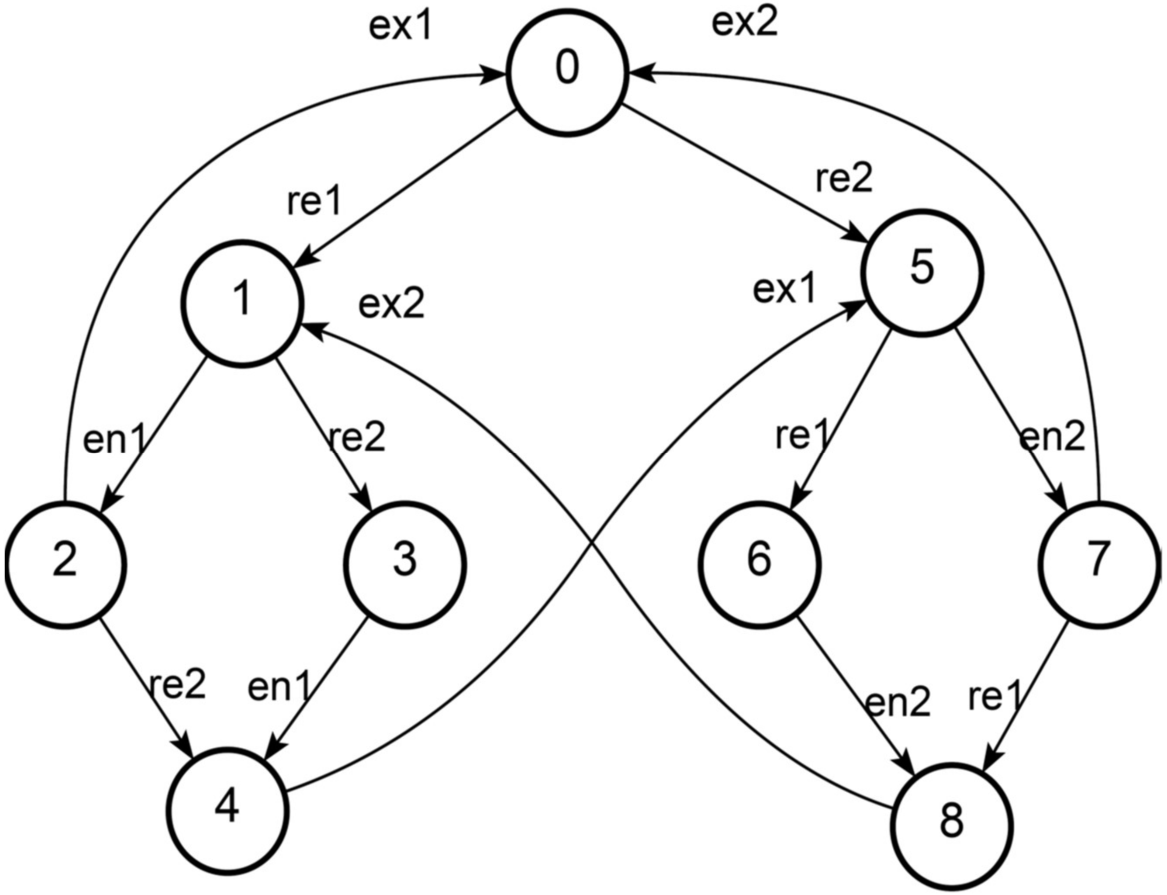
- Proprietà interessanti (safety):
  [](en1 => (-en2 U ex1)) /\ [](en2 => (-en1 U ex2))
  TRUE

Esempio: mutua esclusione



- Proprietà interessanti (liveness):
  ogni volta che un utente richiede di entrare, successivamente entrerà

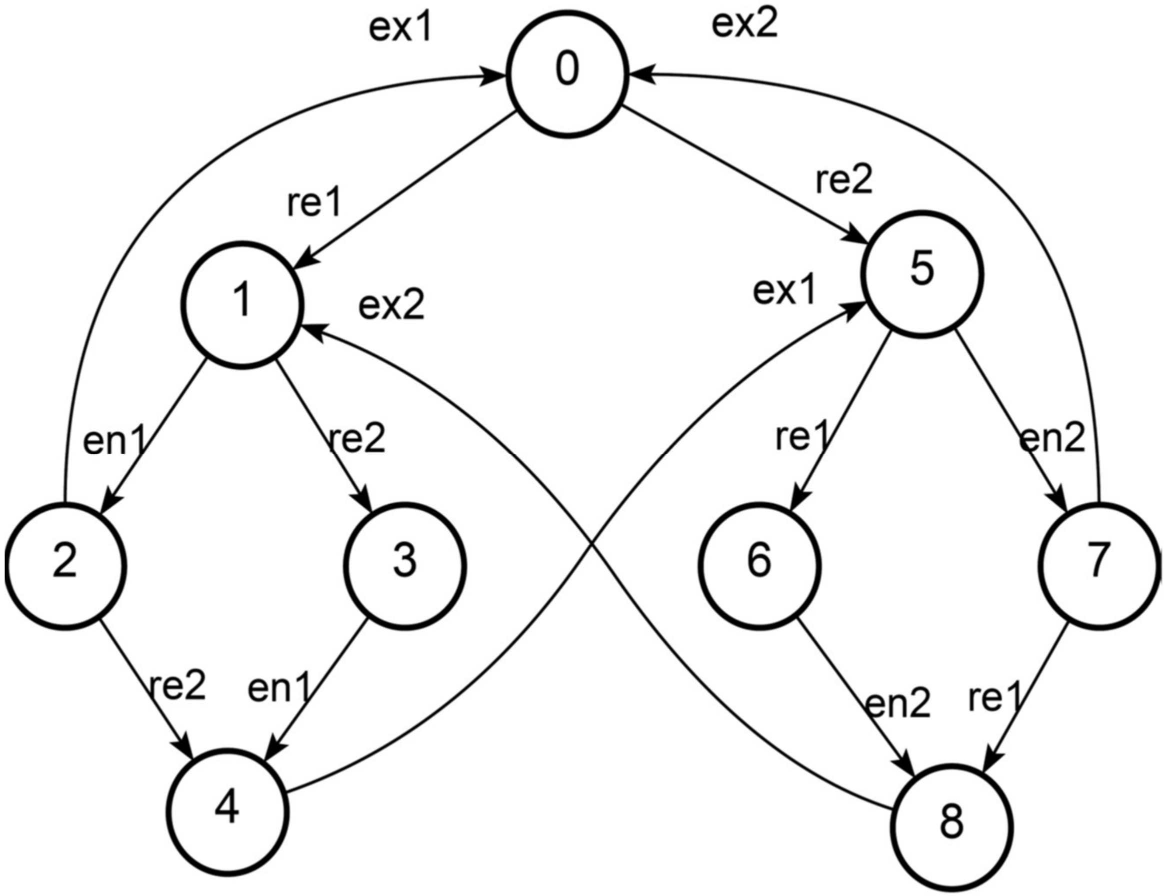
Esempio: mutua esclusione



- Proprietà interessanti (liveness):
  [](re1=>(<>en1)) /\ [](re2=>(<>en2))

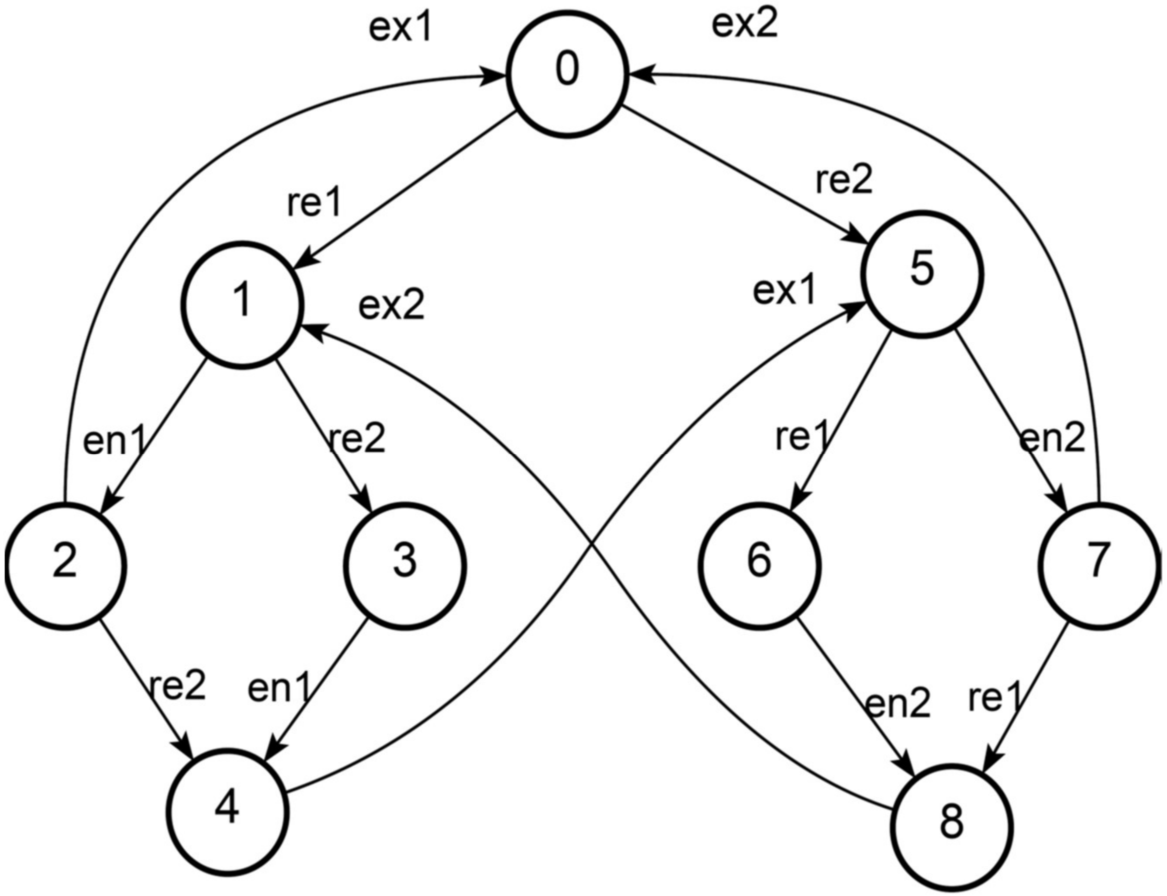
se ho re1 prima o poi (sometimes) faccio en1

Esempio: mutua esclusione



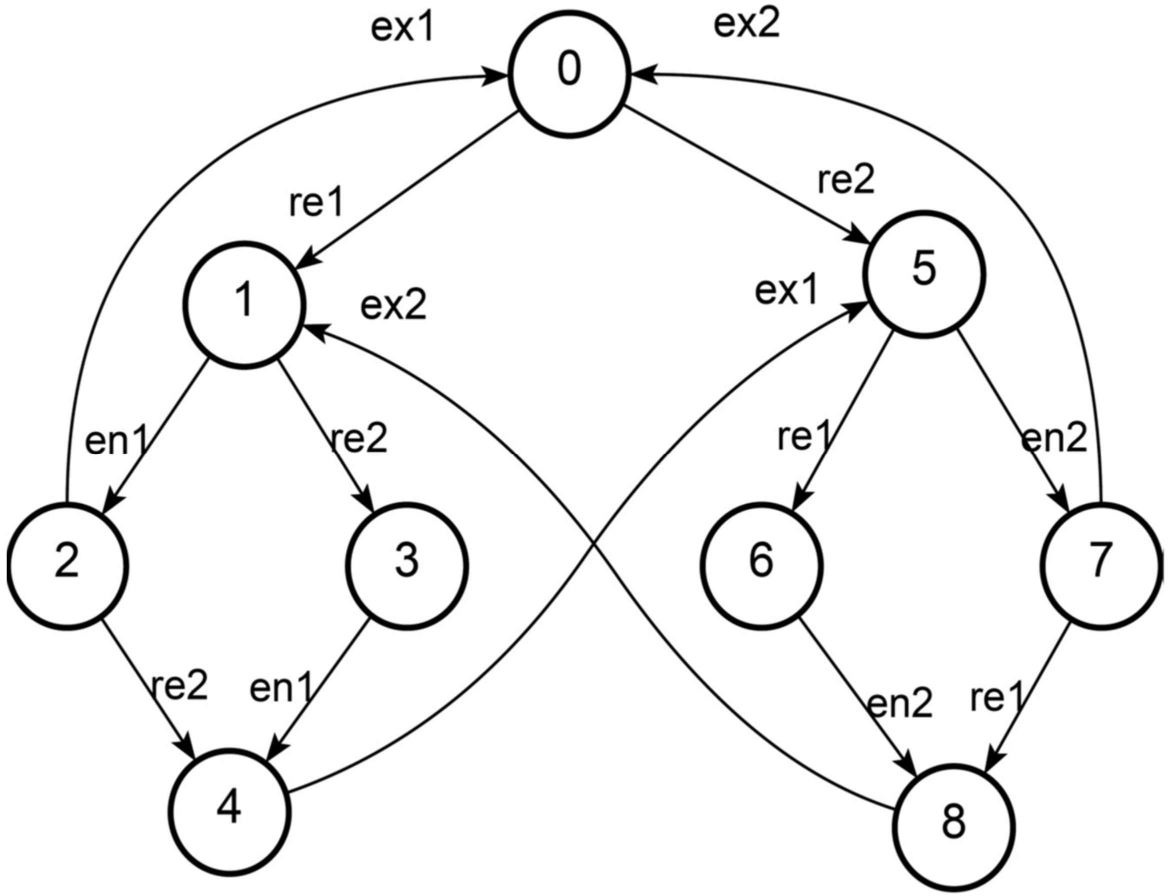
- Proprietà interessanti (liveness):
  [](re1=>(<>en1)) /\ [](re2=>(<>en2))
  TRUE

Esempio: mutua esclusione



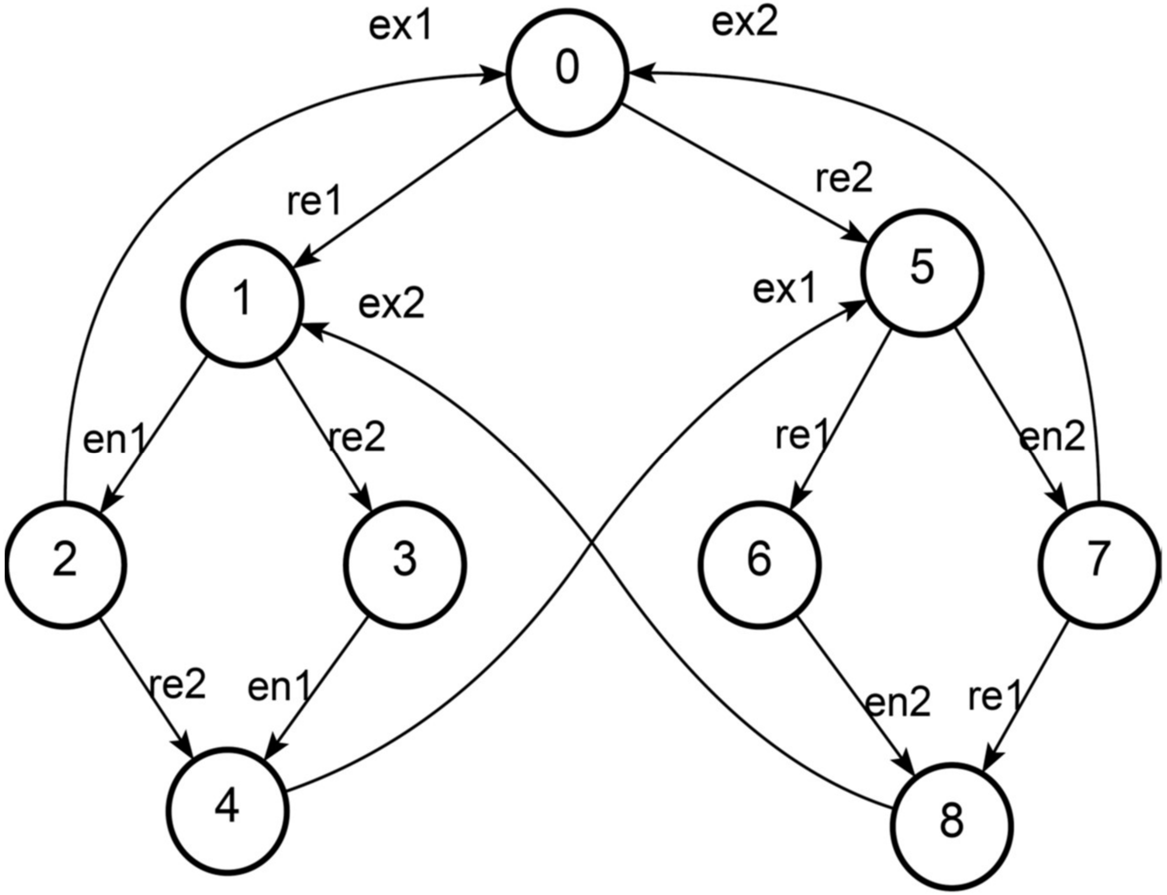
- Proprietà interessanti (fairness):
  si noti che si è sempre pronti ad accettare la request di un utente (a meno che non la si abbia già accettata)…

Esempio: mutua esclusione



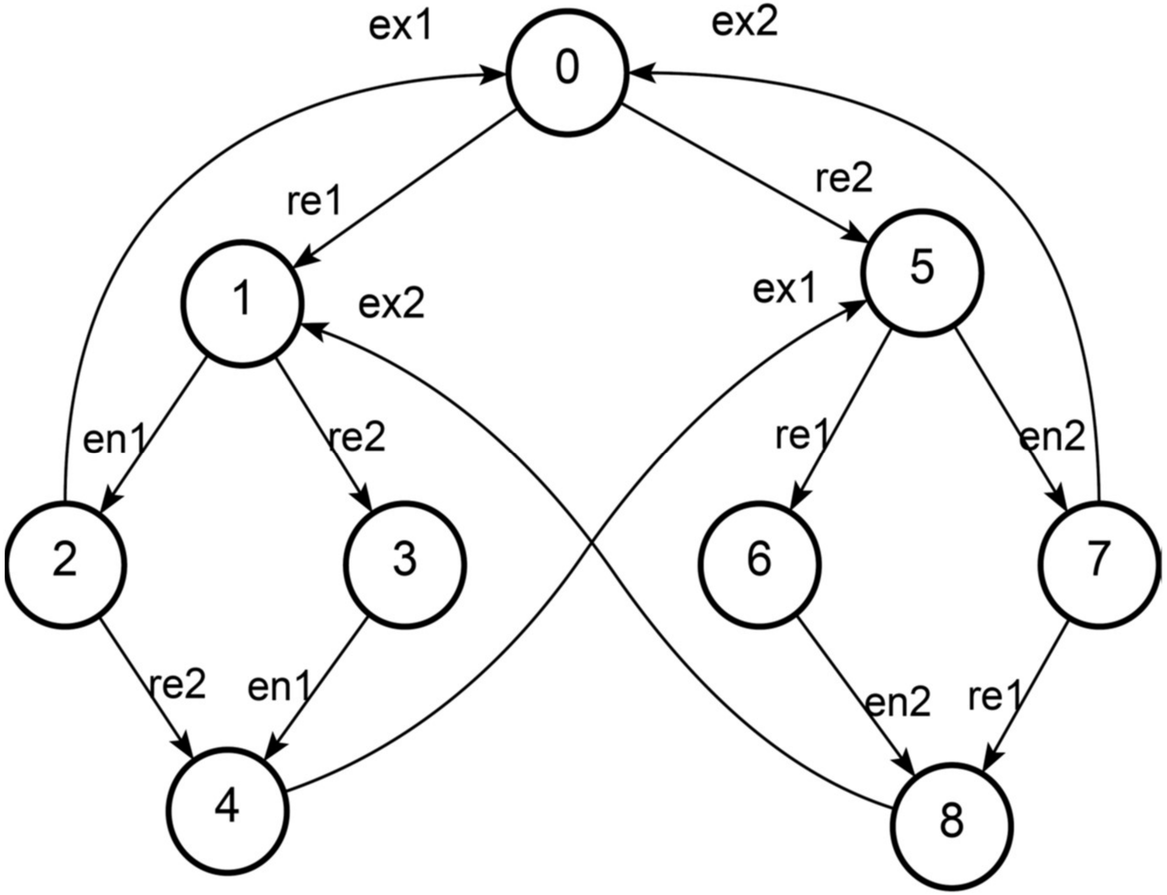
- Proprietà interessanti (fairness):
  ...ma siamo sicuri che la request verrà effettivamente accettata?

Esempio: mutua esclusione



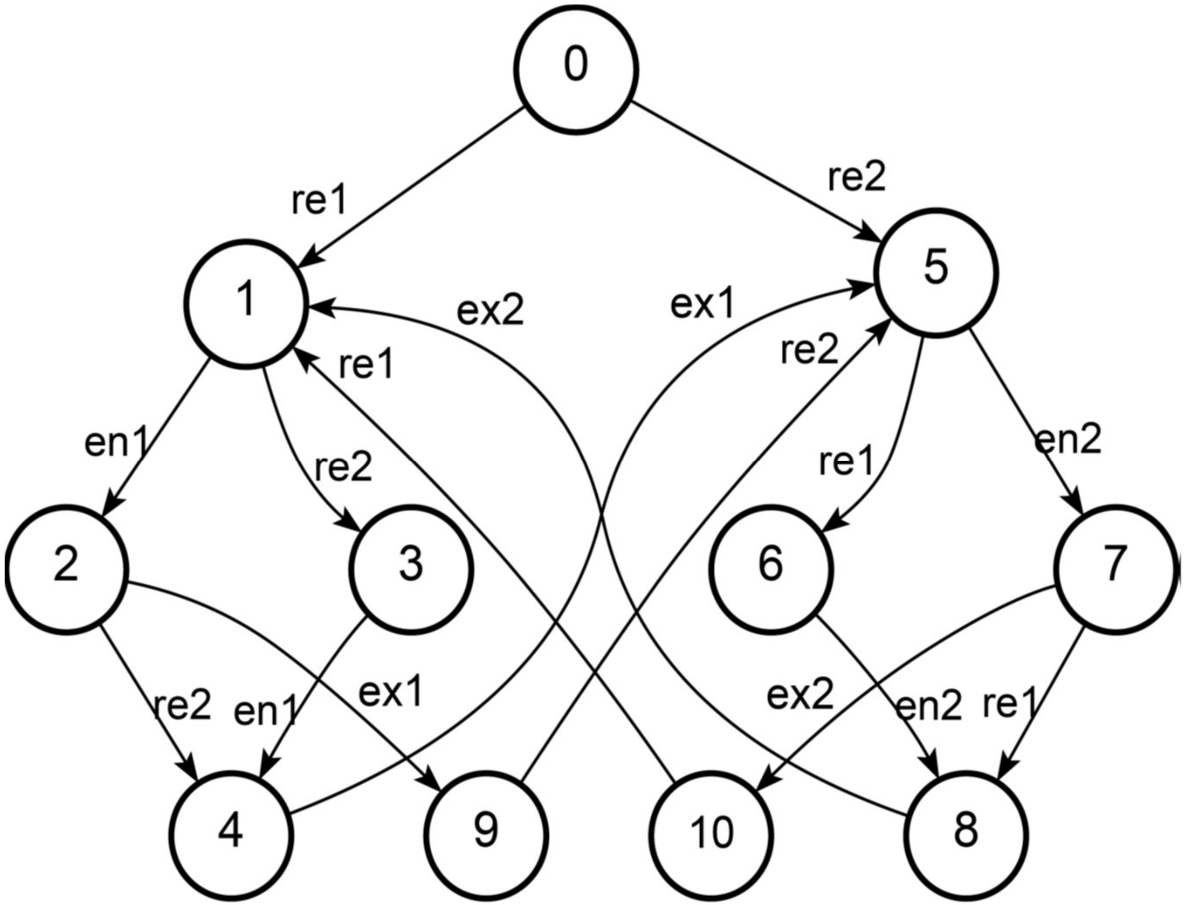
- Proprietà interessanti (fairness):
  <>re1 /\ <>re2

Esempio: mutua esclusione



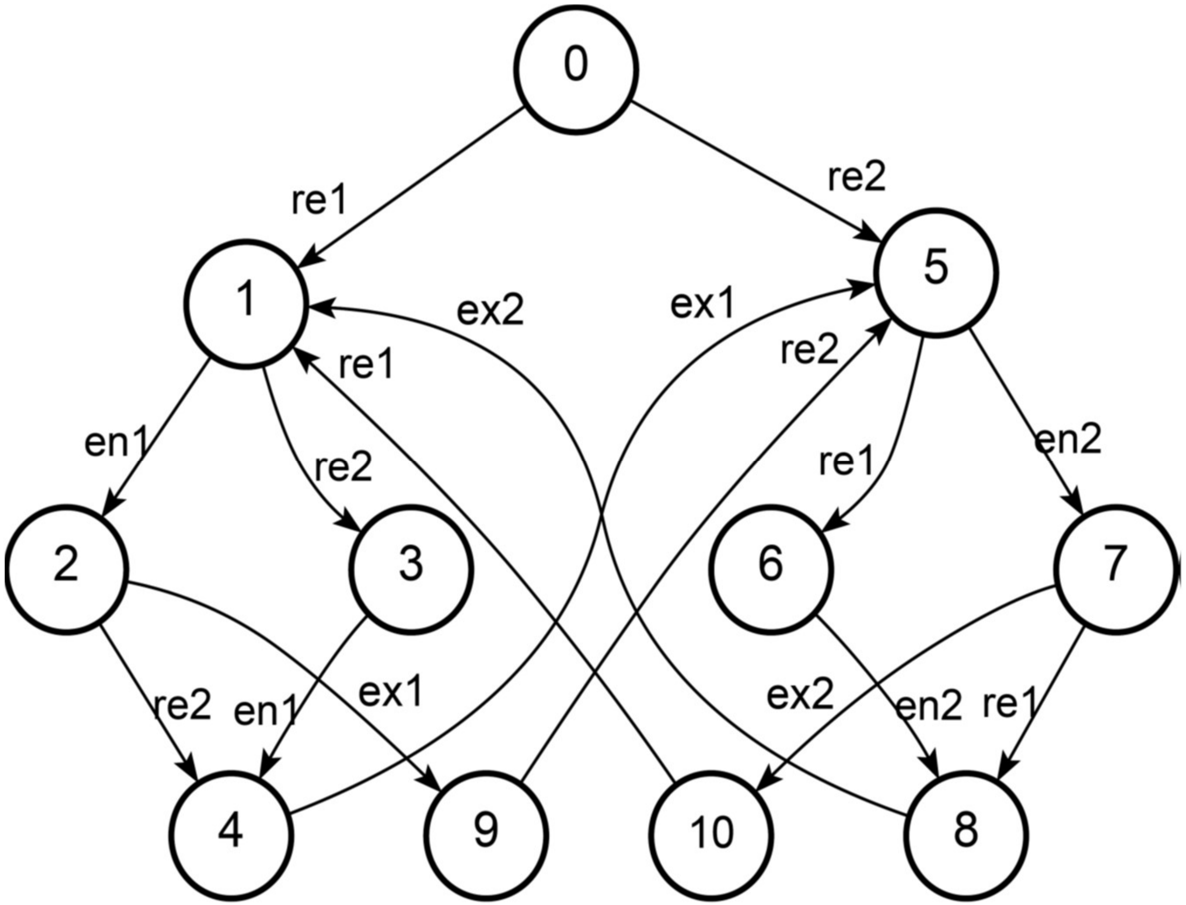
- Proprietà interessanti (fairness):
  <>re1 /\ <>re2 FALSE
  (con controesempio re2-en2-ex2-re2-en2-ex2-…)

Esempio: mutua esclusione (2)



- Nuova versione dell’automa:
  forzando l’alternanza fra gli utenti, sarà sempre vero che entrambi potranno eseguire prima o poi la loro request

Esempio: mutua esclusione (2)



- Nuova versione dell'automa:
  [](<>re1 /\ <>re2)   TRUE

Sistemi concorrenti

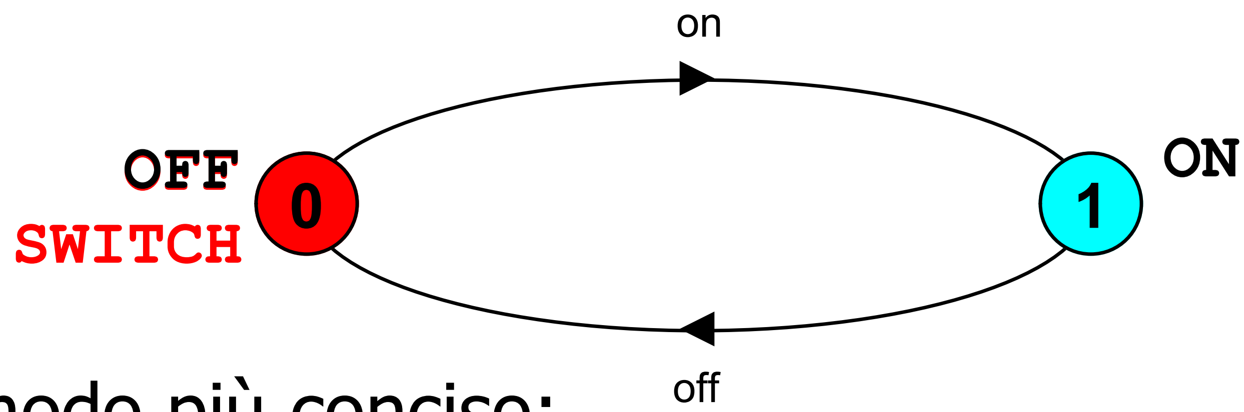
- I sistemi modellati da tali automi (LTS) sono solitamente “sistemi concorrenti”:
  - Vari processi **interagiscono** (es: i due utenti che interagiscono per accedere alla sezione critica)
- Le “process algebra” sono un modo naturale per rappresentare sistemi di questo tipo
  - Descrivere **ogni processo** da solo
  - **Combinare** le descrizioni di tali processi per ottenere l’intero sistema

Process Algebra

- Esistono tanti tipi di process algebra (CCS, ACP, CSP, pi-calculus, …)
- Trattiamo FSP (Finite State Processes) usato dal tool LTSA (Labeled Transition System Analyzer)
- Un processo è rappresentato come un LTS:
  - si danno **nomi agli stati**
  - per ogni stato si descrivono le **transizioni** che lo stato **può fare**

Esempio: interruttore

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

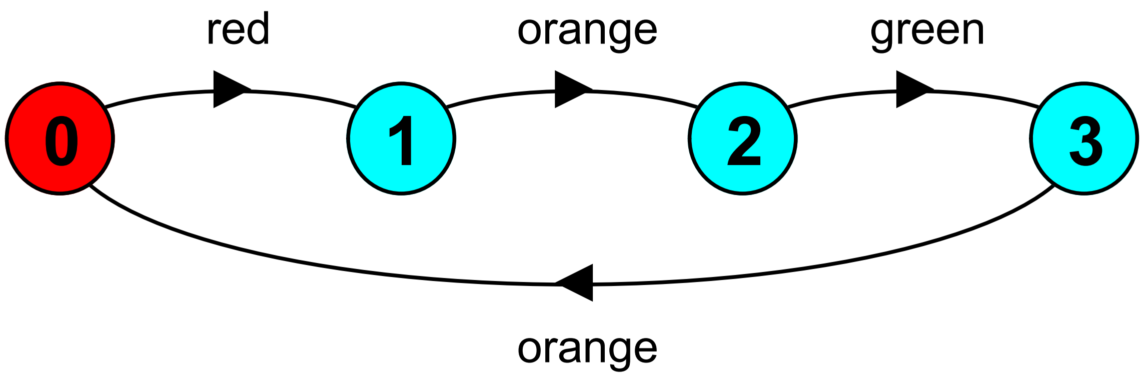


- Oppure, in modo più conciso:

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

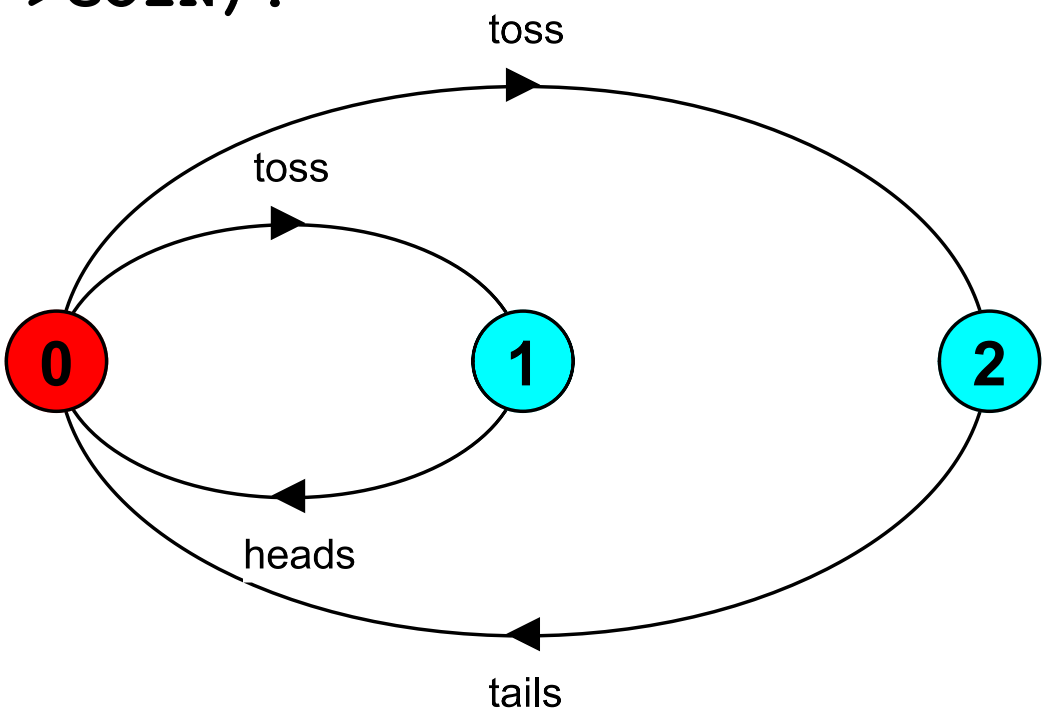
Esempio: semaforo

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



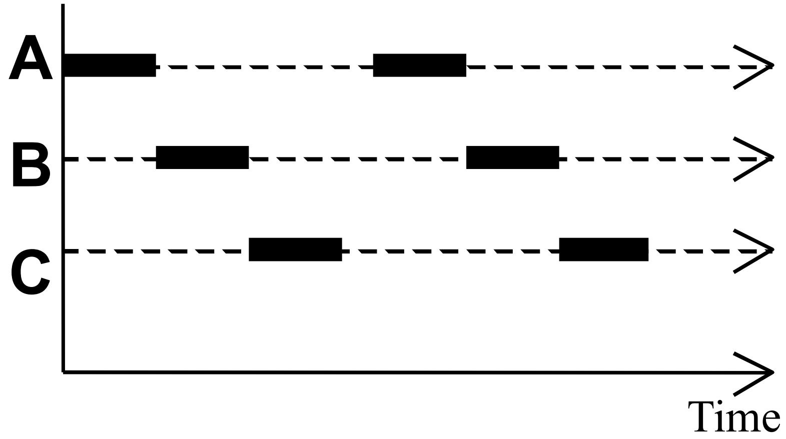
Esempio: lancio della moneta

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



Definizioni

- Concorrenza (interleaving):
  - Esecuzione “logicamente” parallela di processi (LTS), come in un sistema multitasking



- Sincronizzazione:
  - Esecuzione “fisicamente” contemporanea di azioni

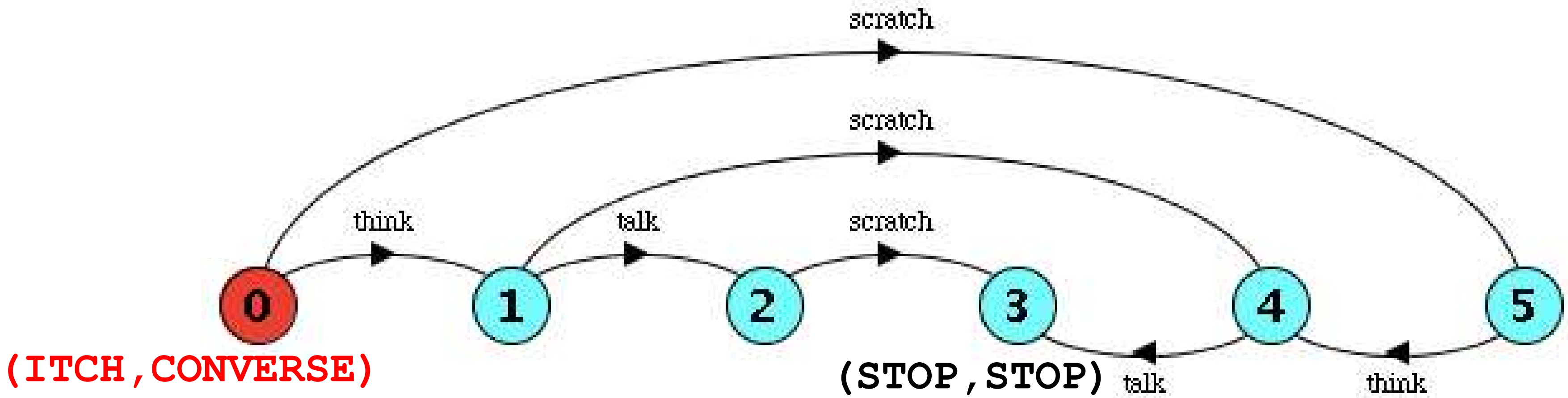
Parallelismo: interleaving e sincronizzazione

- Interleaving:
  - Transizioni **specifiche** di un processo (cioè assenti in altri processi) vengono eseguite in "interleaving"
  - **Interleaving**: una azione alla volta in **ordine arbitrario** (nessuna assunzione sulla velocità relativa dei processi)
- Sincronizzazione:
  - Su **etichette comuni** (nell'alfabeto degli LTS eseguiti in parallelo) è prevista "sincronizzazione"
  - **Sincronizzazione**: esecuzione **simultanea** di una azione da parte di più LTS

Esempio: composizione parallela tramite interleaving

```
ITCH  = (scratch->STOP).
CONVERSE = (think->talk->STOP).

||CONVERSE_ITCH = (ITCH || CONVERSE).
```

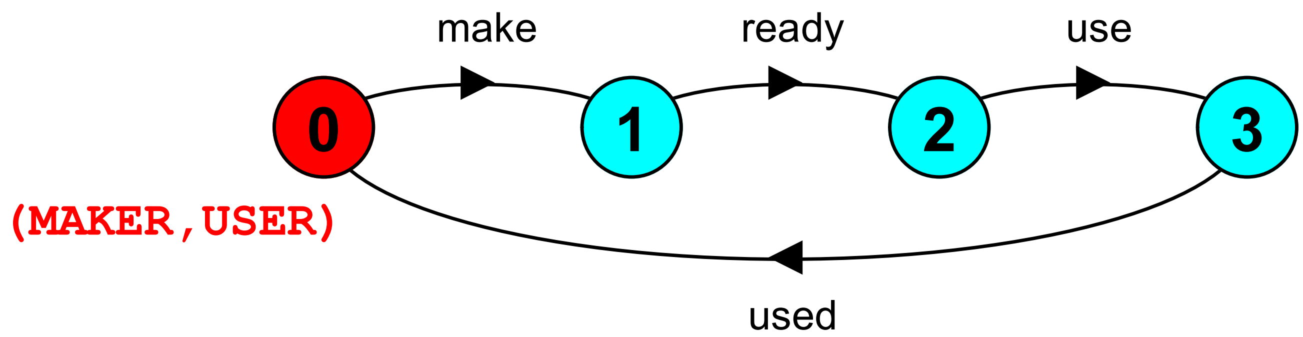
- Avendo **alfabeti disgiunti**, le azioni vengono eseguite in interleaving (STOP è stato di blocco):



Esempio: composizione parallela con sincronizzazione

```
MAKER = (make->ready->used->MAKER).
USER  = (ready->use->used->USER).

||MAKER_USER = (MAKER || USER).
```

- "ready" e "used" sono **in entrambi gli alfabeti**, quindi tali transizioni devono **sincronizzarsi**:



Formalmente

Dati due LTS $A_i = (Q_i, \Sigma_i, \delta_i, q_0^i)$, la loro **composizione parallela** $A_1 || A_2$ è l'LTS $(Q_1 \times Q_2, \Sigma_1 \cup \Sigma_2, \delta, (q_0^1, q_0^2))$, dove δ è definita:

- per ogni $a \in \Sigma_1 \cap \Sigma_2$ (caso della **sincronizzazione**)

  $$\delta((p, q), a) = \delta_1(p, a) \times \delta_2(q, a)$$

  cioè se $p \xrightarrow{a} p'$ e $q \xrightarrow{a} q'$ allora $(p, q) \xrightarrow{a} (p', q')$

- per ogni $a \notin \Sigma_1 \cap \Sigma_2$ (caso dell'**interleaving**)

  $$\delta((p, q), a) = \delta_1(p, a) \times \{q\} \quad \text{se } a \in \Sigma_1$$

  cioè se $p \xrightarrow{a} p'$ allora $(p, q) \xrightarrow{a} (p', q)$

  $$\delta((p, q), a) = \{p\} \times \delta_2(q, a) \quad \text{se } a \in \Sigma_2$$

  cioè se $q \xrightarrow{a} q'$ allora $(p, q) \xrightarrow{a} (p, q')$

Proprietà della composizione parallela

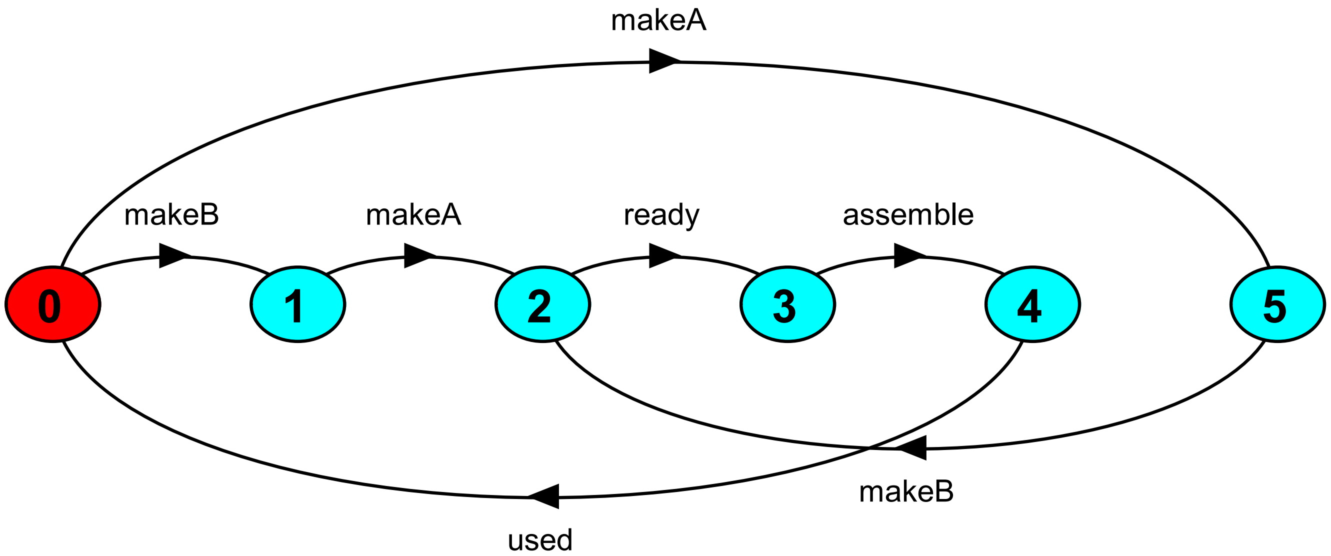
- La composizione parallela è **commutativa** e **associativa**
  - L'LTS di $(A_1||A_2)$ è lo stesso di $(A_2||A_1)$
  - L'LTS di $(A_1||A_2)||A_3$ è lo stesso di $A_1||(A_2||A_3)$

- Possiamo quindi denotare la **composizione parallela di multipli LTS** $A_1, A_2, \ldots, A_n$ scrivendo semplicemente $A_1||A_2||\ldots||A_n$

Esempio: composizione parallela multipla

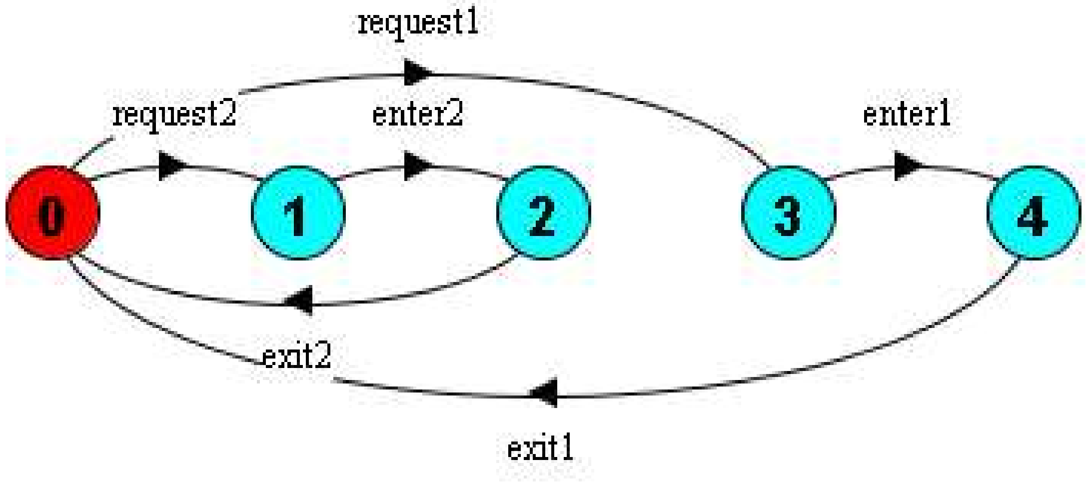
```
MAKE_A   = (makeA->ready->used->MAKE_A).
MAKE_B   = (makeB->ready->used->MAKE_B).
ASSEMBLE = (ready->assemble->used->ASSEMBLE).

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

- "ready" e "used" **sono in tutti e 3 gli alfabeti**, quindi i tre processi devono sincronizzarsi:



Esempio: mutua esclusione

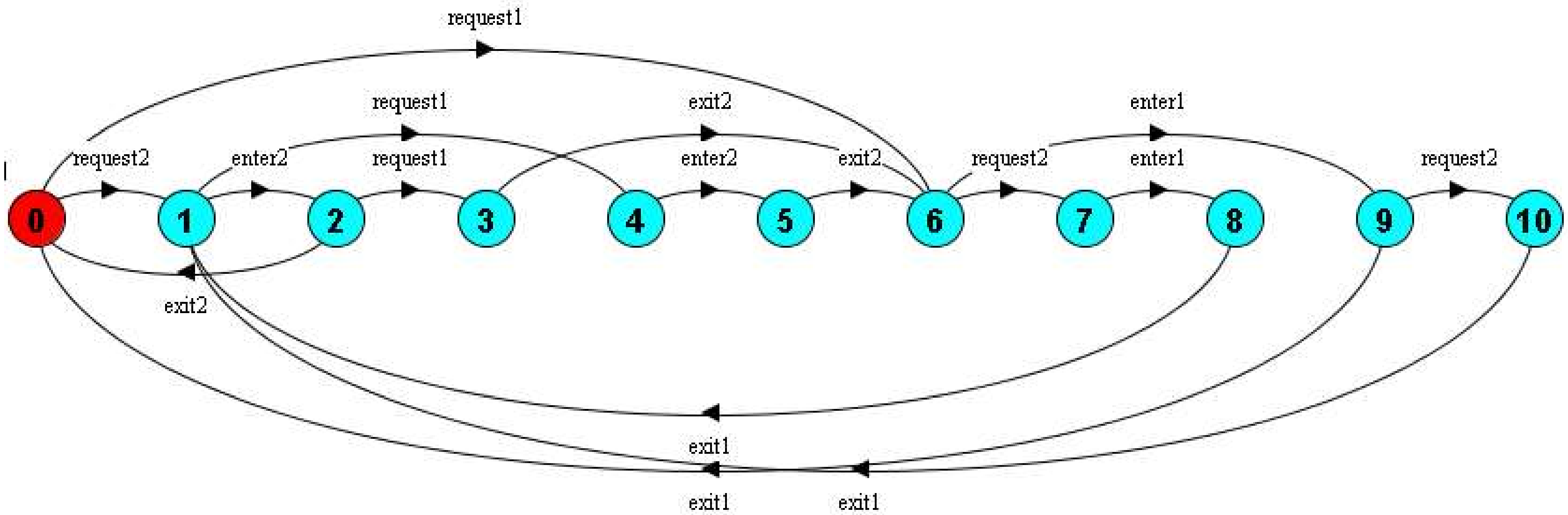
```
USER1 = (request1 -> enter1 -> exit1 -> USER1 ).
USER2 = (request2 -> enter2 -> exit2 -> USER2 ).

CONTROLLER = (request1->enter1->exit1->CONTROLLER |
              request2->enter2->exit2->CONTROLLER).

||MUTUAL_EXCLUSION = ( USER1 || USER2 || CONTROLLER ).
```



Esempio: mutua esclusione (2)

```
USER1 = (request1 -> enter1 -> exit1 -> USER1).
USER2 = (request2 -> enter2 -> exit2 -> USER2).

CONTROLLER = ( request1 -> TURN1
             | request2 -> TURN2 ),

TURN1 = ( request2 -> enter1 -> exit1 -> TURN2
        | enter1 -> ( request2 -> exit1 -> TURN2
                    | exit1 -> CONTROLLER
                    )
        ),

TURN2 = ( request1 -> enter2 -> exit2 -> TURN1
        | enter2 -> ( request1 -> exit2 -> TURN1
                    | exit2 -> CONTROLLER
                    )
        ).

||MUTUAL_EXCLUSION = ( USER1 || USER2 || CONTROLLER ).
```

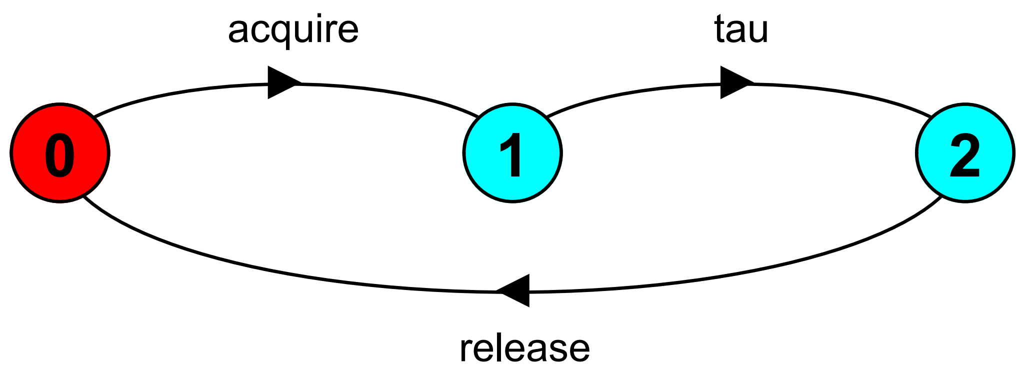
Esempio: mutua esclusione (2)



Hiding

- Con "P@{a1,..,ax}" si **limita l'alfabeto** di P alle azioni {a1,..,ax}
  - **Altre azioni** vengono trasformate in azione locale "**tau**"
  - In questo modo si **evita** che tali azioni vengano **sincronizzate** con altri processi

```
USER = (acquire->use->release->USER)
          @{acquire,release}.
```



Formalmente

Dato un LTS $A = (Q, \Sigma, \delta, q_0)$, l'LTS ottenuto **limitando il suo alfabeto a** $\Sigma' \subseteq \Sigma$ è $A@\Sigma' = (Q, \Sigma', \delta', q_0)$, dove δ' è definita:

- per l'**azione** speciale **tau** (caso di azioni che **si nascondono**)

  $\delta'(p, tau) = \bigcup_{a \notin \Sigma'} \delta(p, a)$

  cioè se $p \xrightarrow{a} p' \wedge a \notin \Sigma'$ allora in $A@\Sigma'$ si ha $p \xrightarrow{tau} p'$

- per ogni $a \in \Sigma'$ (caso di azioni che **non si nascondono**)

  $\delta'(p, a) = \delta(p, a)$

  cioè se $p \xrightarrow{a} p' \wedge a \in \Sigma'$ allora in $A@\Sigma'$ si ha $p \xrightarrow{a} p'$

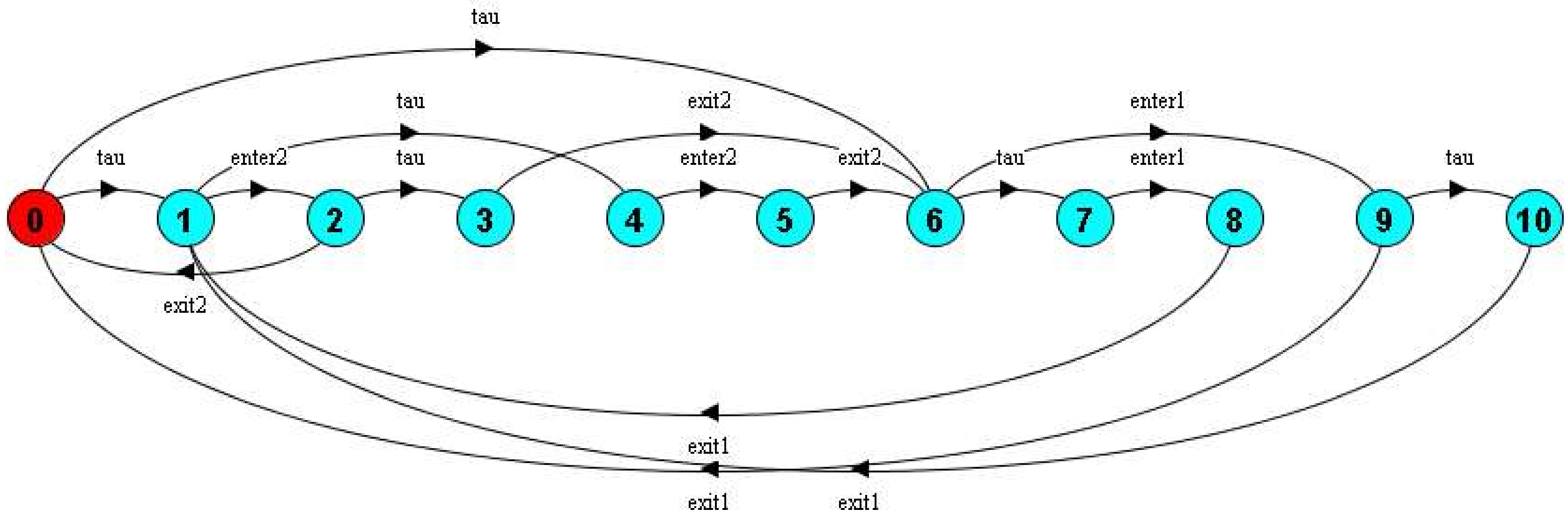
Esempio: mutua esclusione (3)

- Possiamo **nascondere le azioni di richiesta**

```
||MUTUAL_EXCLUSION =
      ( USER1 || USER2 || CONTROLLER )@{enter1,enter2,exit1,exit2}.
```

- In questo modo possiamo **comporre** l'LTS risultante **con un altro LTS che usa azioni di richiesta** senza che avvenga sincronizzazione

Esempio: mutua esclusione (3)



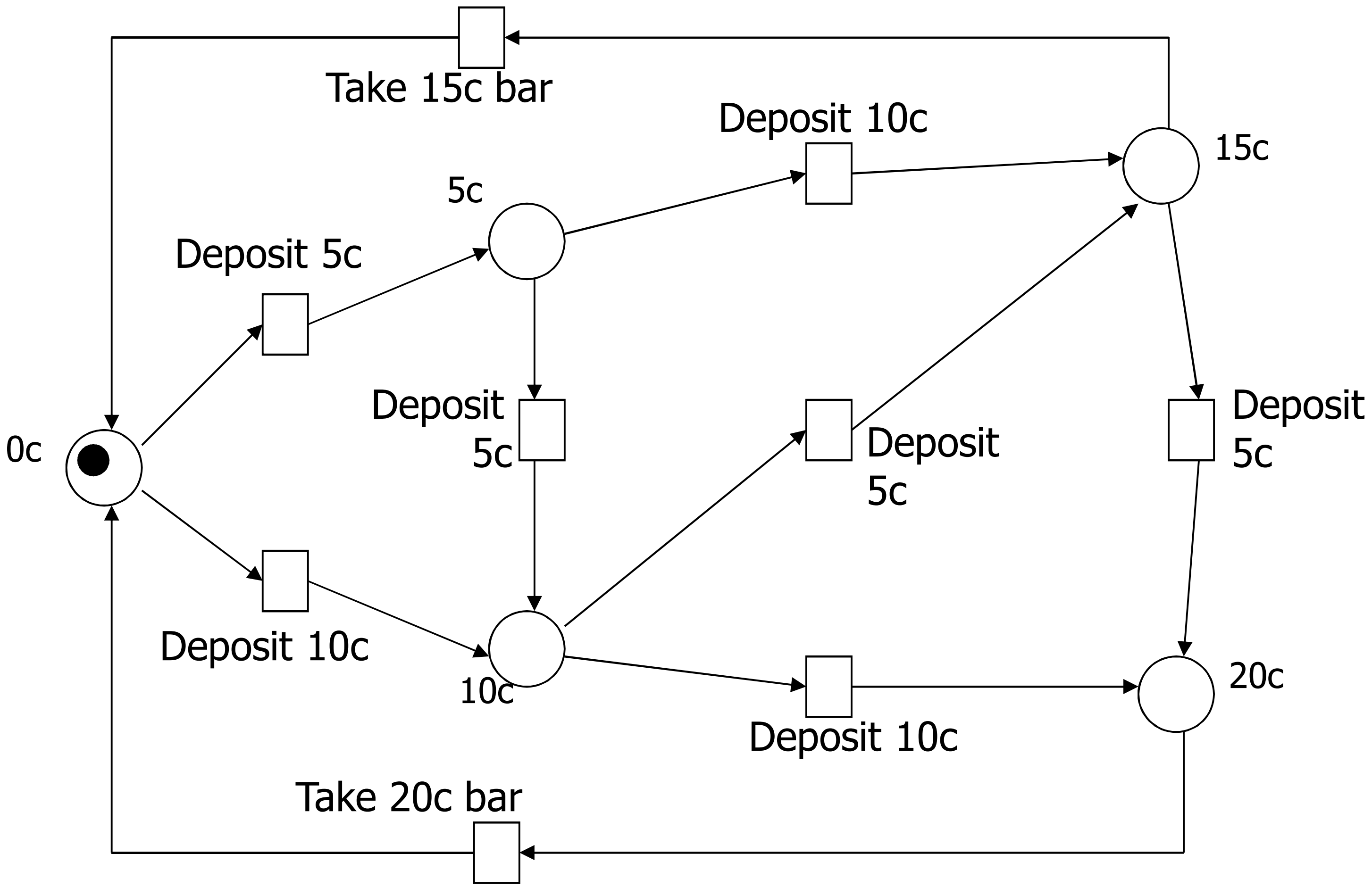
Reti di Petri (Petri nets)

- Approccio “**grafico**” alla rappresentazione dei **sistemi concorrenti**
  - Può essere considerato un’**estensione naturale degli automi** per sistemi concorrenti
  - Gli stati sono detti “**piazze**” (place)
  - Più piazze sono contemporaneamente **attive**
    - indicato dalla presenza di "**token**" al loro interno
  - Le **transizioni** modificano gli stati attivi:
    - **Consumano** un multi-insieme di stati attivi (token)
    - **Generano** un nuovo multi-insieme di stati attivi (token)

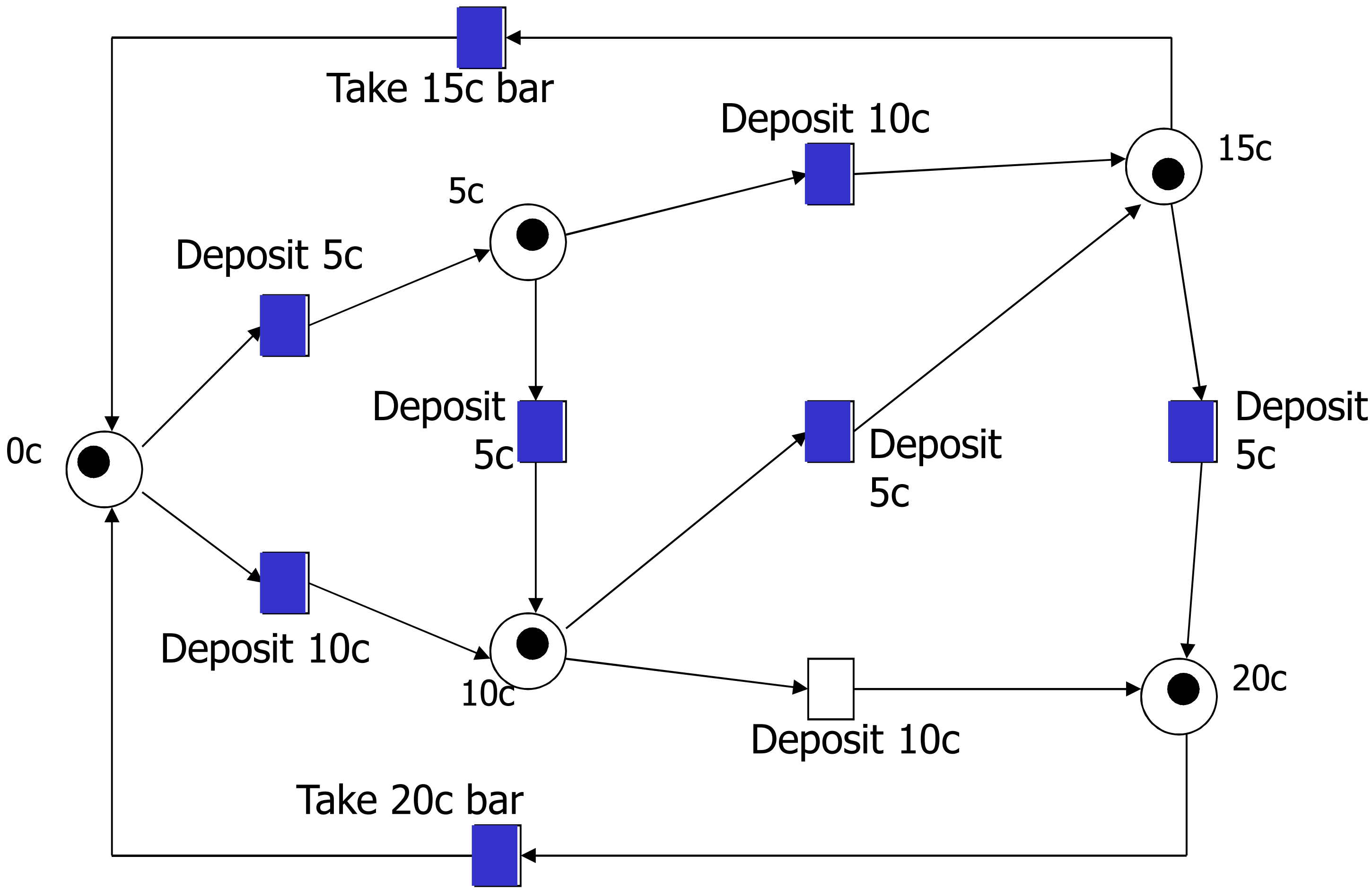
Esempio: distributore automatico

- Il distributore fornisce due tipi di snack, da 15 e 20 centesimi
- Si usano monete da 10 e 5 centesimi
- Il distributore non prevede resto

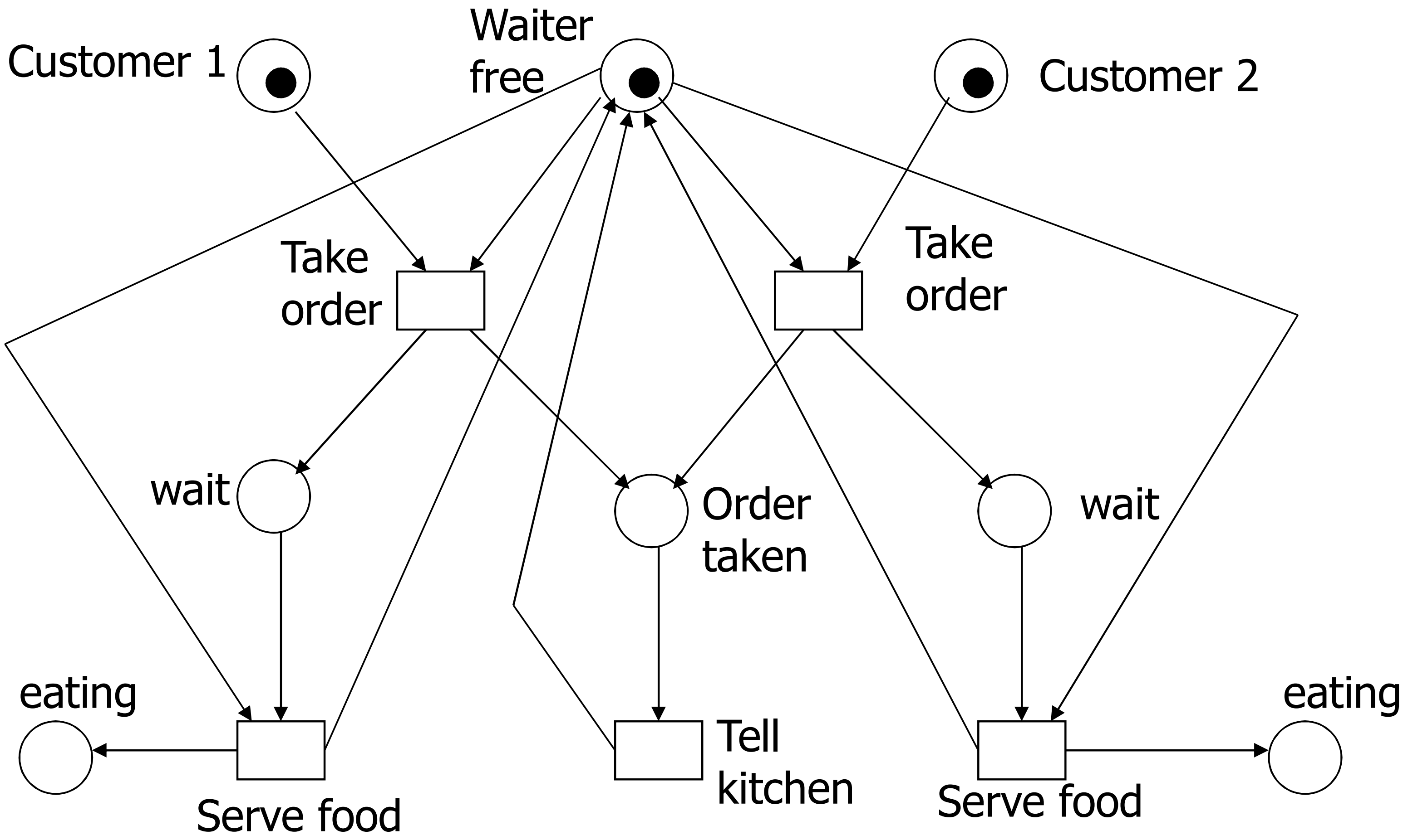
Esempio: distributore automatico



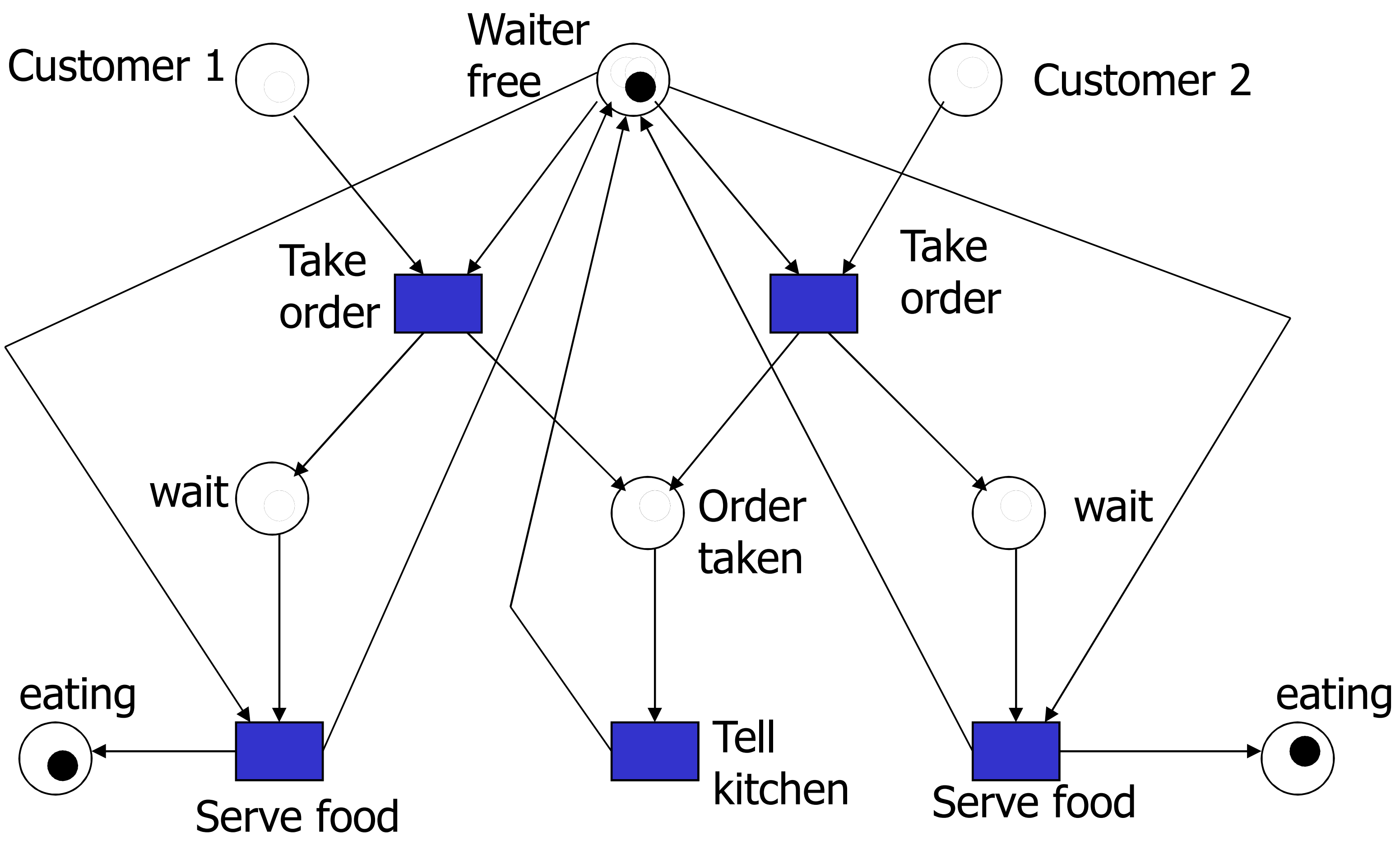
Esempio: distributore automatico



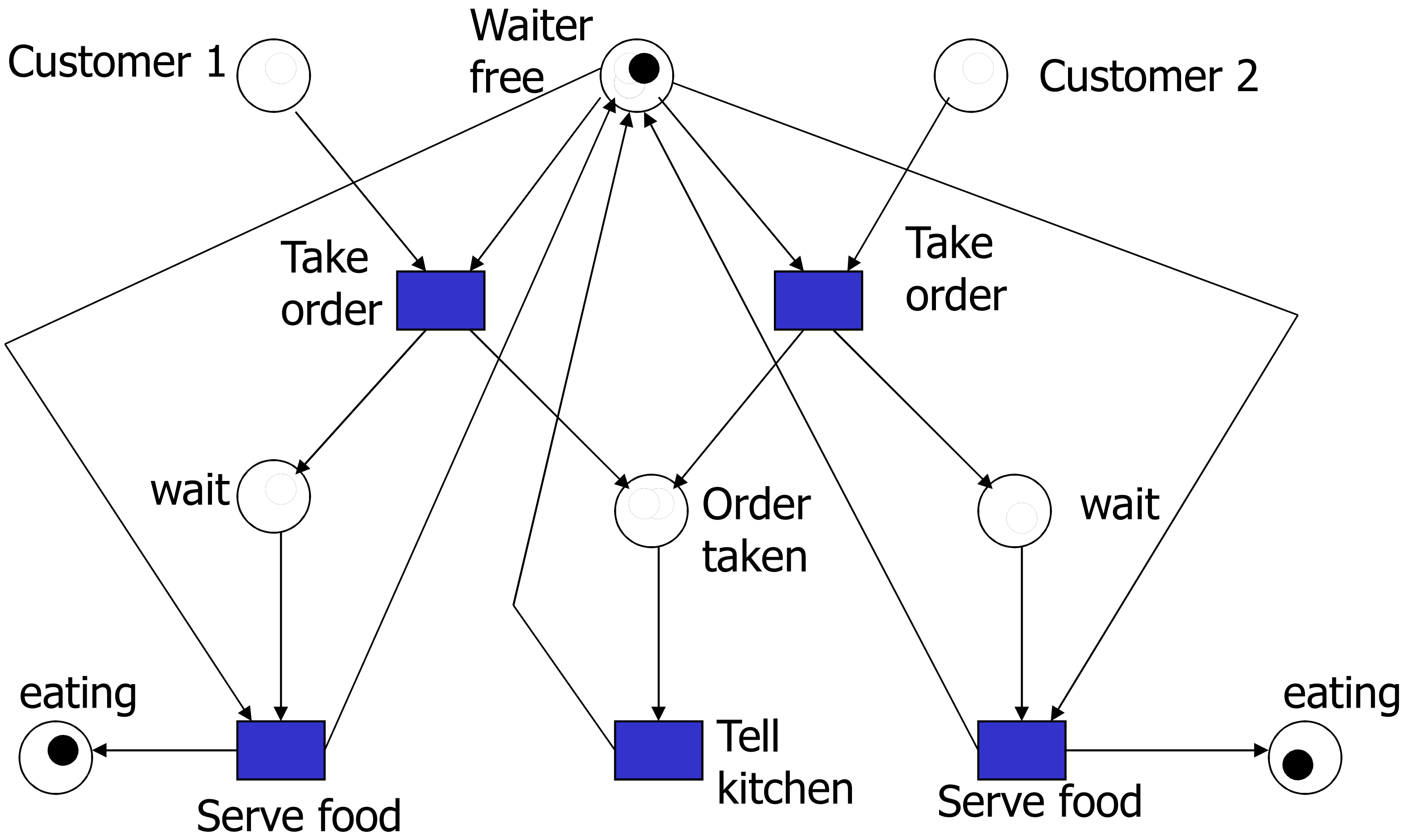
Esempio: ristorante



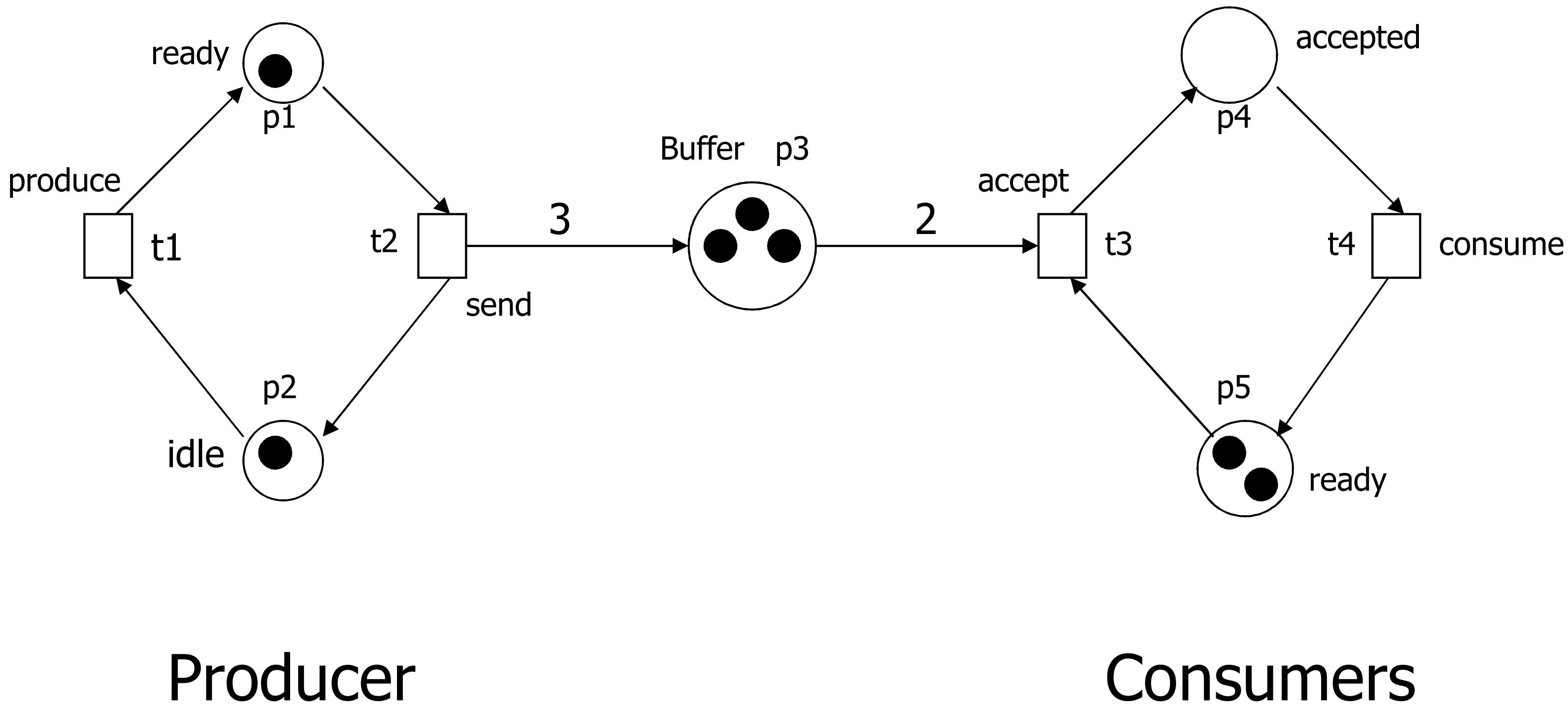
Esempio: ristorante (scenario 1)



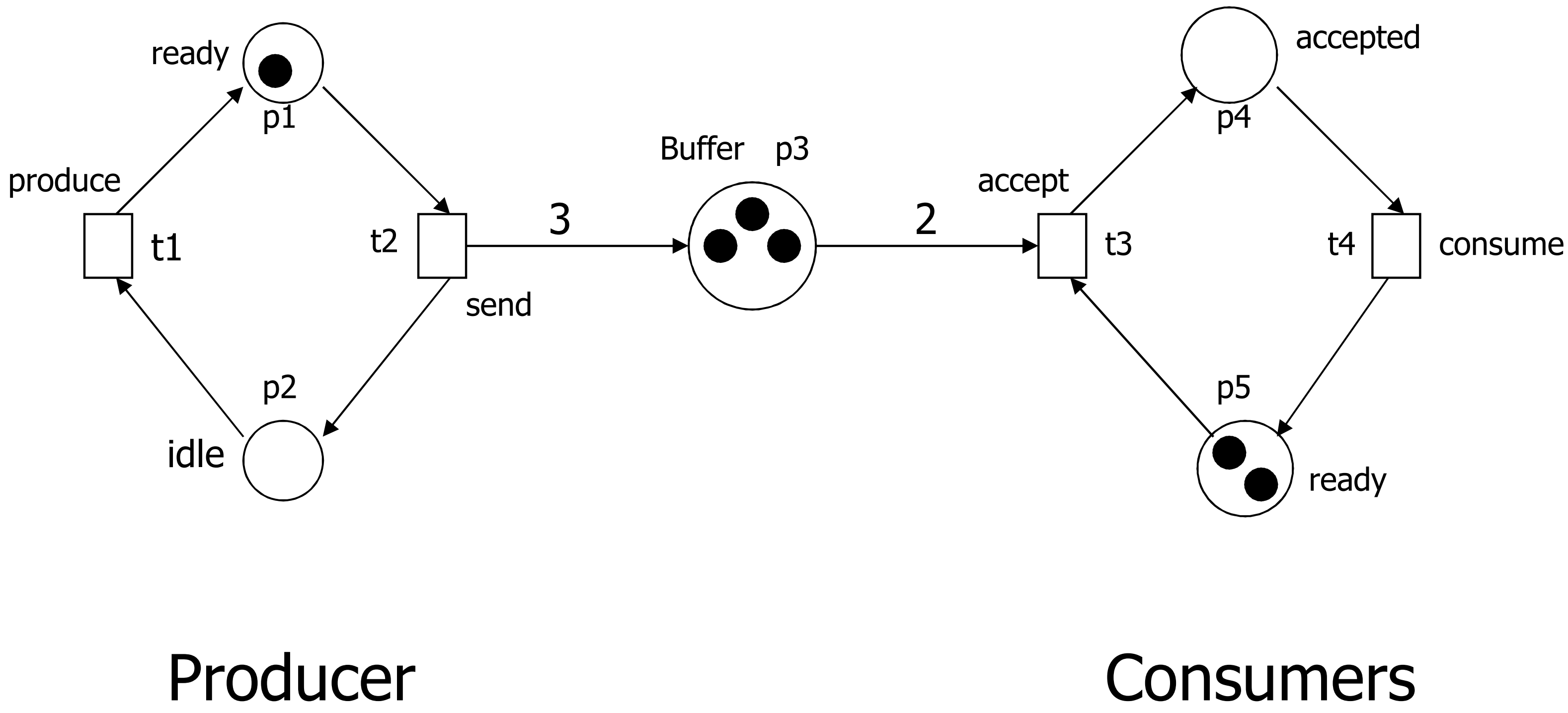
Esempio: ristorante (scenario 2)



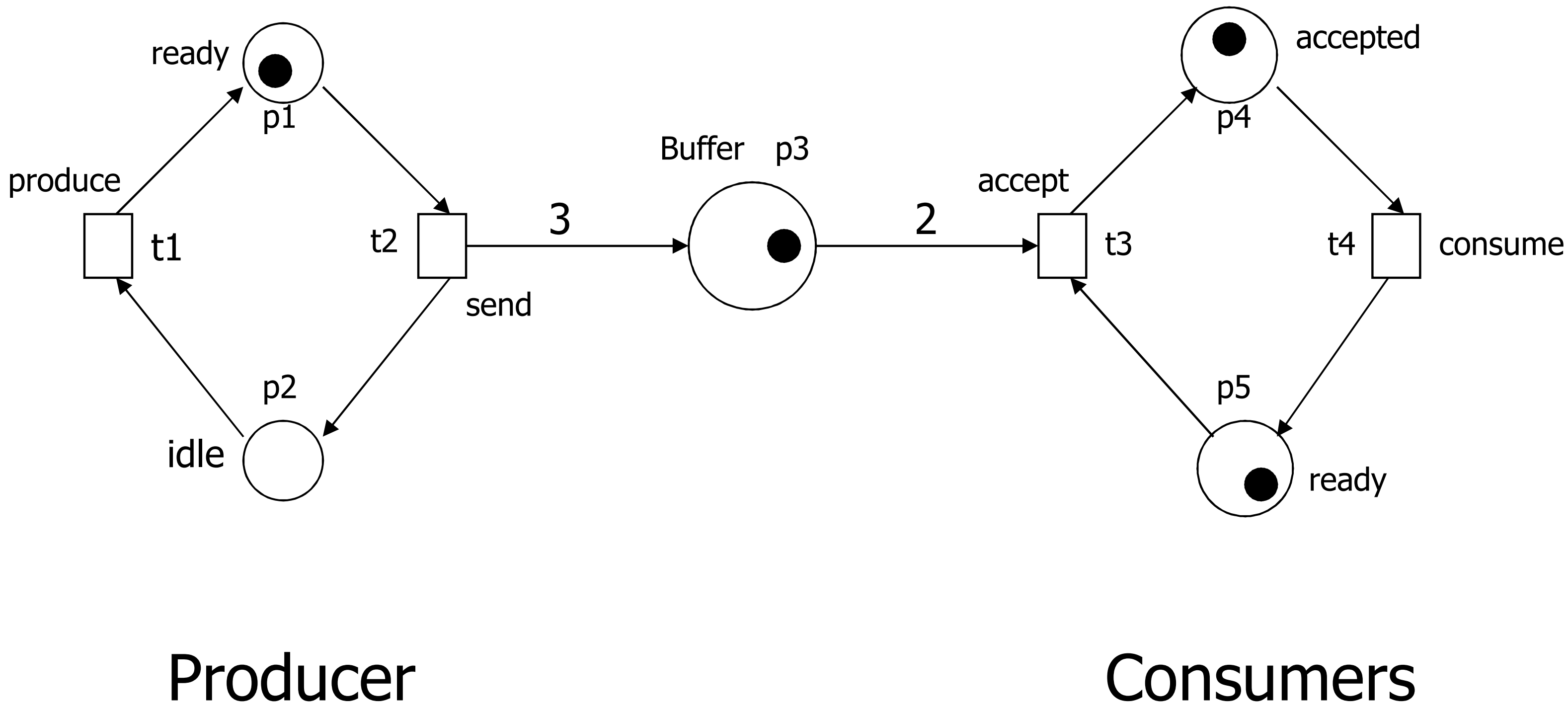
Esempio: produttore / consumatori



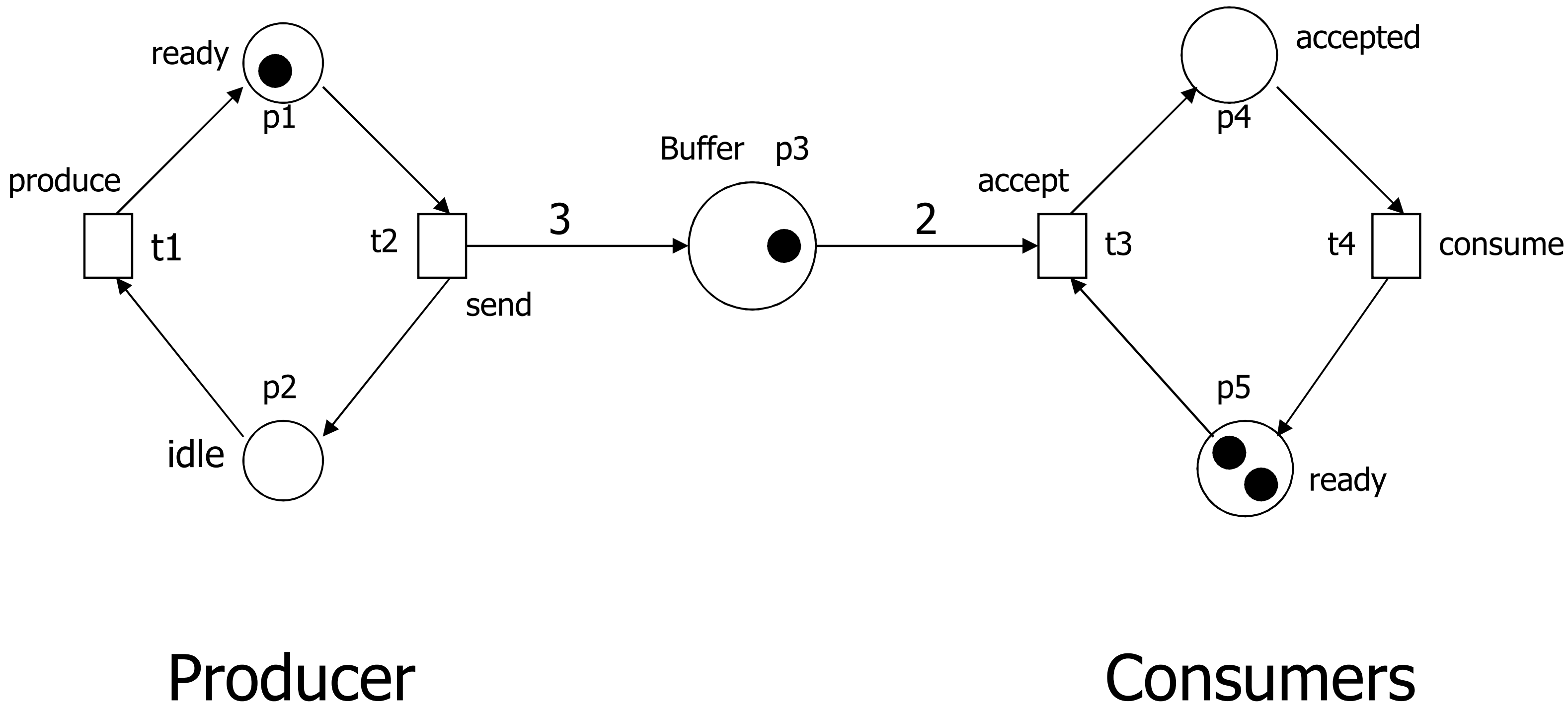
Esempio: produttore / consumatori



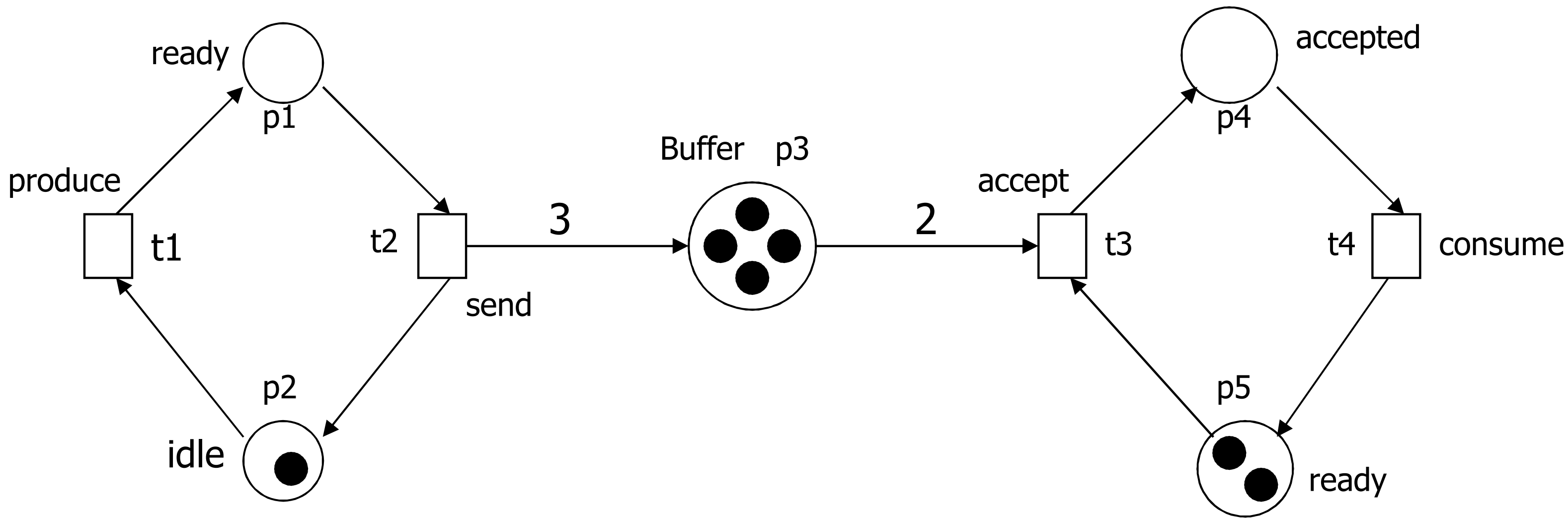
Esempio: produttore / consumatori



Esempio: produttore / consumatori



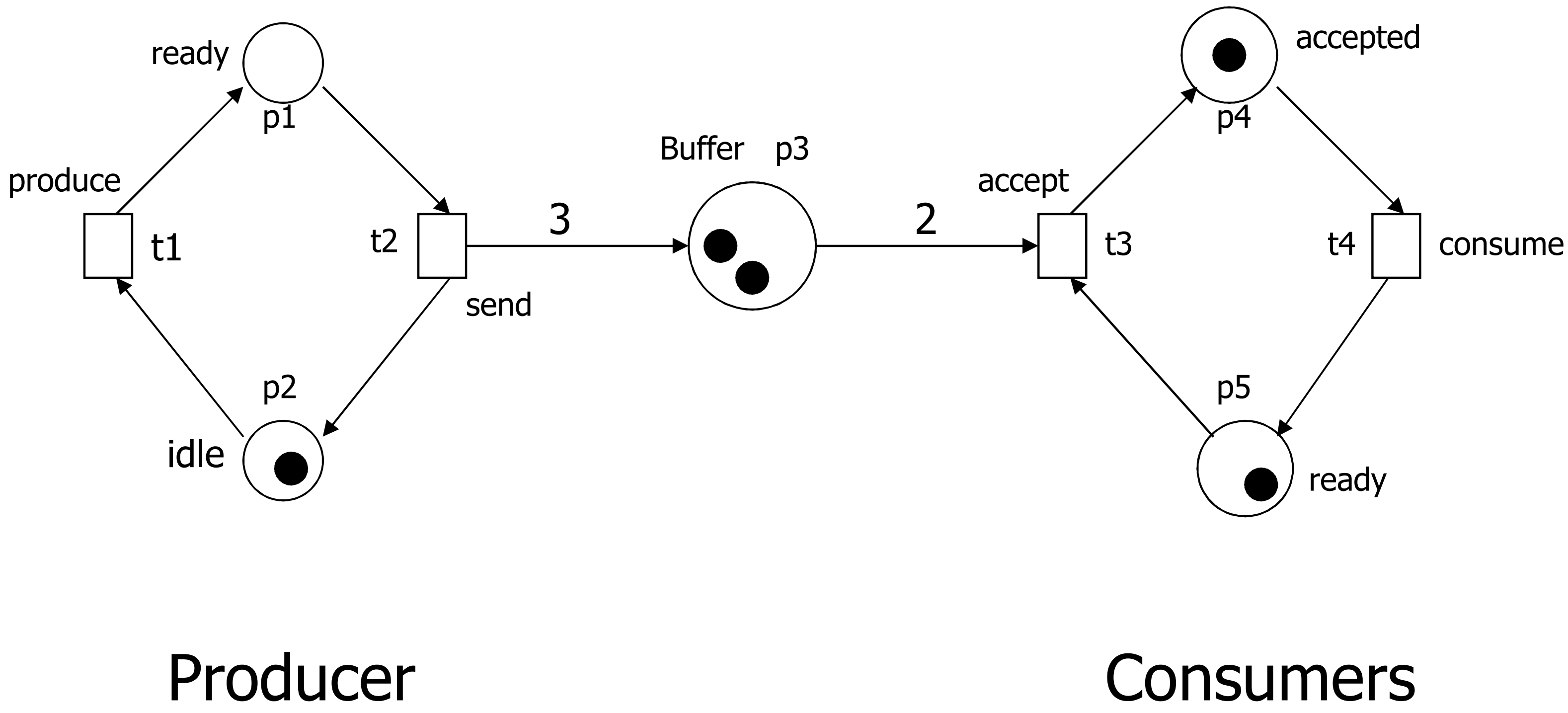
Esempio: produttore / consumatori



Producer

Consumers

Esempio: produttore / consumatori



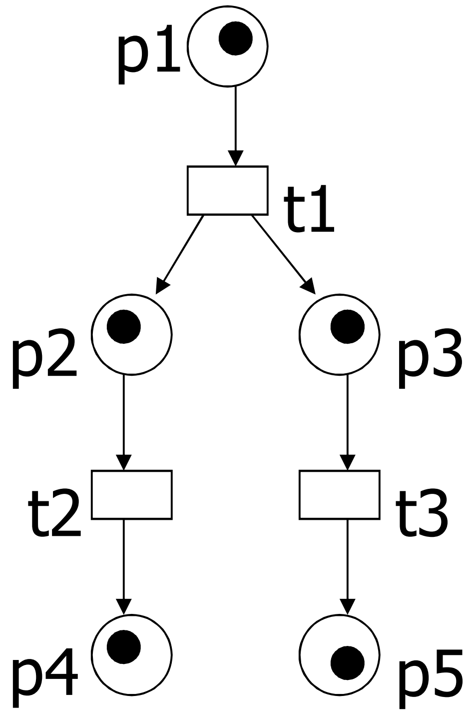
Definizione formale di rete di Petri

- Una **Petri Net** (PN) è una quintupla del seguente tipo:
  - **PN = (P, T, F, W, M_0)**
    - $P = \{p_0, p_1, \ldots, p_m\}$: a finite set of places
    - $T = \{t_1, t_2, \ldots, t_n\}$: a finite set of transitions
    - $F \subseteq (P \times T) \bigcup (T \times P)$: a set of arcs (flow relation)
    - $W: F \rightarrow \{1,2,3,\ldots\}$ weight function
    - $M_0: P \rightarrow \{0,1,2,\ldots\}$ initial marking
    - $P \bigcap T = \varnothing \qquad P \bigcup T \neq \varnothing$

Marking graph

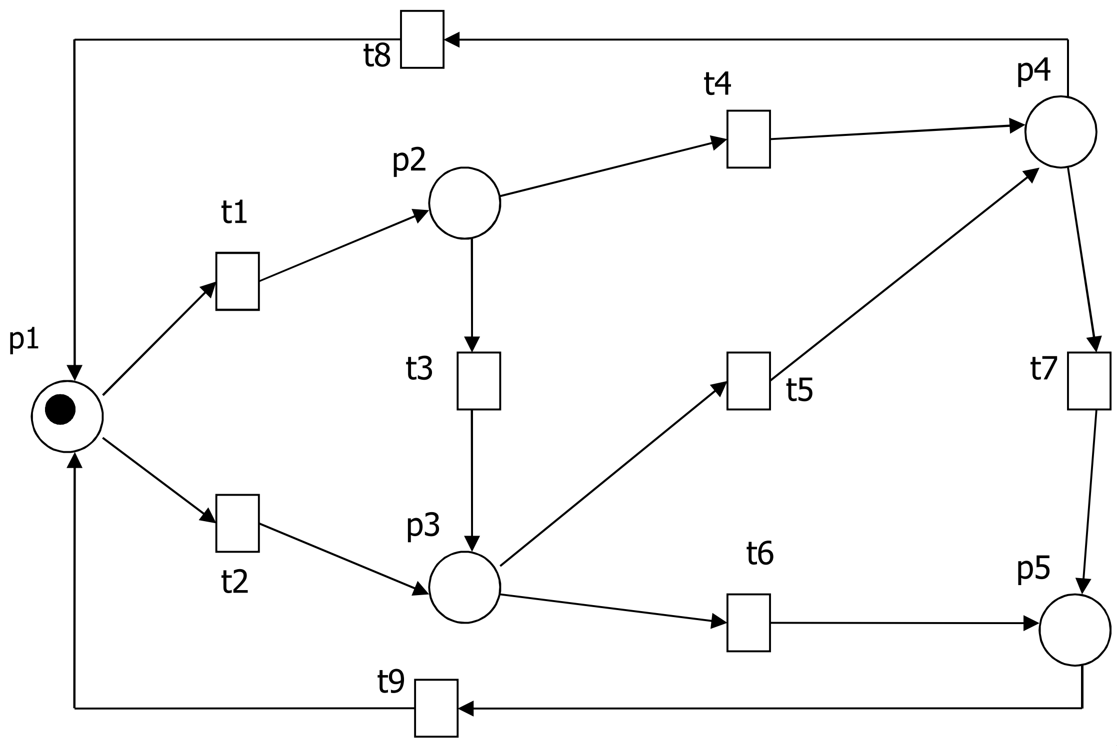
- Il marking graph di una Petri Net è un **LTS** che ne definisce il comportamento
  - **Configurazione** della PN (chiamato **marking**): un **multi-insieme di piazze** che indica la quantità di token presenti in ciascuna piazza
  - **Passaggio di configurazione**: indica in che modo l'**esecuzione di una transizione** della rete di Petri modifica il marking corrente
  - **Configurazione iniziale**: marking iniziale M_0

Esempio: markings



M0 = (1, 0, 0, 0, 0)
M1 = (0, 1, 1, 0, 0)
M2 = (0, 0, 1, 1, 0)
M3 = (0, 0, 0, 1, 1)

Esempio: marking graph



Marking iniziale: M0

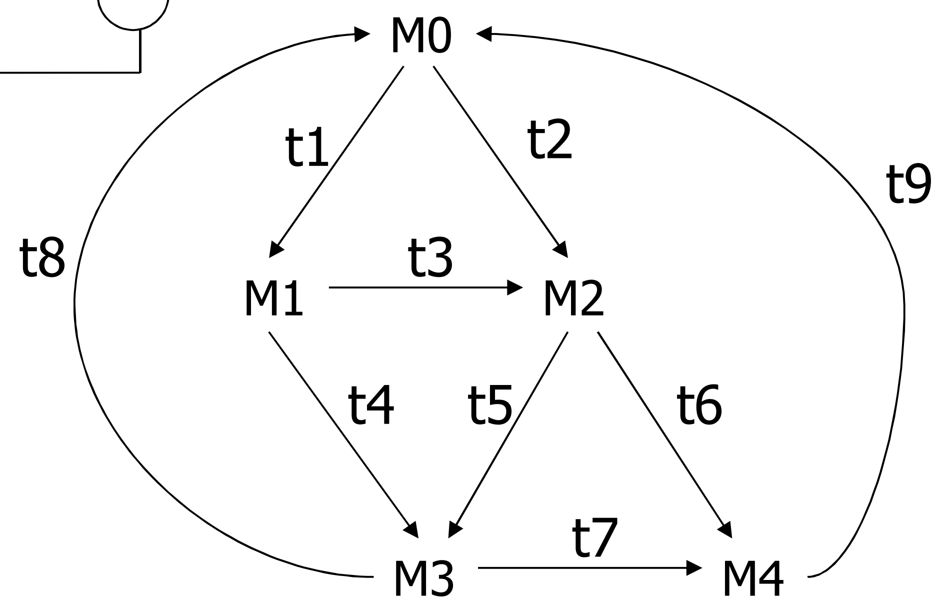
M0 = (1,0,0,0,0)

M1 = (0,1,0,0,0)

M2 = (0,0,1,0,0)

M3 = (0,0,0,1,0)

M4 = (0,0,0,0,1)



Formalmente

- Un **marking** M è un multi-insieme di piazze:

  $M : P \to \{0, 1, 2, \ldots\}$

- Una **transizione** t∈T di una PN è **abilitata** in un marking M se:

  Per ogni $p \in P \quad M(p) \geq W(p, t)$

- L'**esecuzione** di una t∈T abilitata nel marking M fa **passare** la PN al marking M' dato da:

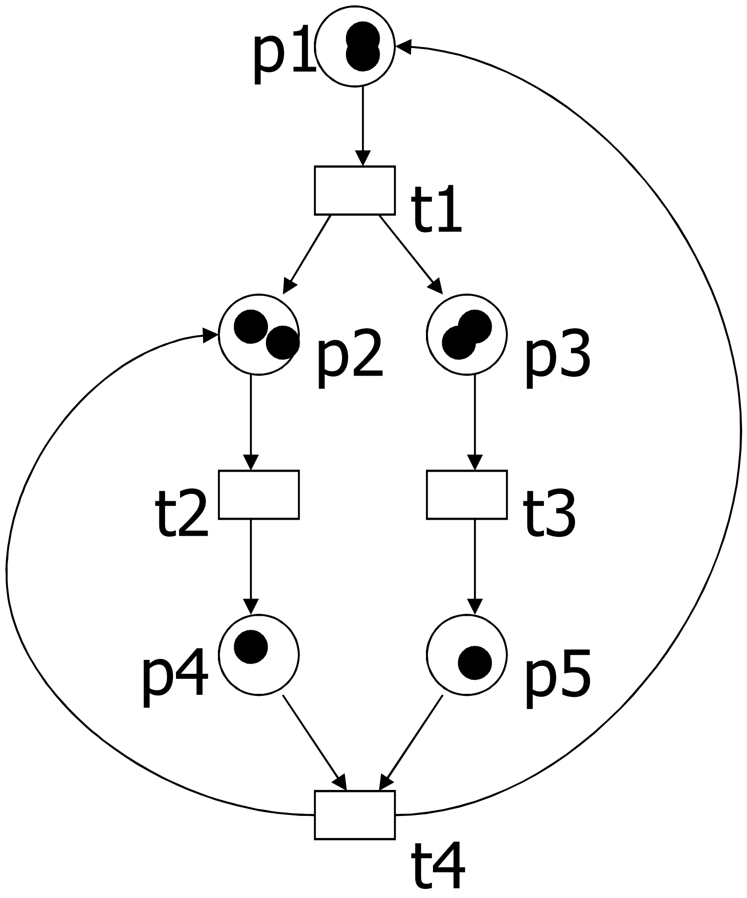
  Per ogni $p \in P \quad M'(p) = M(p) - W(p, t) + W(t, p)$

Nota: si assume che il peso W di un arco non in F sia 0

Marking graph (continua)

- A volte il marking graph risulta essere **infinito**:
  - Succede quando la rete di Petri è “unbounded”, cioè può produrre una quantità illimitata di token
  - Esempio: si veda prossima slide

Marking graph (continua)



M0 = (1, 0, 0, 0, 0)
M1 = (0, 1, 1, 0, 0)

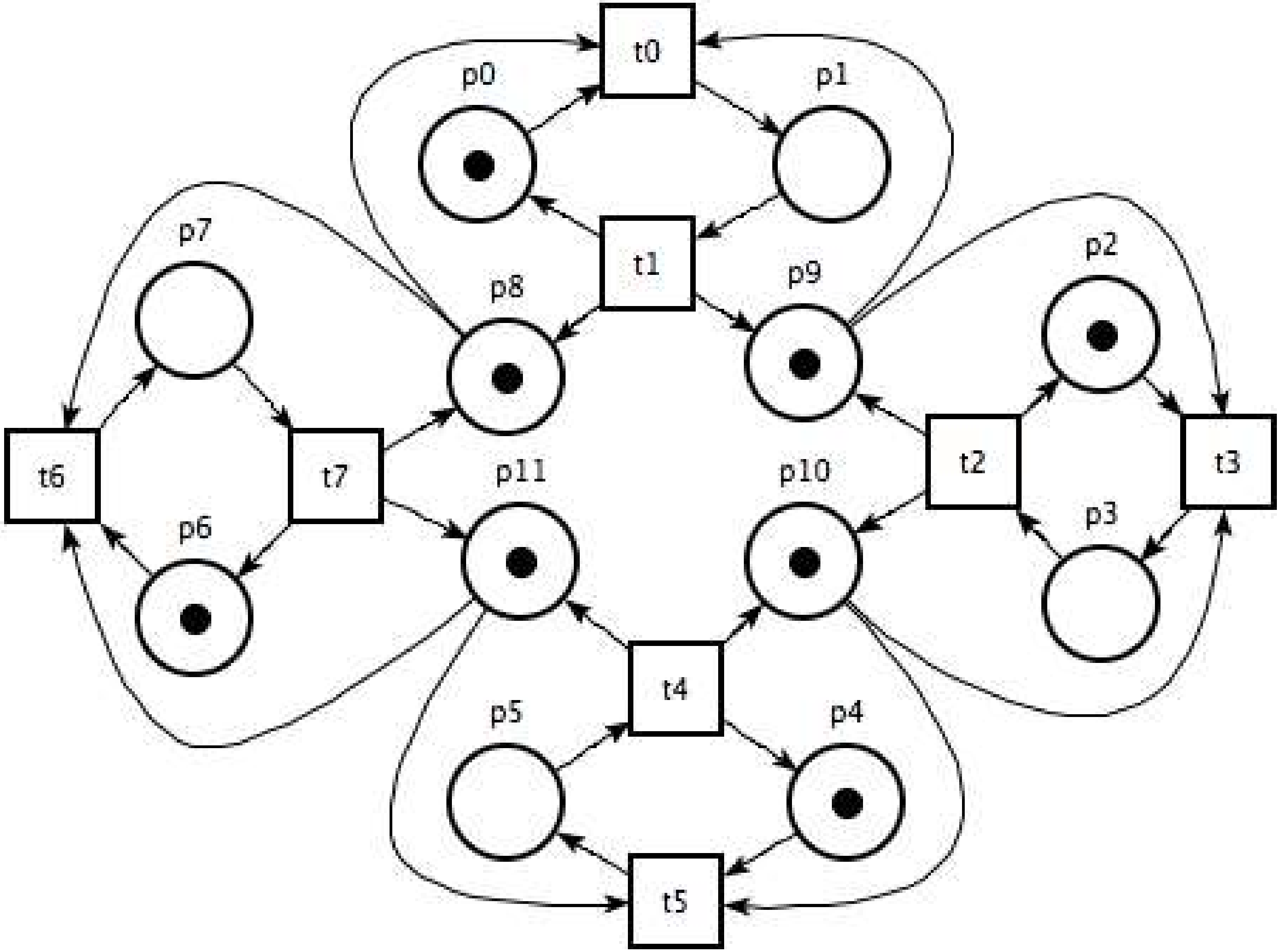
M3 = (0, 0, 0, 1, 1)
M4 = (1, 1, 0, 0, 0)
M5 = (0, 2, 1, 0, 0)

Rete di Petri unbounded

Model checking

- Su Petri nets “bounded” si verifica se una **formula** di LTL è **soddisfatta** come già visto:
  - Si considera il **marking graph** come un **LTS** che rappresenta il comportamento della rete di Petri
  - E poi i **modelli** π di tutte le sue tracce massimali
- Ora però **proposizioni** da **stati** dell’LTS:
  - In questo caso gli stati sono i **marking** della PN
  - Sono definite proposizioni che consentono di osservare le piazze: **proposizione** “**p**” vera se e solo se **piazza p contiene almeno un token**

Esempio: 4 dining philosopher



Esempio: 4 dining philosopher

- Alcune proprietà esprimibili in LTL (alcune vere, alcune false – usare TINA per controllare):
  - [](-(p1 /\ p3))
    due filosofi vicini non mangiano contemporaneamente
  - []((-p8) => (<>p8))
    una forchetta in uso, verrà rilasciata
  - <>p1
    il primo filosofo mangerà
  - -(<>(p1 /\ p5))
    non si raggiunge uno stato in cui due filosofi lontani mangiano insieme