Verifica di Sistemi

Modelli di sistemi con Automi e
Sistemi a Transizioni

Transition systems

- A *transition system* is a structure

  $$\mathbf{TS = (S, S_0, R)}$$

  where:
  - **S** is a finite set of states.
  - $\mathbf{S_0} \subseteq \mathbf{S}$ is the set of initial states.
  - $\mathbf{R} \subseteq \mathbf{S} \times \mathbf{S}$ is a transition relation
  - **R** must be ***total***, that is
    - $\forall s \in S.\ \exists s' \in S.\ (s, s') \in R$ or, equivalently,
    - for every state **s** in **S**, there exists **s'** in **S** such that **(s, s')** is in **R** (the system is non blocking).

Notions and Notations

- **TS = (S, S_0, R)**
- **Transitions**: **(s, s') ∈ R** or **R(s, s')** or **s → s'**
- A (finite) ***path*** from **s** is a sequence of states:

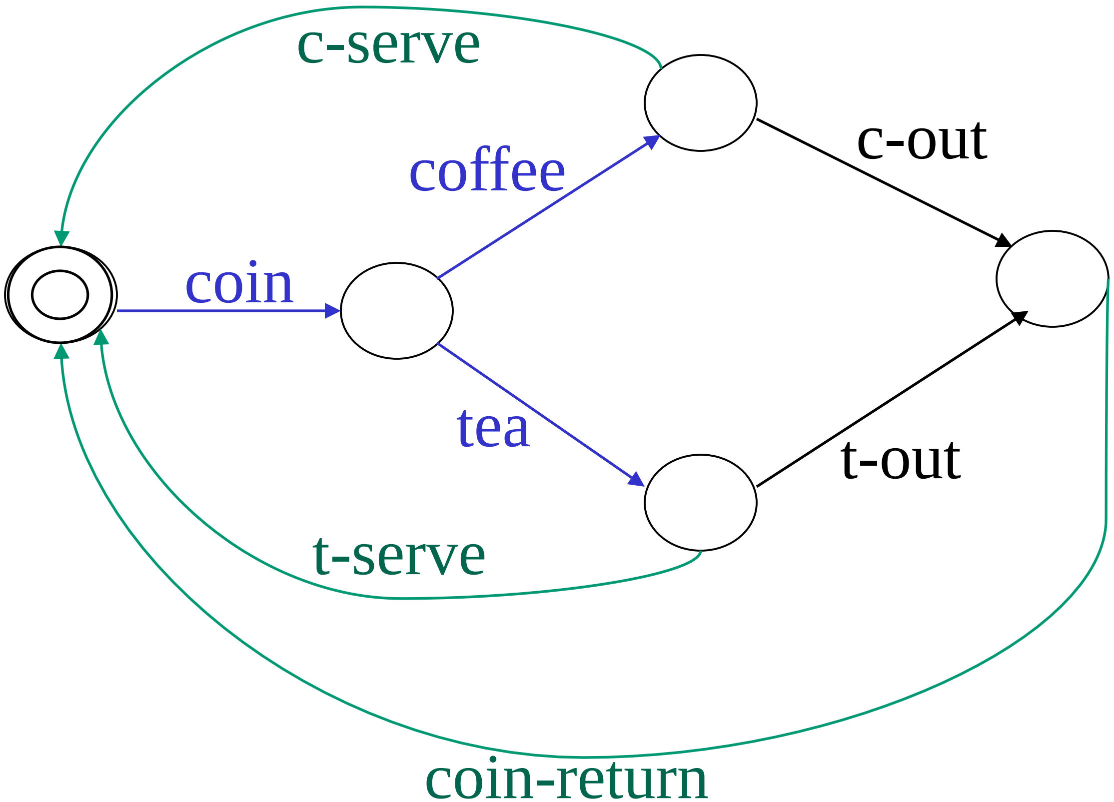
  $$s_1, s_2, \ldots, s_n$$

  such that
  - $s = s_1$
  - $s_i \rightarrow s_{i+1}$ for $0 < i < n$.
- It is from **s** to **s'** if $s_n = s'$.
- An **infinite** path from **s** is an ***infinite sequence*** $s_1, s_2, \ldots, s_n, \ldots$, satisfying the same conditions above

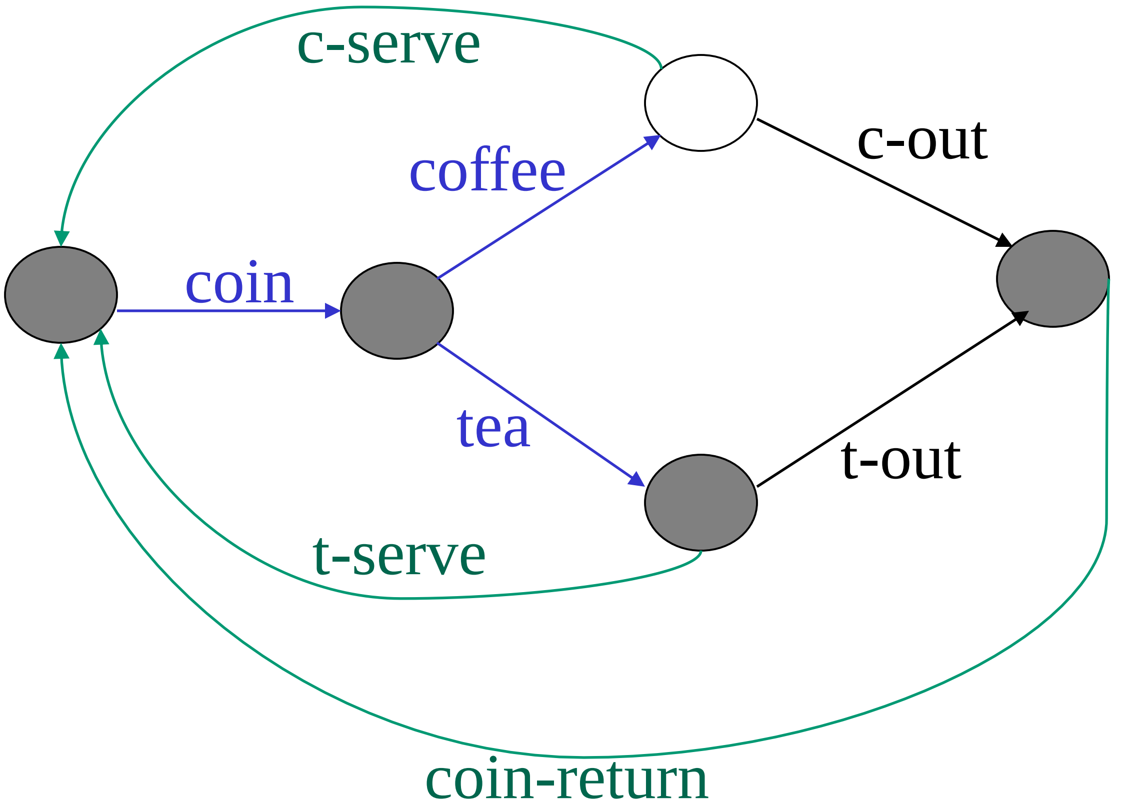
Labeled transition systems

- Sometimes we may use a ***finite*** set of actions:
  - **Act = {a, b, ..}**
- The actions will be used to label the transitions.
- **TS = (S, Act, S_0, R)**
  - **R ⊆ S × Act × S**, labeled transitions.
  - **(s, a, s') ∈ R** - **R(s, a, s')** - $s \xrightarrow{a} s'$

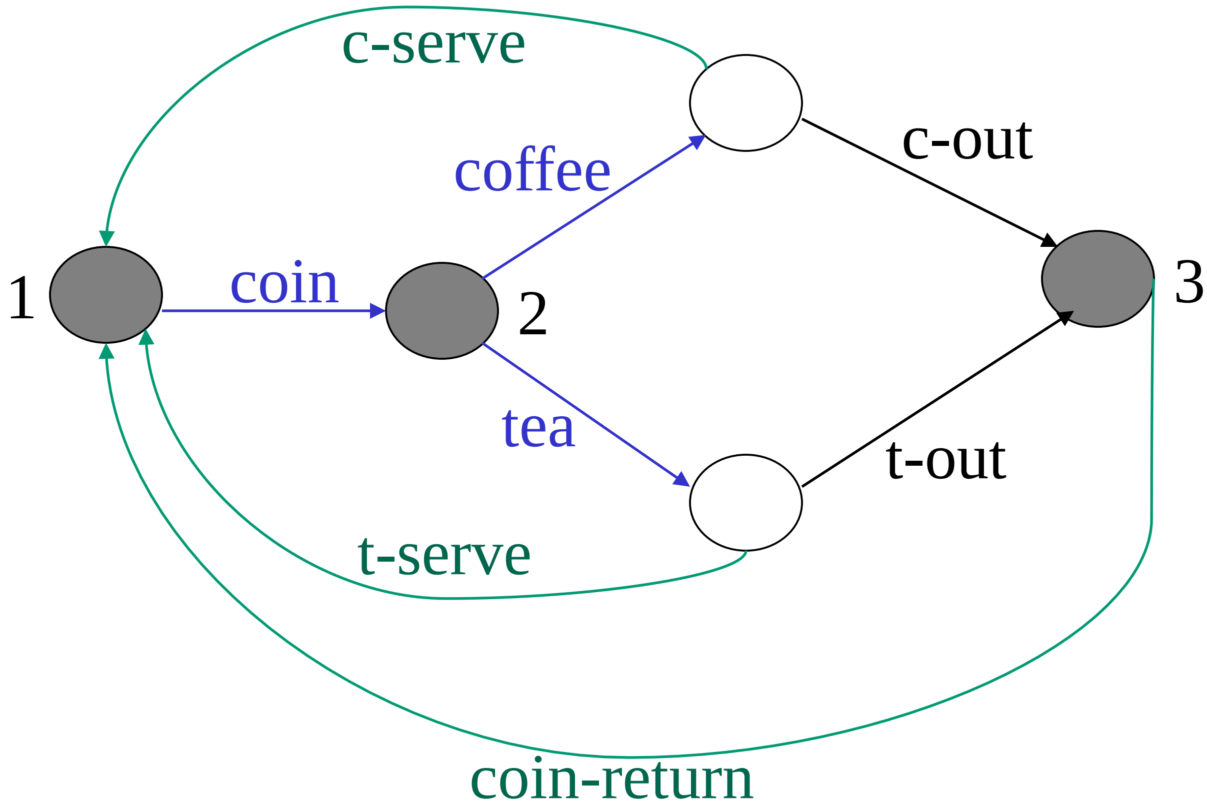
A vending machine



A path



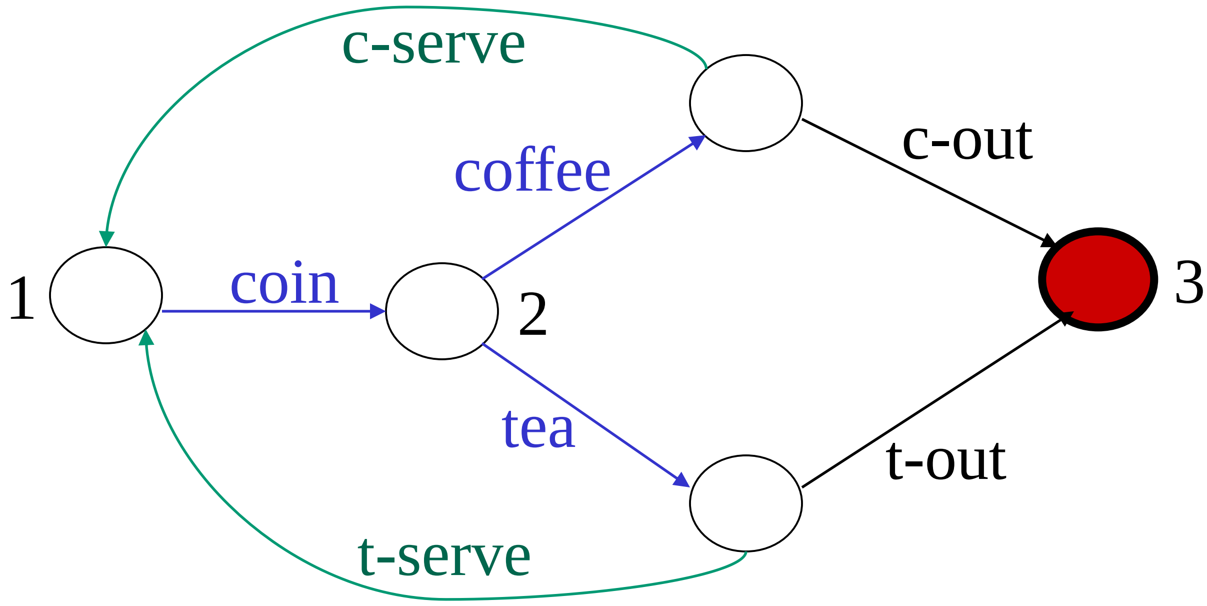
A non-path



1 2 3 No!

3 1 2 yes!

A non-total transition relation



Kripke Structures

- **AP** is a finite set of atomic propositions.
  - "**value of x is 5**"
  - "**x = 5**"
- **M = (S, S_0, R, L)**, a Kripke Structure.
  - **(S, S_0, R)** is a transition system.
  - **$L : S \longrightarrow 2^{AP}$**
  - **2^{AP}** ---- The set of subsets of AP

    (**$L(s) \in 2^{AP}$** identifies a **state**

    2^{AP} identifies the **state space**)

Kripke Structures

- The atomic propositions and **L** together convert a transitions system into a model.
- We can start interpreting ***formulas*** over the ***Kripke structure***.
- The atomic propositions make basic (easy) assertions about system states.

Automata and Kripke Structures

- **AP - set of elementary property**
- **$<S,A,R,s_0,L>$**
- **S - set of states**
- **A - set of transition labels**
- **$R \subseteq S \times A \times S$ - (labeled) transition relation**
- **L - interpretation mapping $L: S \longrightarrow 2^{AP}$**
- **In *FO representation* we would need two sets of variables: V and Act (for actions or input).**

Modeling Data-Dependent Systems

- Let $Var = \{v_1, v_2, \ldots, v_k\}$ be a set of variables with values in domain $D = \cup_{1 \le i \le k} D_i$ (D_i the domain for v_i)

- A Program graph over Var is a tuple

$$\boxed{PG = \langle Loc, Act, Effect, \hookrightarrow, Loc_0, g_0 \rangle}$$

Where

– Loc is a set of locations and Act a set of actions

– $Effect : Act \times Eval(Var) \to Eval(Var)$ captures the effects of the actions on the variables

– $\hookrightarrow \subseteq Loc \times Cond(Var) \times Act \times Loc$

– Loc_0, is the set of initial locations and g_0 is the initial condition

Program Conditions and Actions

- Let $Expr(Var \cup D)$ be the set of (arithmetic) expression over $Var \cup D$.
  - examples: $v+1$, $v+2*d$, $v+2*v'$,… (with $d \in D$)
- The conditions $Cond(Var)$ on Var is the set of Boolean combinations of comparisons of the form

$$exp_1 \bullet exp_2$$

with $\bullet \in \{<,>,\leq,\geq,=,\neq\}$ and $exp_i \in Expr(Var \cup D)$

- The actions on Var is the set of assignements of

$$v := exp$$

where $v \in Var$ and $exp \in Expr(Var \cup D)$

State space

- The ***state space*** of a program is the set of ***all its possible valuations Eval(Var)*** of the state variables.
- For example, if **V={a, b, c}** and the variables range over the natural numbers, then the ***state space*** includes:

<a=0,b=0,c=0>, <a=1,b=0,c=0>,

<a=1,b=1,c=0>, <a=932,b=5609,c=6658>

…

The set Loc can be considered as the domain of an implicit variable pc encoding a program counter.

Action Effects

- Given an evaluation $\eta \in \text{Eval(Var)}$ and an action of the form

$$a \stackrel{\text{def}}{=} v := exp$$

Where exp is an expression on $Var \cup D$, the effect of a on η is

$$\text{Effect}(a,\eta) = \eta[v \leftarrow exp]$$

- For example if $a = v := v+1$ and $\eta(v) = 5$, then $\text{Effect}(a,\eta)$ is the valuation η' such that $\eta'(v) = 6$

Transition system of a Program Graph

If $PG = \langle Loc, Act, Effect, \hookrightarrow, Loc_0, g_0 \rangle$ then

$$\boxed{TS(PG) = \langle S, Act, \rightarrow, S_0, AP, L \rangle}$$

- $S = Loc \times Eval(Var)$
- $\rightarrow \subseteq S \times Act \times S$ such that
  - If $l \xhookrightarrow{g:a} l'$ in PG and $\eta \models g$, then $\langle l,\eta \rangle \xrightarrow{a} \langle l',\eta' \rangle$ in TS(PG), with $\eta' = Effect(a,\eta)$
- $S_0 = \{\langle l,\eta \rangle \mid l \in Loc_0$ and $\eta \models g_0\}$
- $AP = Loc \cup Cond(Var)$
- $L(\langle l,\eta \rangle) = \{l\} \cup \{g \mid g \in Cond(Var)$ and $\eta \models g\}$

Composition and Synchronization

- Complex systems are very hard to specify in their entirety.
- The difficulty is to account for all the possible interactions among their components, in particular if they execute in a concurrent fashion.
- The natural approach is to specify them as ***composition*** of smaller and sequential ***subsystems*** (or ***modules***), which are easier to describe.
- We need to describe the way in which these modules coordinate (composition) and cooperate (communication).
- There are several methods to define composition and communication (i.e., to ***synchronize*** the components).

Synchronous Composition

- The system model is the cartesian product of the simpler modules.
- Let $TS_1, \ldots, TS_n$ be n TSs, s.t. $TS_i = <S_i, A_i, R_i, S_{i0}>$
- Then $TS = TS_1 \| \ldots \| TS_n = <S, A, R, S_0>$ is s.t.
  - $S = S_1 \times \ldots \times S_n$
  - $A = A_1 \times \ldots \times A_n$
  - $S_0 = S_{10} \times \ldots \times S_{n0}$
  - R contains $<s_1, \ldots, s_n> \xrightarrow{<a_1, \ldots, a_n>} <s_1', \ldots, s_n'>$, if $s_i \xrightarrow{a_i} s_i'$ for all $1 \leq i \leq n$ and $<a_1, \ldots, a_n> \in A$

Asynchronous Composition

- The system model is the cartesian product of the simpler modules with an additional null action -
- Let $TS_1, \ldots, TS_n$ be n TSs, s.t. $TS_i = \langle S_i, A_i, R_i, S_{i0} \rangle$
- Then $TS = TS_1 \mid\mid \ldots \mid\mid TS_n = \langle S, A, R, S_0 \rangle$ is s.t.
  - $S = S_1 \times \ldots \times S_n$
  - $\boxed{A = (A_1 \cup \{\text{-}\}) \times \ldots \times (A_n \cup \{\text{-}\})}$
  - $S_0 = S_{10} \times \ldots \times S_{n0}$
  - R contains $\langle s_1, \ldots, s_n \rangle \xrightarrow{\langle a_1, \ldots, a_n \rangle} \langle s_1', \ldots, s_n' \rangle$, if, for all $1 \leq i \leq n$, $a_i = \text{-}$ or $s_i \xrightarrow{a_i}_i s_i'$ and $a_i \neq \text{-}$, and $\langle a_1, \ldots, a_n \rangle \in A$

Asynchronous Composition: Interleaving

- The system model is the cartesian product of the simpler modules with an additional null action -
- Let $TS_1, \ldots, TS_n$ be n TSs, s.t. $TS_i = <S_i,A_i,R_i,S_{i0}>$
- Then $TS = TS_1 \parallel \ldots \parallel TS_n = <S,A,R,S_0>$ is s.t.
  - $S = S_1 \times \ldots \times S_n$
  - $A \subset (A_1 \cup \{\text{-}\}) \times \ldots \times (A_n \cup \{\text{-}\})$ s.t. $<a_1,\ldots,a_n> \in A$ iff $a_i \in A_i$ implies $a_j = \text{-}$, for all $j \neq i$
  - $S_0 = S_{10} \times \ldots \times S_{n0}$
  - R contains $<s_1,\ldots,s_n> \xrightarrow{<a_1,\ldots,a_n>} <s_1',\ldots,s_n'>$, if, for all $1 \leq i \leq n$, $a_i = \text{-}$ or $s_i \xrightarrow{a_i}_i s_i'$ and $a_i \neq \text{-}$, and $<a_1,\ldots,a_n> \in A$

Interleaving of Program Graphs

- Let

$$PG_i = \langle Loc_i, Act_i, Effect_i, \hookrightarrow_i, Loc_{i0}, g_{i0} \rangle$$

be n program graphs, each one over Var_i .

Then the Program Graph of the interleaving composition of $PG = PG_1 \parallel PG_2 \parallel \ldots \parallel PG_n$ is

$$\boxed{PG = \langle Loc, Act, Effect, \hookrightarrow, Loc_0, g_0 \rangle}$$ where

– $Loc = Loc_1 \times Loc_2 \times \ldots \times Loc_n$

– $Act = \biguplus_{1 \leq i \leq n} Act_i$ ($\uplus$ disjoint union) and $Var = \bigcup_{1 \leq i \leq n} Var_i$

– $Loc_0 = Loc_{10} \times Loc_{20} \times \ldots \times Loc_{n0}$

– $g_0 = g_{10} \wedge g_{20} \wedge \ldots \wedge g_{n0}$

Interleaving of Program Graphs

- If $l_i \overset{g:a}{\hookrightarrow}_i l_i'$, then

  $$\langle l_1,\ldots, l_i,\ldots, l_n\rangle \overset{a}{\hookrightarrow} \langle l_1,\ldots, l_i',\ldots, l_n\rangle$$ in PG.

- $Effect : Act \times Eval(Var) \rightarrow Eval(Var)$ is defined as:

$$\boxed{\mathrm{Effect}(a,\eta)(v) = \begin{cases} \mathrm{Effect}_i^{\eta}(a,\eta_i)(v) & \text{if } a \in \mathrm{Act}_i \\ \eta(v) & \text{otherwise} \end{cases}}$$

where η_i is the restriction of η to the variables in Var_i and $\mathrm{Effect}_i^{\eta}(a,\eta_i)$ is the extension of $\mathrm{Effect}_i(a,\eta_i)$ to the variables in Var that gives the value $\mathrm{Effect}_i(a,\eta_i)(v)$, for all the variables $v \in Var_i$, and the value $\eta(v)$ for all the variables $v \in Var \setminus Var_i$.

A Mutual Exclusion Protocol

PROCESS A

```
repeat
   non-critical code
   /* entry_protocol; */
   flag1 := true;
   while flag2 do
       skip;
  /* end entry_protocol; */
  critical section;
  /* exit_protocol; */
  flag1 := false;
  /* end exit_protocol; */
 non-critical code
forever;
```

PROCESS B

```
repeat
   non-critical code
   /* entry_protocol; */
   flag2 := true;
   while flag1 do
       skip;
  /* end entry_protocol; */
  critical section;
  /* exit_protocol; */
  flag2 := false;
  /* end exit_protocol; */
 non-critical code
forever;
```

A Mutual Exclusion Protocol

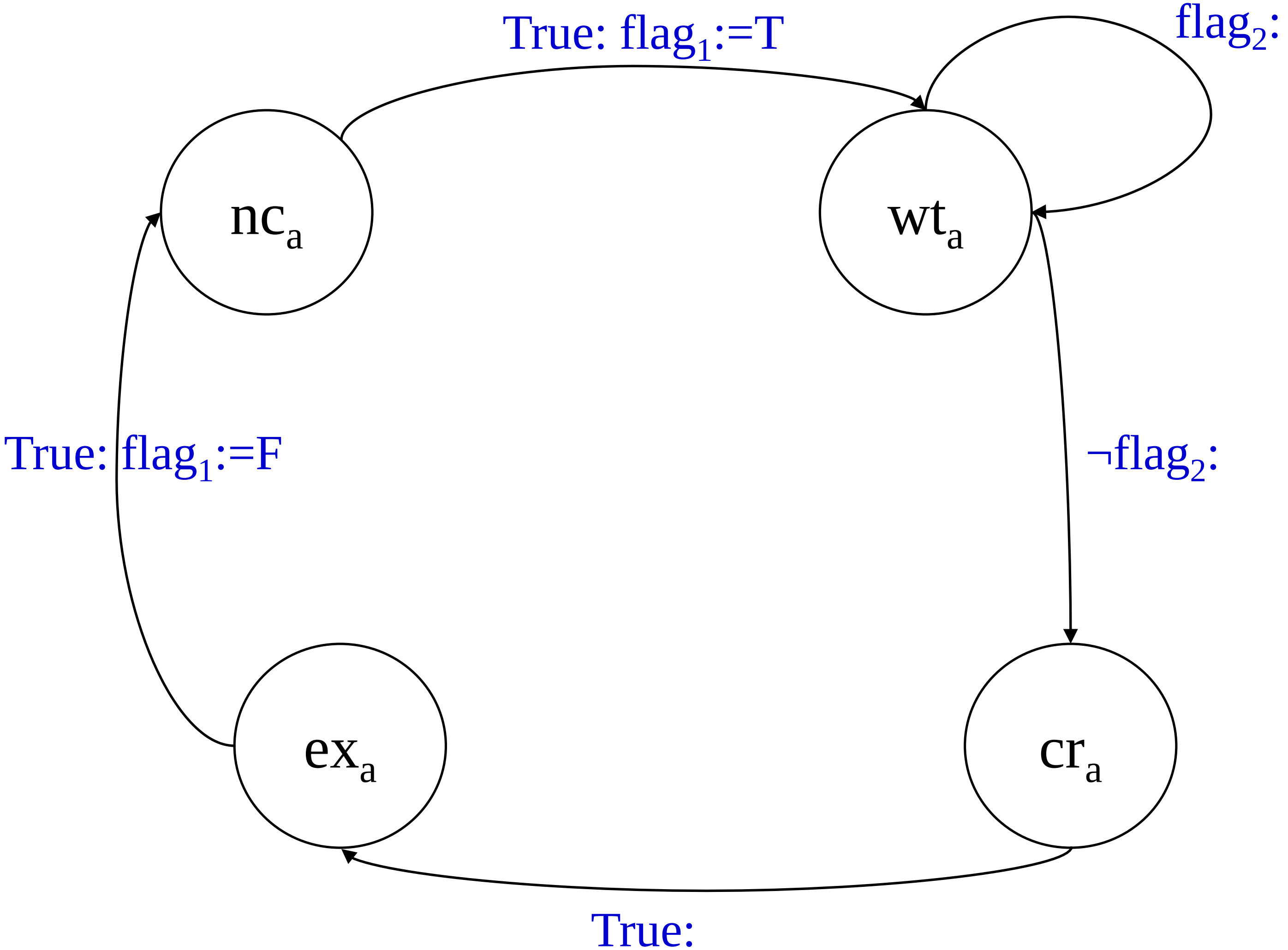
PROCESS A

```
 repeat
    non-critical code                 nca
    /* entry_protocol; */
    flag1 := true;
    while flag2 do                    wta
        skip;
   /* end entry_protocol; */
   critical section;                  cra
   /* exit_protocol; */
   flag1 := false;                    exa
   /* end exit_protocol; */
  non-critical code
forever;
```

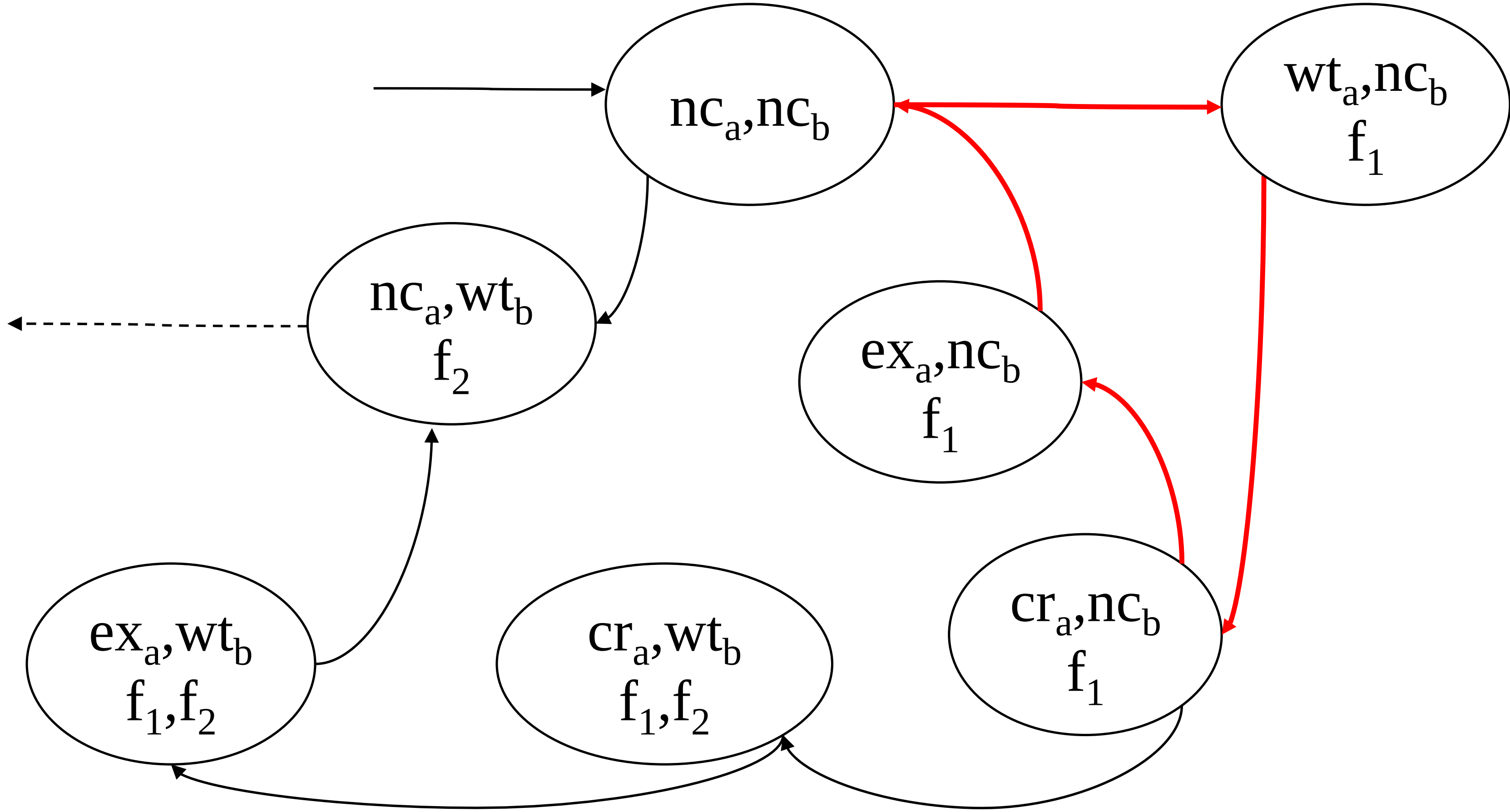
PROCESS B

```
 repeat
    non-critical code
    /* entry_protocol; */
    flag2 := true;
    while flag1 do
        skip;
   /* end entry_protocol; */
   critical section;
   /* exit_protocol; */
   flag2 := false;
   /* end exit_protocol; */
  non-critical code
forever;
```

The Automaton for Process A



Composition: LTS (fragment)



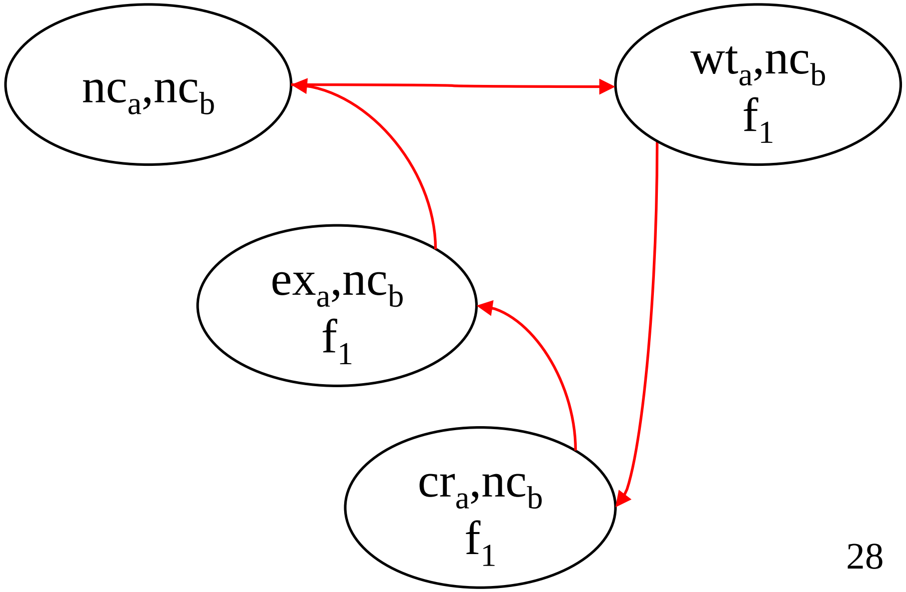
Fairness

- Fairness constraints are meant to capture general constraints of «good behavior» of concurrent systems.
- For instance: concurrent systems (multi-threaded, multi-process) rely on a scheduling mechanism that select the next process (or thread) to execute during computation.
- Fairness constraints capture very general constraints that every reasonable scheduling mechanism should guarantee, without requiring any detailed specification of the scheduling mechanism itself.

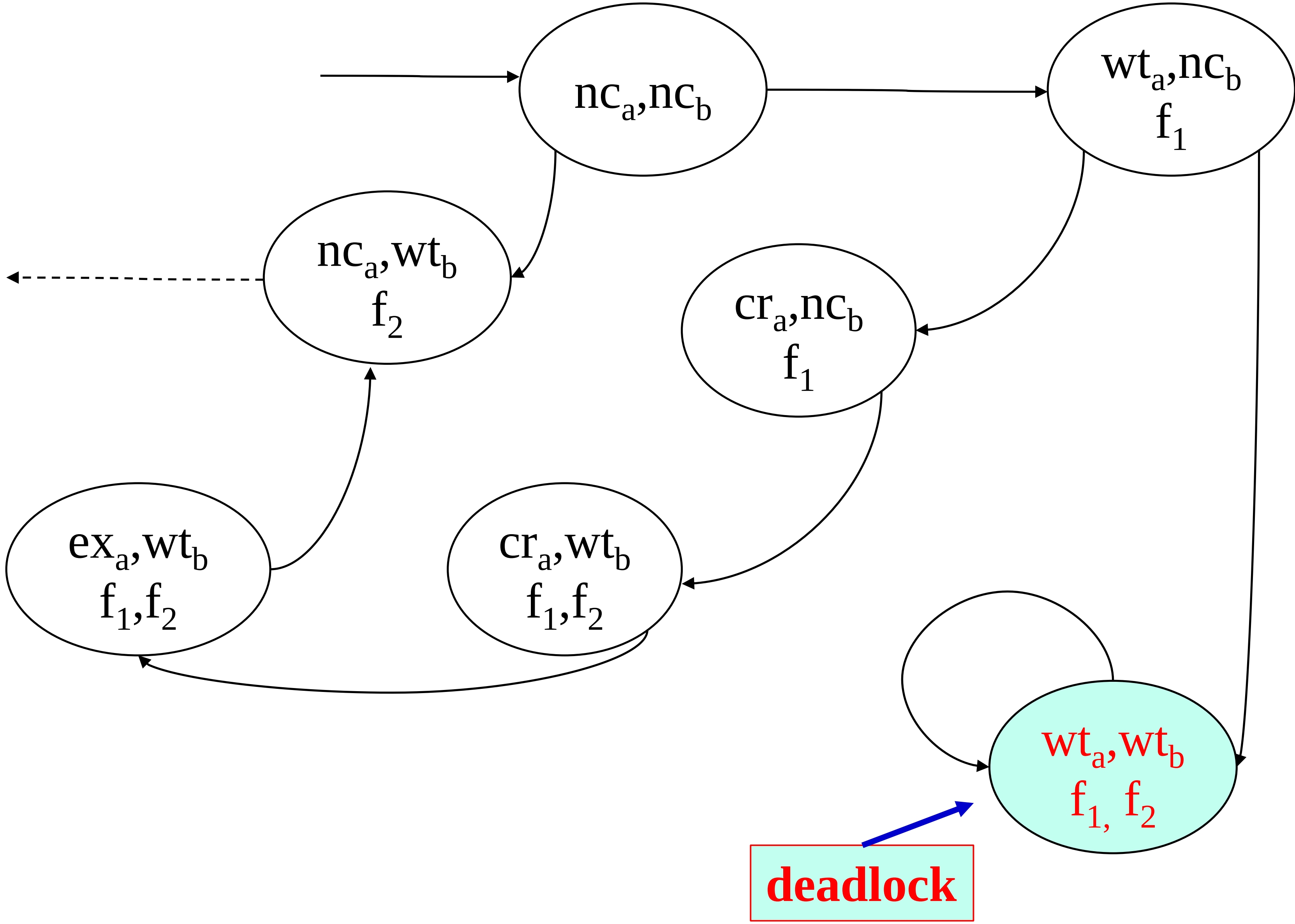
Popular Fairness Conditions

- Unconditional fairness: each process must be scheduled for execution infinitely often
- Weak fairness: a process continuously enabled must be scheduled for execution infinitely often
- Strong fairness: a process infinitely often enabled must be scheduled for execution infinitely often.

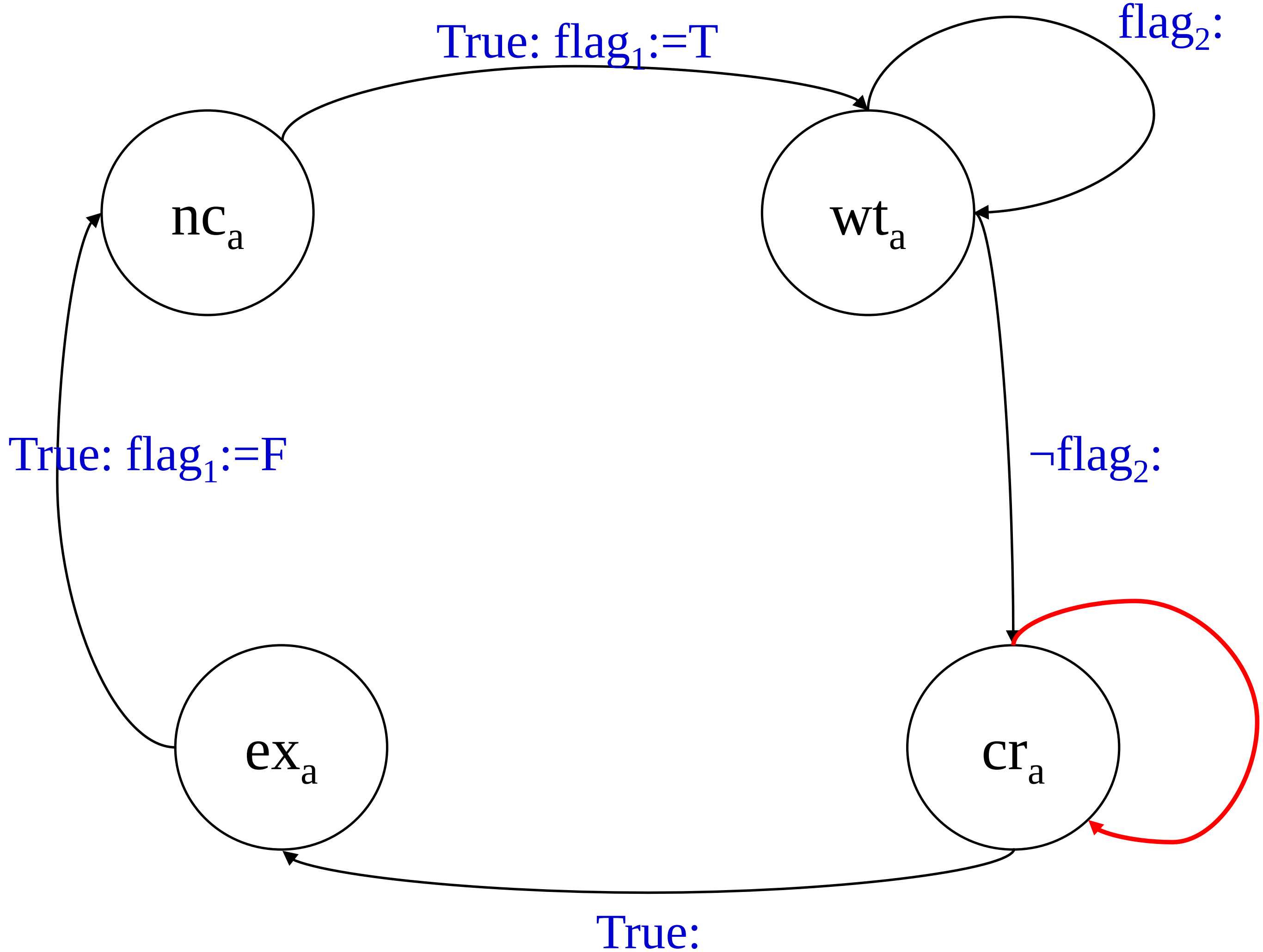
Under any of the above fairness conditions the computation remaining in the loop forever is no longer admissible.



Composition: LTS (fragment)



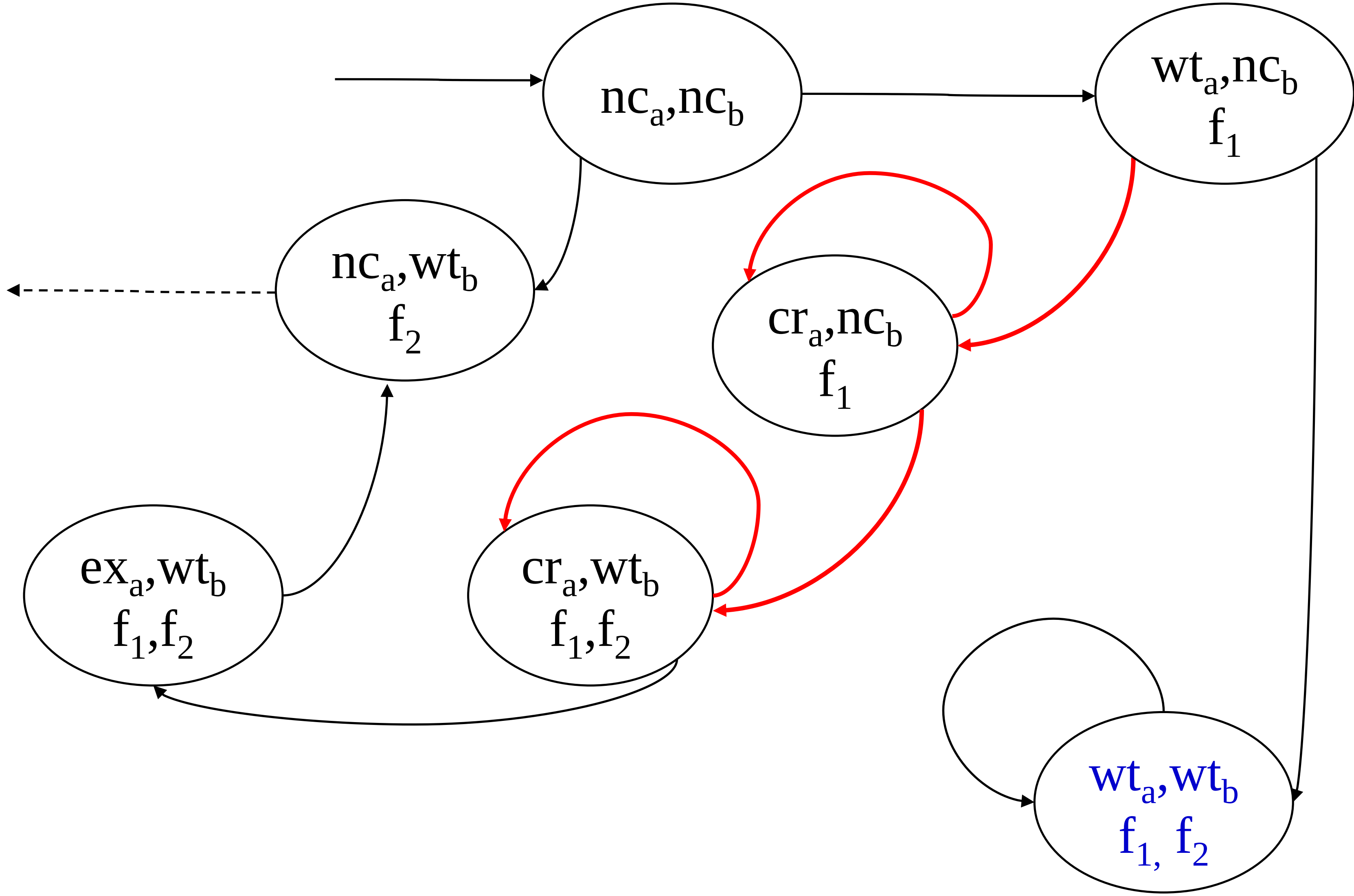
Adding non-determinism



Non Determinism

- Non deterministic choices are often used to model partially specified systems or behaviors.
- Non determinism used to:
  - Model systems under incomplete information on its behavior (e.g. system internal decisions not known, unpredictable behavior of the environment,…).
  - Increase the abstraction level of the specification: less details are explicitly given so as to obtain more compact/general models (e.g., state sequences involving only internal computations of the system modeled by a single state).

Composition: LTS (fragment)

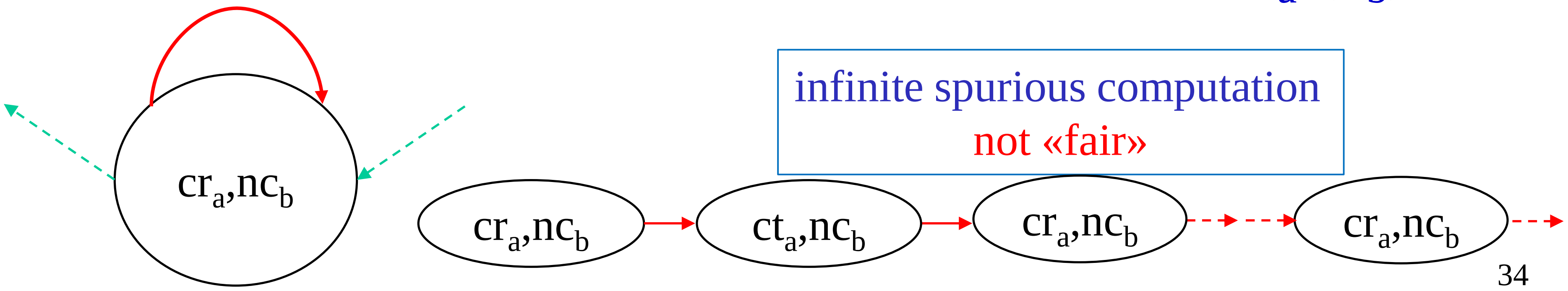


Problems with non-determinism

- May introduce «spurious» computations in the model.
- A computation is «spurious» if it is admitted in the model but not by the actual system.
- E.g., Process A enters its critical section but never leaves it.
- «Spurious» computations can be eliminated by means of appropriate fairness constraints
- E.g.: Process A (B) must be infinitely often outside its critical sections, i.e. in a state different from c_a (c_b).
  - the computation where Process A never releases the critical section is no longer an admissible computation.

Problems with non-determinism

- May introduce «spurious» computations in the model.
- A computation is «spurious» if it is admitted in the model but not by the actual system.
- E.g., Process A enters its critical section but never leaves it.
- «Spurious» computations can be eliminated by means of appropriate fairness constraints
- E.g.: Process A (B) must be infinitely often outside its critical sections, i.e. in a state different from c_a (c_b).



Atomic transition

- Each ***atomic transition*** represents a small piece of code (or ***execution step***), such that ***no smaller*** peace of code (or ***step***) is observable.
- Often is not easy to identify which actions are atomic transitions and which are not.
- Atomicity may even depend on the abstraction level of the specification.
- Is a:=a+1 atomic?

Atomic transition

- Each ***atomic transition*** represents a small peace of code (or ***execution step***), such that ***no smaller*** peace of code (or ***step***) is observable.
- Often is not easy to identify which actions are atomic transitions and which are not.
- Atomicity may even depend on the abstraction level of the specification.
- Is a:=a+1 atomic? It may or may not be!
- In some systems it is, e.g., when a is a register and the transition is executed using an inc command.

(Non) Atomicity (race conditions)

- Execute the following when **a=0** in two concurrent processes:

P1:a=a+1 **P2:a=a+1**

- Result: **a=2**.
- Is this always the case?

- Consider the actual translation:

P1:load R1,a
inc R1
store R1,a

P2:load R2,a
inc R2
store R2,a

- **a may also be 1**

Mutual Exclusion II

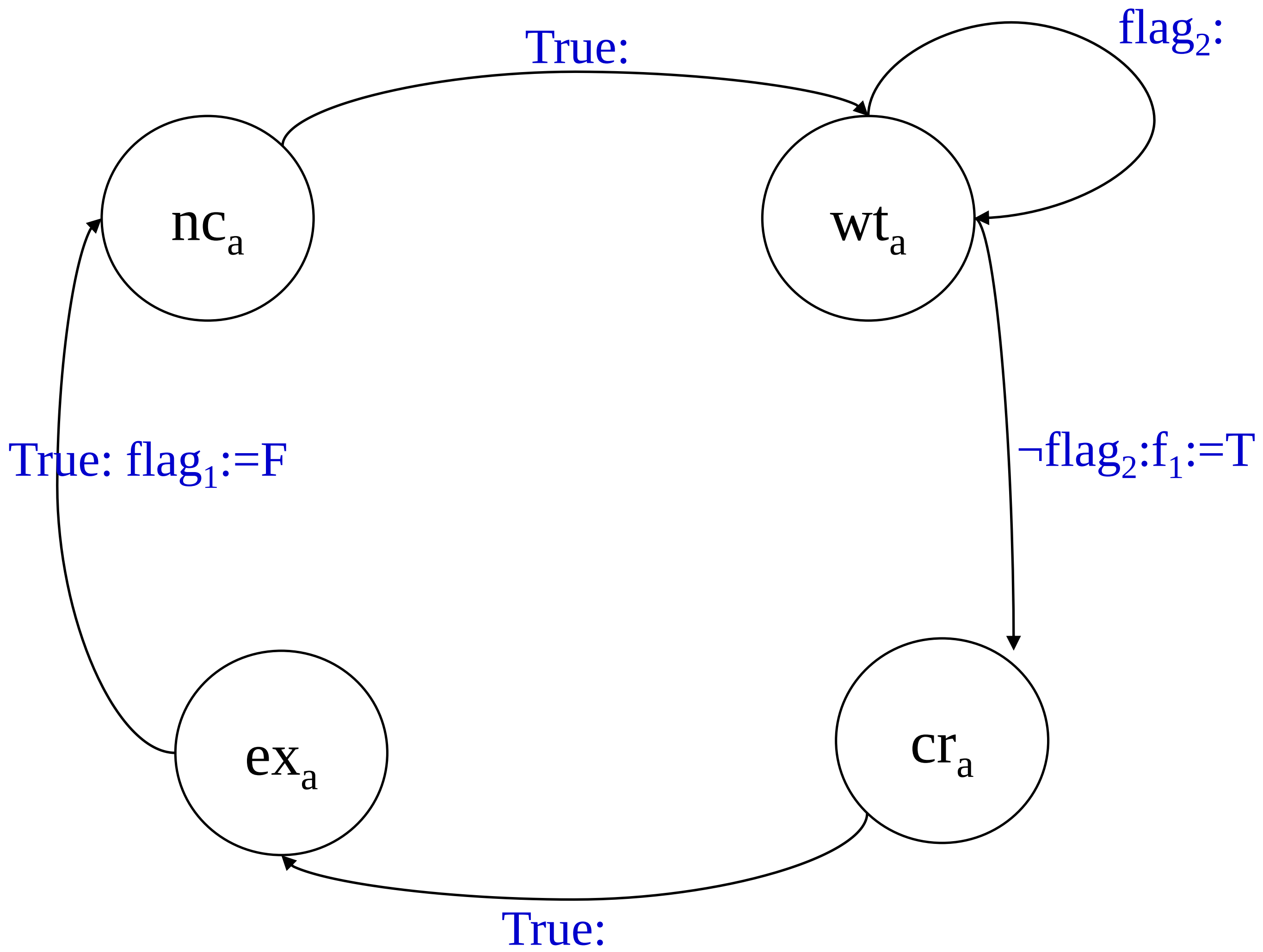
PROCESS A

```
 repeat
    non-critical code
    /* entry_protocol; */
    while flag2 do
        skip;
    flag1 := true;
    /* end entry_protocol; */
    critical section;
    /* exit_protocol; */
    flag1 := false;
    /* end exit_protocol; */
    non-critical code
forever;
```

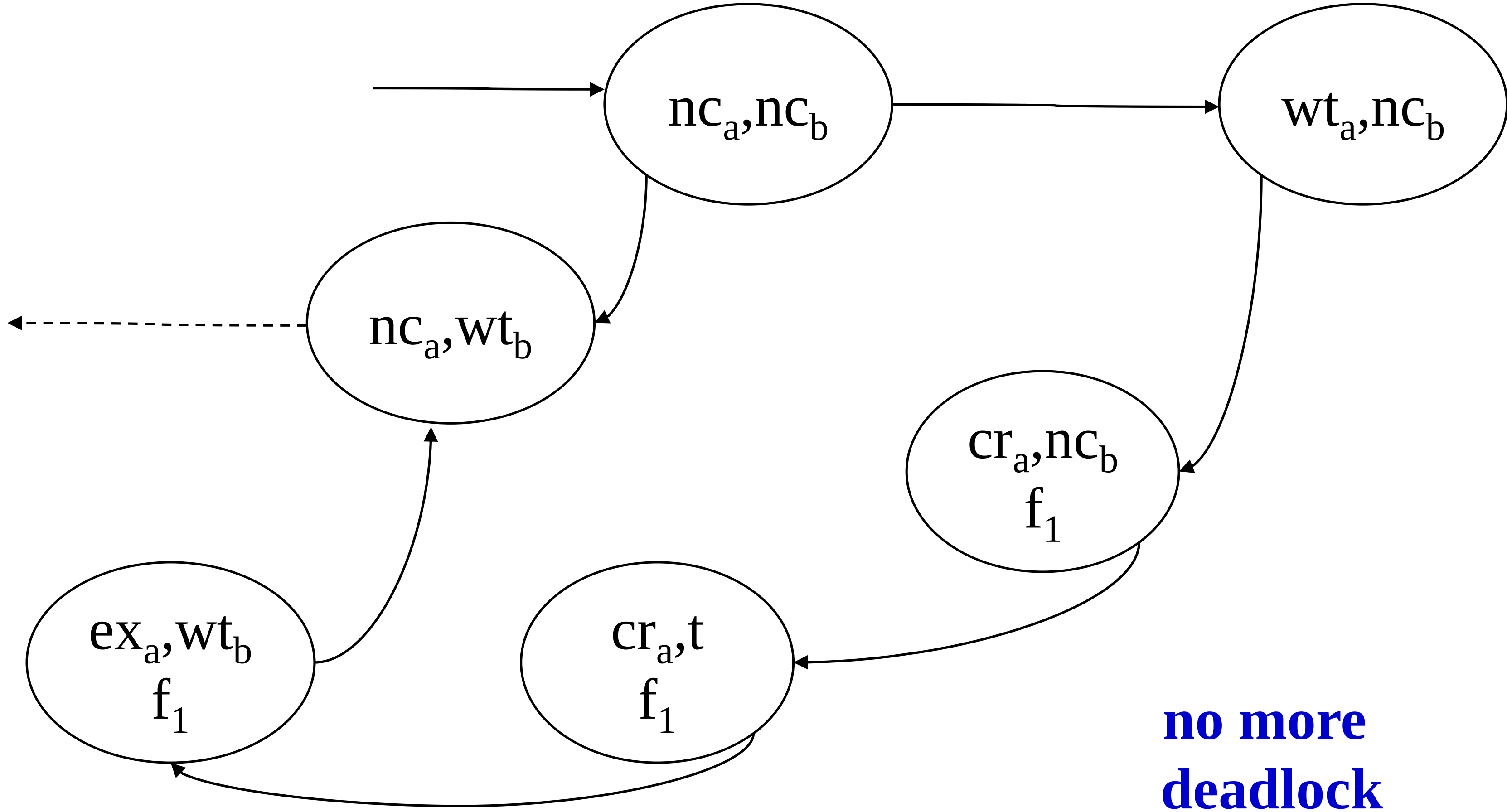
PROCESS B

```
 repeat
    non-critical code
    /* entry_protocol; */
    while flag1 do
        skip;
    flag2 := true;
    /* end entry_protocol; */
    critical section;
    /* exit_protocol; */
    flag2 := false;
    /* end exit_protocol; */
    non-critical code
forever;
```

A possible automaton for Process A

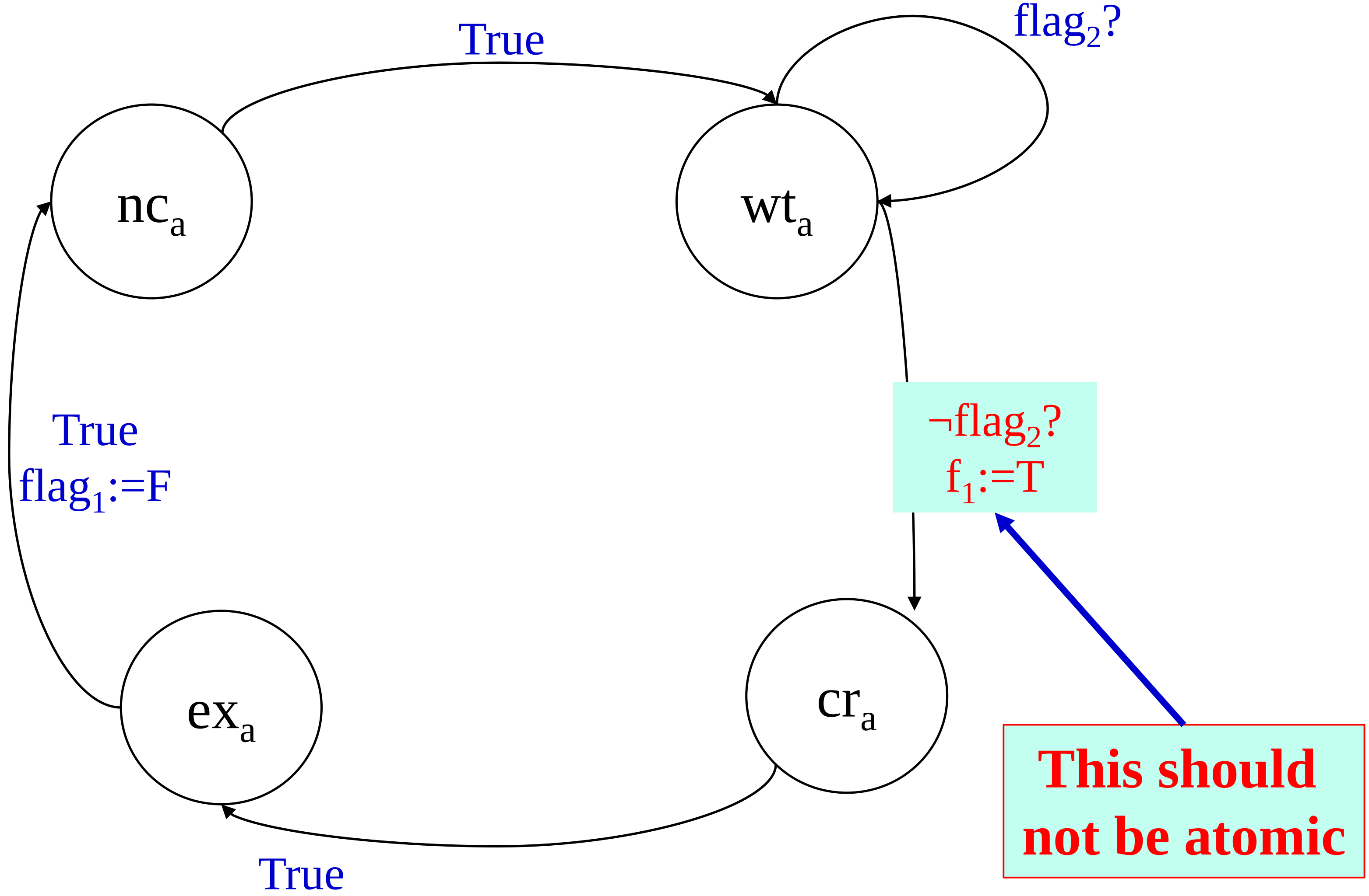


Composition: LTS (fragment)

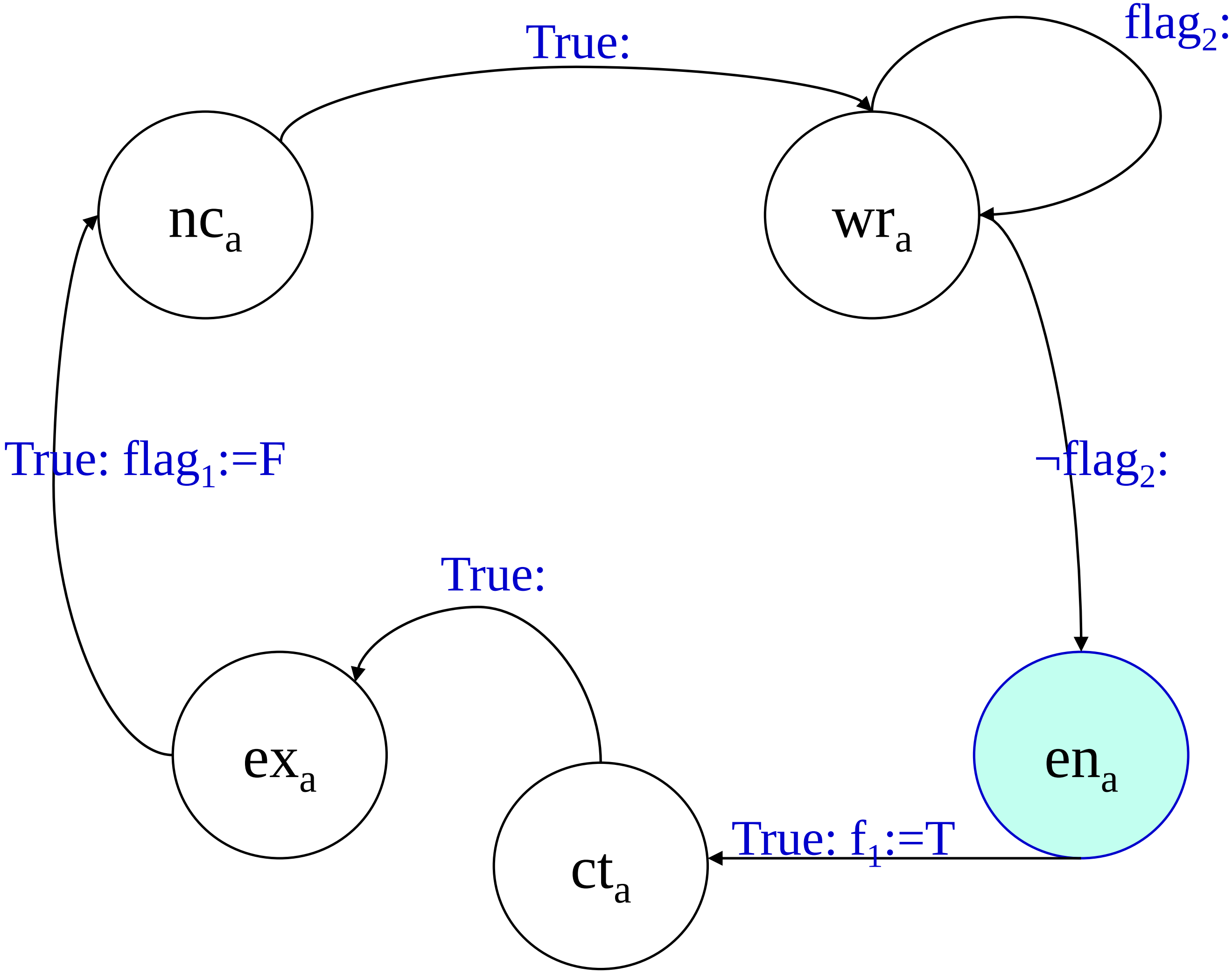


no more deadlock

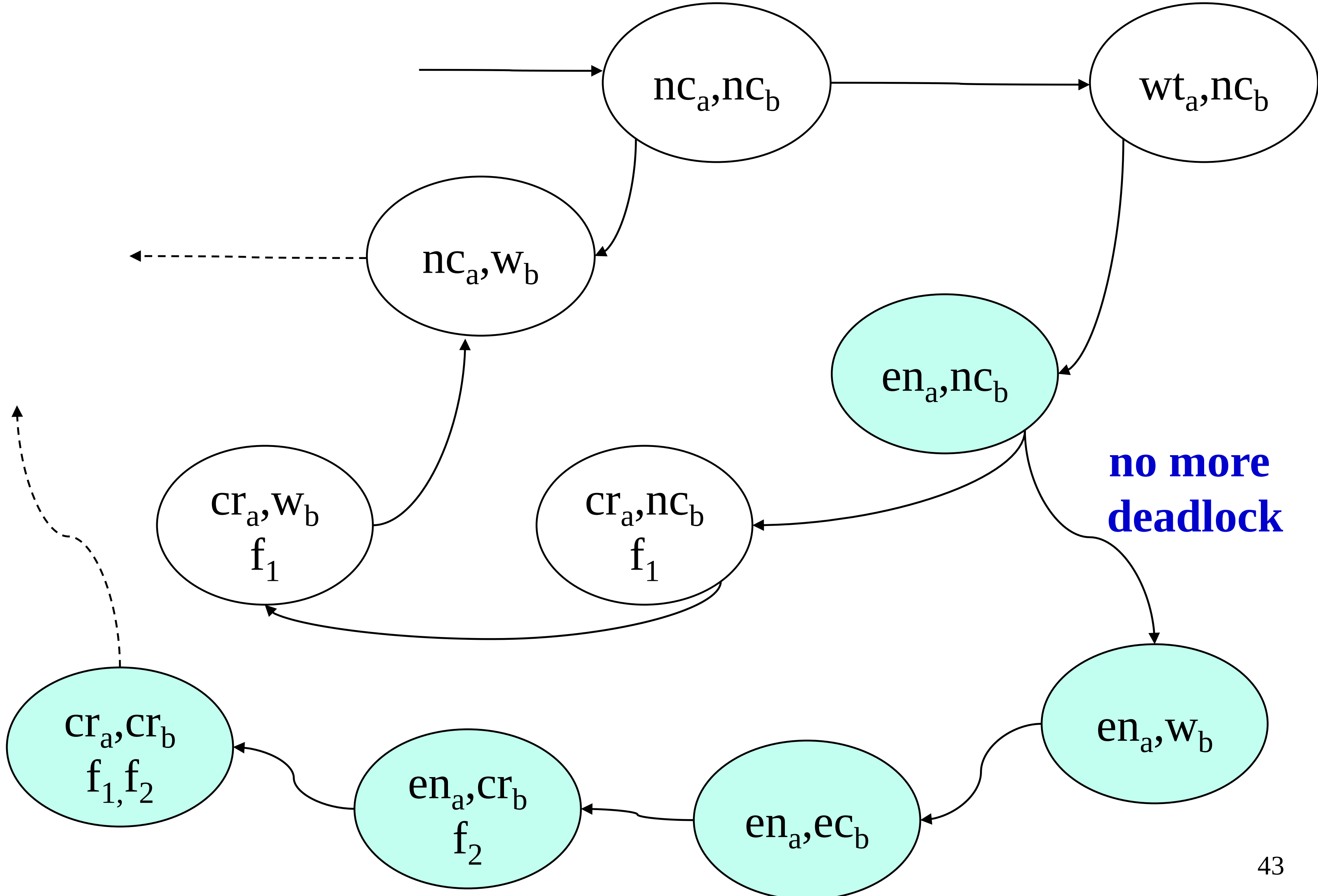
Non atomicity in Process A



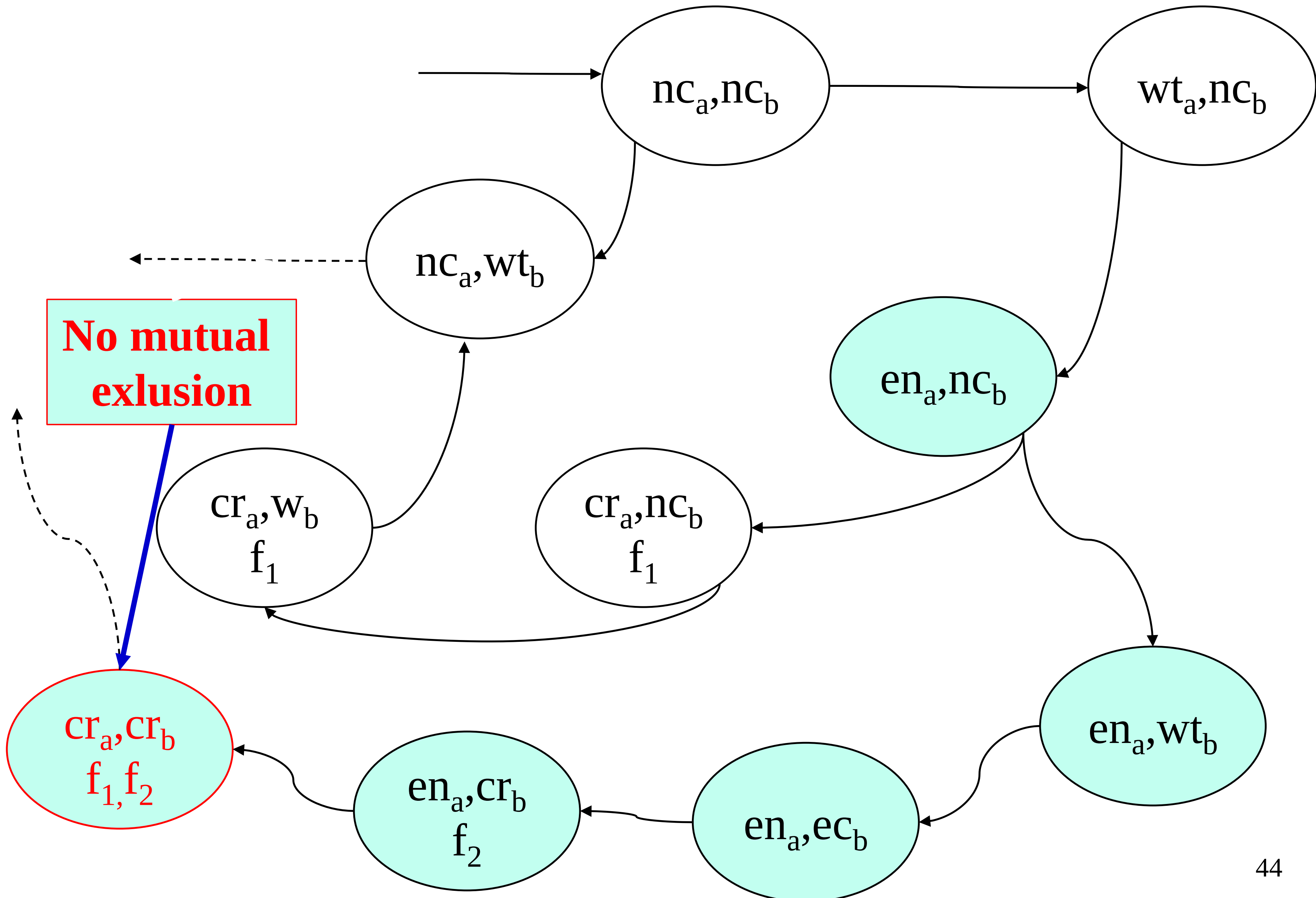
A more adequate automaton for Proc. A



Composition: LTS (fragment)



Composition: LTS (fragment)



Synchronization via handshake

- Let TS_1 and TS_2 be 2 TSs, with $TS_i = <S_i,A_i,R_i,S_{i0}>$
- Let $Sync = A_1 \cap A_2$ be the synchronization actions
- The system model is the cartesian product of the simpler modules with an additional null action **-**
- Then $TS = TS_1 \parallel TS_2 = <S,A,R,S_0>$ is s.t.
  - $S = S_1 \times S_2$
  - $A \subset Sync \cup ((A_1 \cup \{-\} \setminus Sync) \times (A_2 \cup \{-\} \setminus Sync))$ s.t.
    - $Sync \subseteq A$ and $<a_1, a_2> \in A$ iff either $a_1 = -$ or $a_2 = -$
  - $S_0 = S_{10} \times S_{20}$
  - R contains $<s_1,s_1> \xrightarrow{a} <s_1', s_2'>$, if $a \in Sync$ and $s_i \xrightarrow{a}_i s_i'$, for $i \in \{1,2\}$, or $a = <a_1, a_2> \in A$ and $s_i \xrightarrow{a_i}_i s_i'$, if $a_i \neq -$, and $s_i' = s_i$, otherwise

Communication via channels

- A ***channel*** c is a fifo buffer of some capacity $cap(c) \geq 0$
- When $cap(c) = 0$ communication is synchronous
- With each channel c a domain dom(c) is associated
- Two processes modeled as program graphs on $Var = Var_1 \cup Var_2$ and channels Chan communicate by means of communication actions of the form

$$c?x \text{ and } c!d$$

where c is a channel in Chan, x a variable in Var and d a value in D

–c?x stands for a receive of a value from channel c, which is then assigned to variable x.

–c!d stands for a send of value d over channel c

- The set of communication actions is defined as:

$$\boxed{Com = \{c!d, c?x \mid c \in Chan,\ d \in D,\ x \in Var \text{ and } dom(c) \subseteq dom(x)\}}$$

Communication via channels

- A Program Graph over (Var,Chan) is

$$PG_i = (Loc_i, Act_i, Effect_i, \hookrightarrow_i, Loc_{i0}, g_{i0})$$

is a program graph on Var_i such that $\hookrightarrow_i$ is defined as

$$\boxed{\hookrightarrow_i \subseteq Loc_i \times Cond(Var_i) \times (Act_i \cup Com) \times Loc_i}$$

- A ***Channel System*** CS over (Var,Chan) is a composition

$$\boxed{CS = PG_1 \mid \ldots \mid PG_n}$$

of program graphs PG_i over $(Var_i, Chan)$, where we take $Var = \cup_{1 \leq i \leq n} Var_i$.

Effects of channel actions

- If $cap(c) = 0$ then
  - transition $l_i \overset{c!d}{\hookrightarrow}_i l_i'$ is executable by process P_i only if transition $l_j \overset{c?x}{\hookrightarrow}_j l_j'$ is executable by process P_j , and
  - the two transitions occur *simultaneously* and $x=d$.
- If $cap(c) > 0$ then
  - transition $l_i \overset{c!d}{\hookrightarrow}_i l_i'$ is executable by process P_i only if channel c is *not full*, i.e. it contains less than $cap(c)$ messages: $\boxed{c=\langle m_1,\ldots,m_k\rangle \Rightarrow c=\langle m_1,\ldots,m_k,d\rangle}$.
  - transition $l_j \overset{c?x}{\hookrightarrow}_j l_j'$ is executable by process P_j only if channel c is *not empty*, i.e. it contains at least one message. The first value in c is assigned to variable x: $\boxed{c=\langle m_1,\ldots,m_k\rangle \Rightarrow c=\langle m_2,\ldots,m_k\rangle \text{ and } x=m_1}$.

Transition system for Channel System

- The resulting transition system TS(CS) has states of the form

$$<l_1,\ldots,l_n,\eta,\chi>$$

where is the evaluation function for channels and assigns to each channel with cap(c) > 0 its content (a sequence of messages), i.e., χ: Chan $\rightarrow$ dom(c)* $\in$ Eval(Chan).

- For the initial states, the locations are the initial locations of the processes, $\eta \models g_{i0}$ and $\chi(c) = \varepsilon$ (i.e., the empty sequence), for each $c \in$ Chan.
- Let χ_0 denote such initial channel evaluation function.

Transition system for Channel System

- TS(CS) is defined as follows:

$$(S, Act, \rightarrow, S_0, AP, L)$$

  - $S = (Loc_1 \times \ldots \times Loc_n) \times Eval(Var) \times Eval(Chan)$
  - $Act = \biguplus_{1 \le i \le n} Act_i \uplus \{\tau\}$ (τ denotes an internal action)
  - $S_0 = \{<l_1, \ldots, l_n, \eta, \chi> \mid l_i \in Loc_{i0}, \eta \models g_{i0}$ and $\chi = \chi_0\}$
  - $AP = \biguplus_{1 \le i \le n} Loc_i \uplus Cond(Var)$
  - $L(<l_1, \ldots, l_n, \eta, \chi>) = \{l_1, \ldots, l_n\} \cup \{g \in Cond(Var) \mid \eta \models g\}$
  - and the transition relation $\rightarrow$ is defined as follows:

Transition system for Channel System

If $l_i \overset{g:a}{\hookrightarrow}_i l_i'$, $a \in Act_i$, and $\eta \models g$ then

$$<l_1,\ldots,l_i,\ldots,l_n,\eta,\chi> \xrightarrow{a} <l_1,\ldots,l_i',\ldots,l_n,\eta',\chi>$$

where $\eta' = Effect(a,\eta)$

If $l_i \overset{g:c!d}{\hookrightarrow}_i l_i'$, $\eta \models g$, $len(c)=k<cap(c)$ and $\chi(c) = d_1,\ldots,d_k$ then

$$<l_1,\ldots,l_i,\ldots,l_n,\eta,\chi> \xrightarrow{\tau} <l_1,\ldots,l_i',\ldots,l_n,\eta,\chi'>$$

where $\chi' = \chi[c := d_1,\ldots,d_k,d]$.

If $l_i \overset{g:c?x}{\hookrightarrow}_i l_i'$, $\eta \models g$, $len(c)=k> 0$ and $\chi(c) = d_1,\ldots,d_k$ then

$$<l_1,\ldots,l_i,\ldots,l_n,\eta,\chi> \xrightarrow{\tau} <l_1,\ldots,l_i',\ldots,l_n,\eta',\chi'>$$

where $\eta' = \eta[x := d_1]$ and $\chi' = \chi[c := d_2,\ldots,d_k]$.

If $l_i \overset{g_1:c?x}{\hookrightarrow}_i l_i'$, $l_j \overset{g_2:c!d}{\hookrightarrow}_j l_j'$, $\eta \models g_1 \wedge g_2$, $cap(c)=0$ and $i \neq j$ then

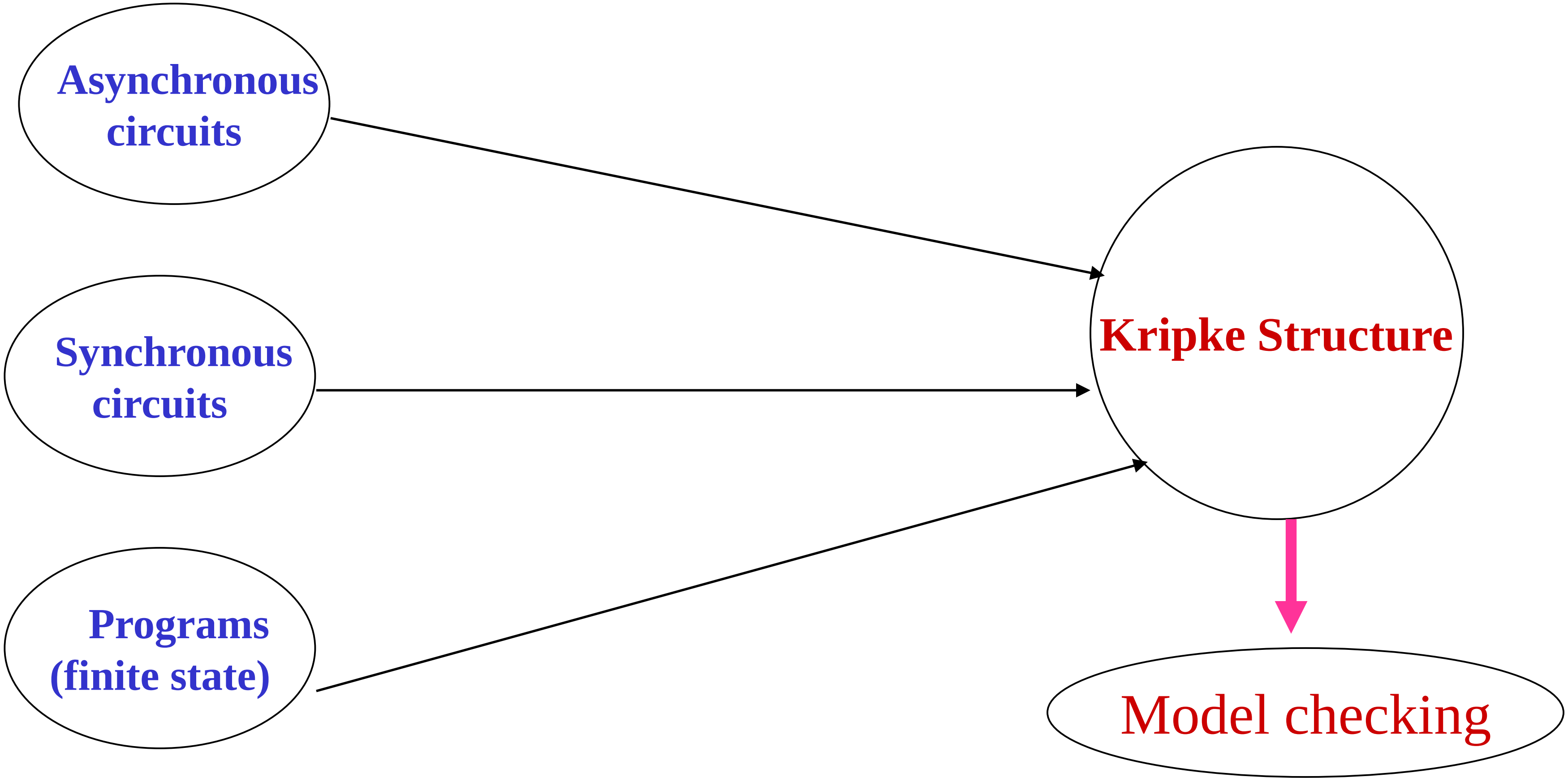
$$<l_1,\ldots, l_i,\ldots l_j,\ldots,l_n,\eta,\chi> \xrightarrow{\tau} <l_1,\ldots,l_i', \ldots l_j',\ldots,l_n,\eta',\chi>$$

where $\eta' = \eta[x := d]$.

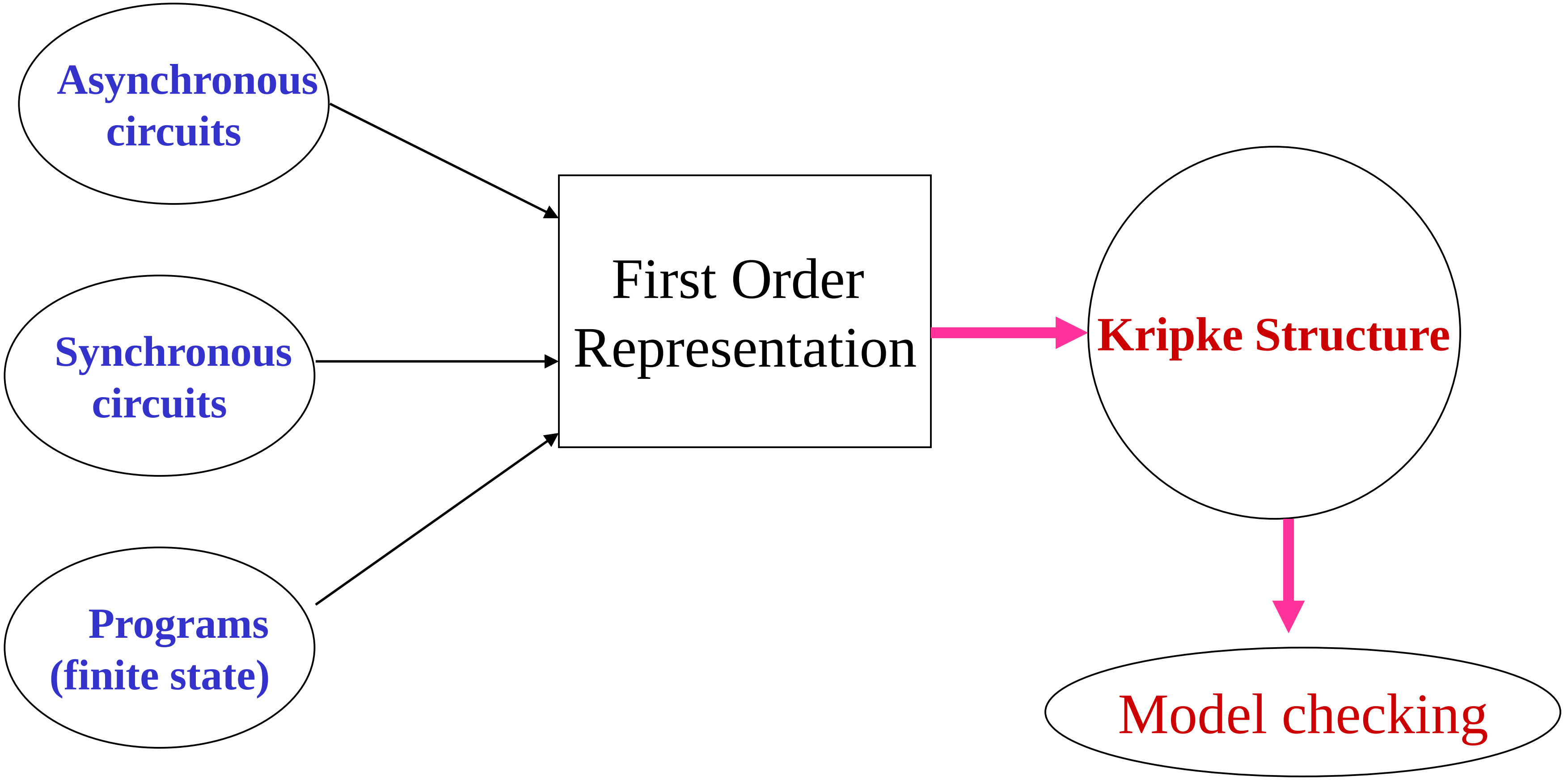
The common framework

- Many systems need to be modeled.
  - Digital circuits
    - **Synchronous**
    - **Asynchronous**
  - Programs
- Strategy : Capture the main features using a logical framework (nothing to do with temporal logics!) : ***First order representation***

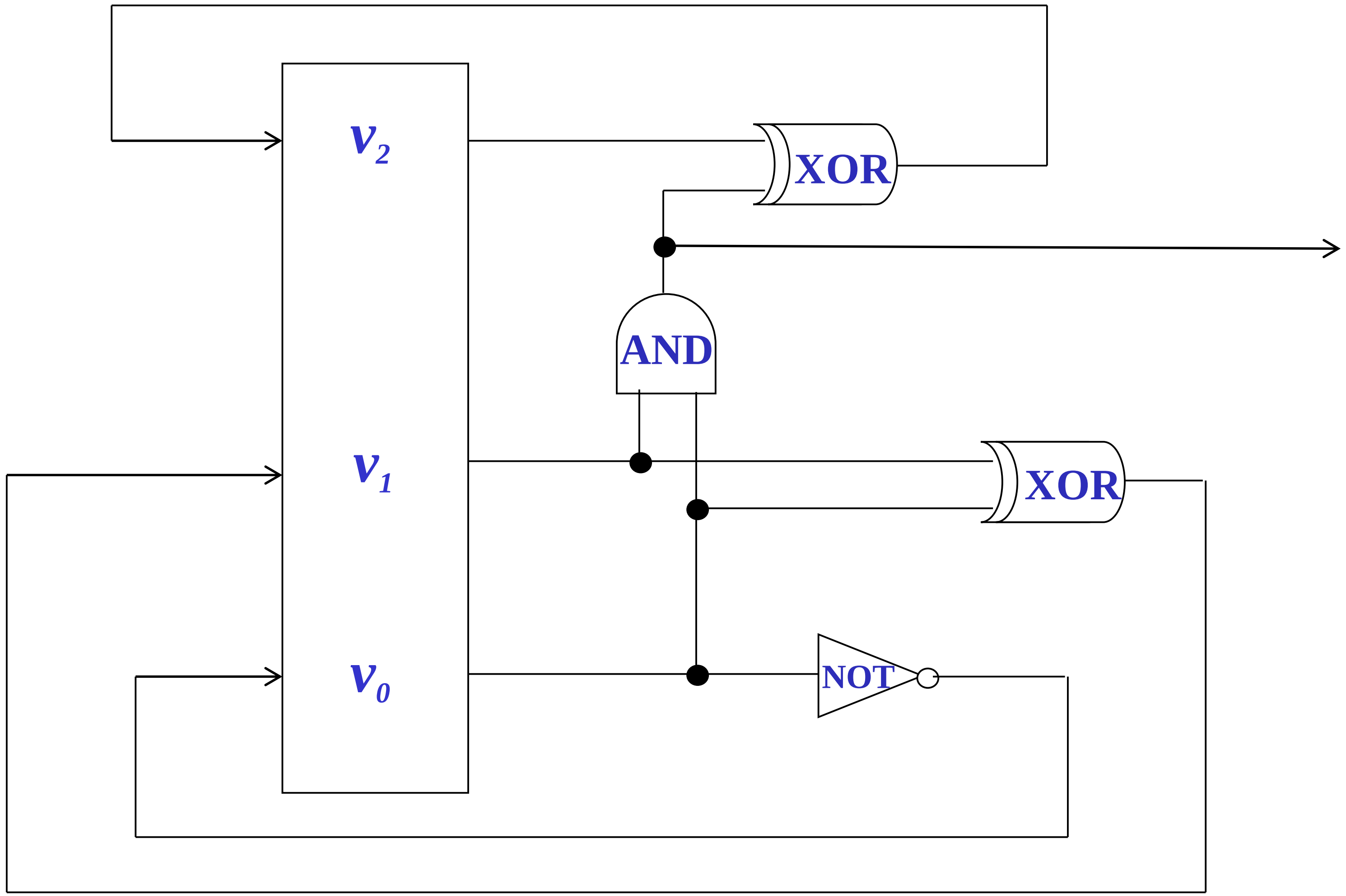
The inefficient way



The efficient way



Synchronous counter modulo 8

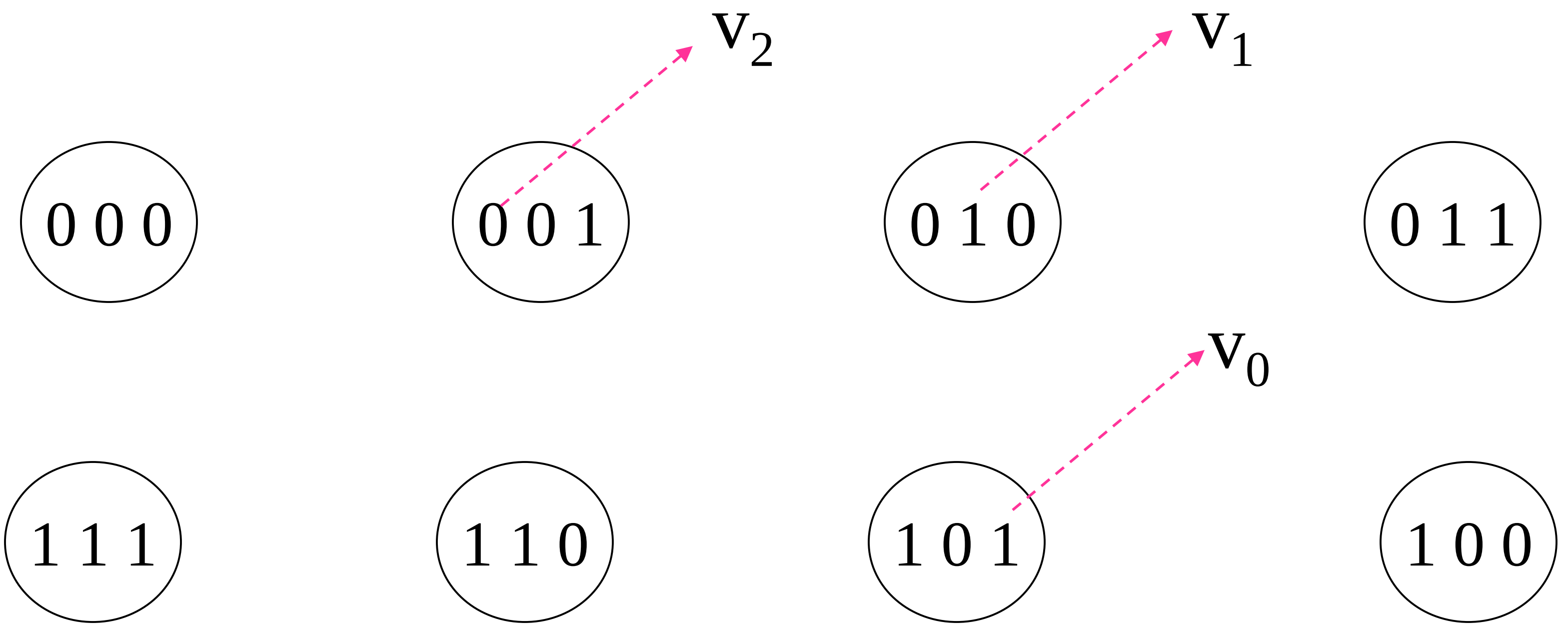


The mod-8 counter

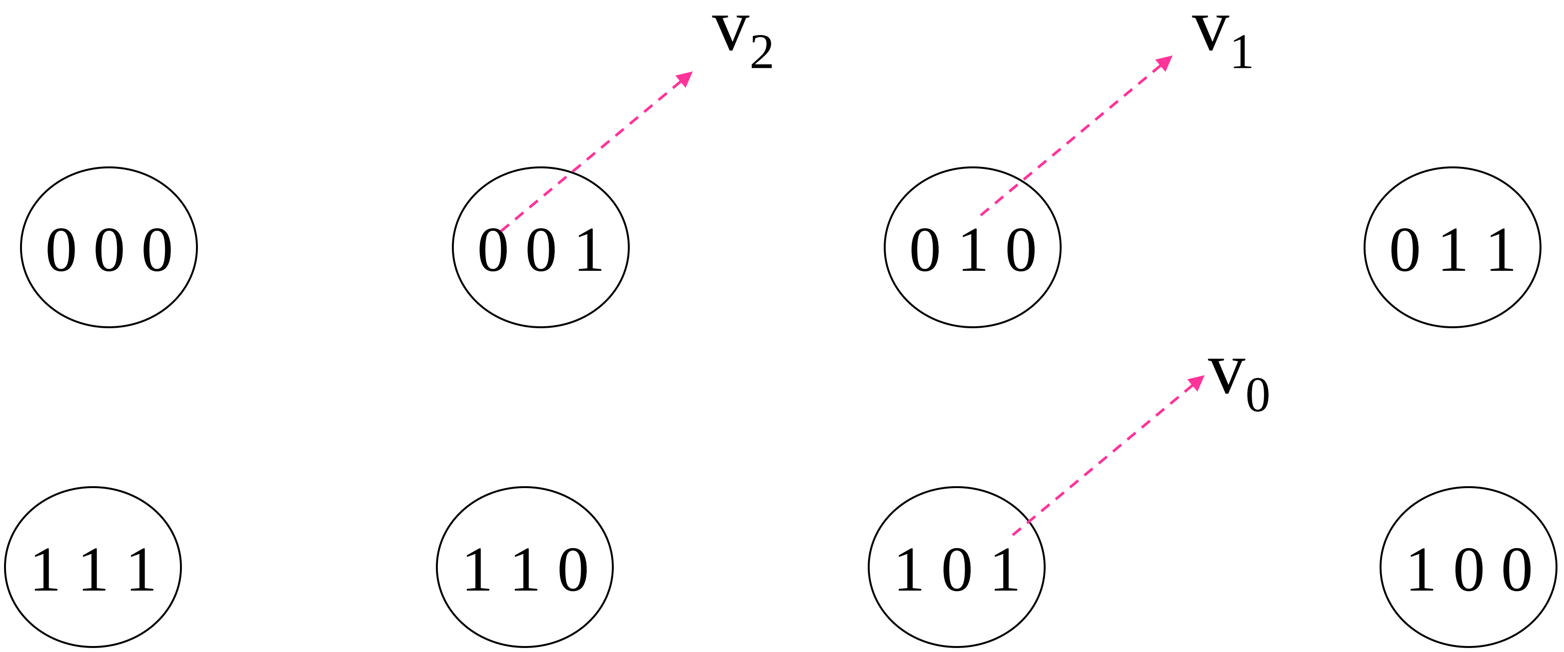
- System variables : $V = \{v_2\ v_1\ v_0\}$
- Domain of v_2 is $\{0, 1\}$

  Same domain for v_1 and v_0 as well.
- Special case : These variables are boolean
- Each state s can also be seen as a function assigning to each variable a value in its domain.
  - $\mathbf{s : V \rightarrow B}$
  - $\mathbf{s(v_0) = 0 \quad s(v_1) = 1 \quad s(v_2) = 1}$
  - This specifies the state **s** = (1 1 0) !

A mod-8 counter: states



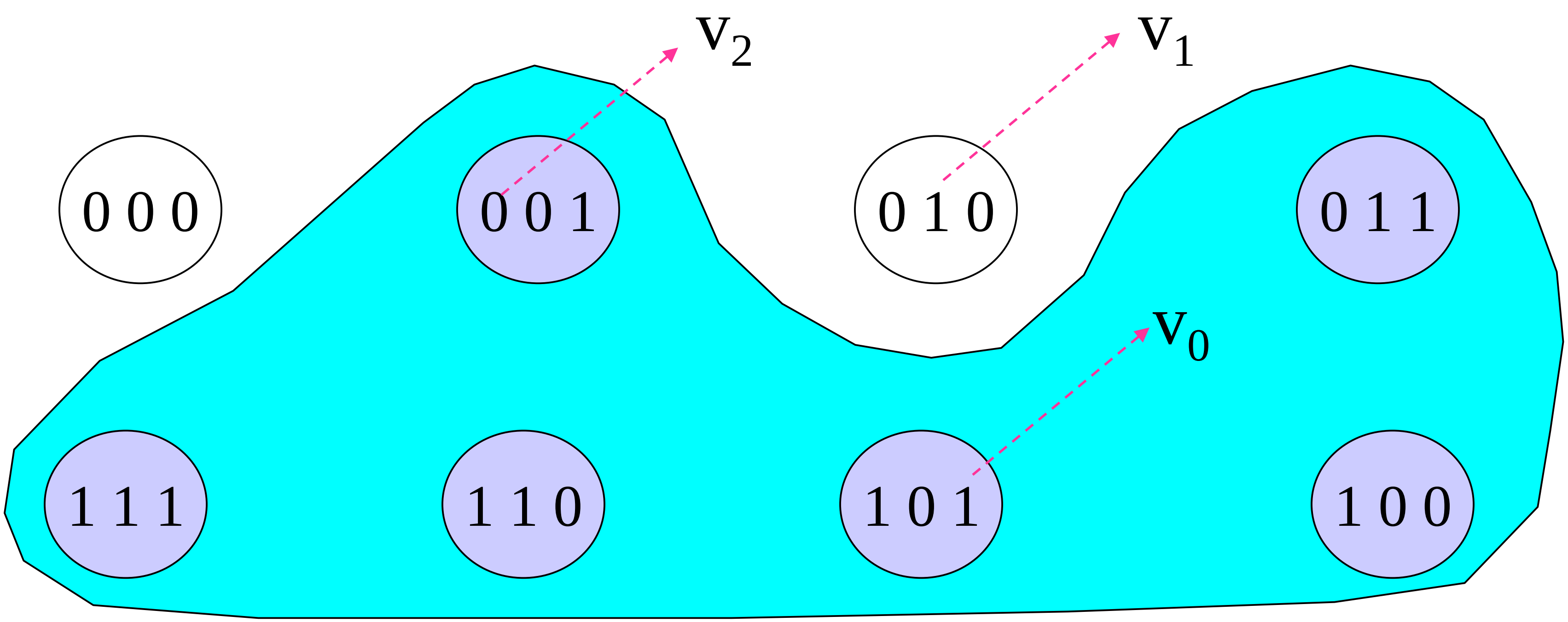
State Predicates



A set of states can be picked out by a propositional formula:

$\mathbf{X = v_2 \vee v_0}$ is the set {...}

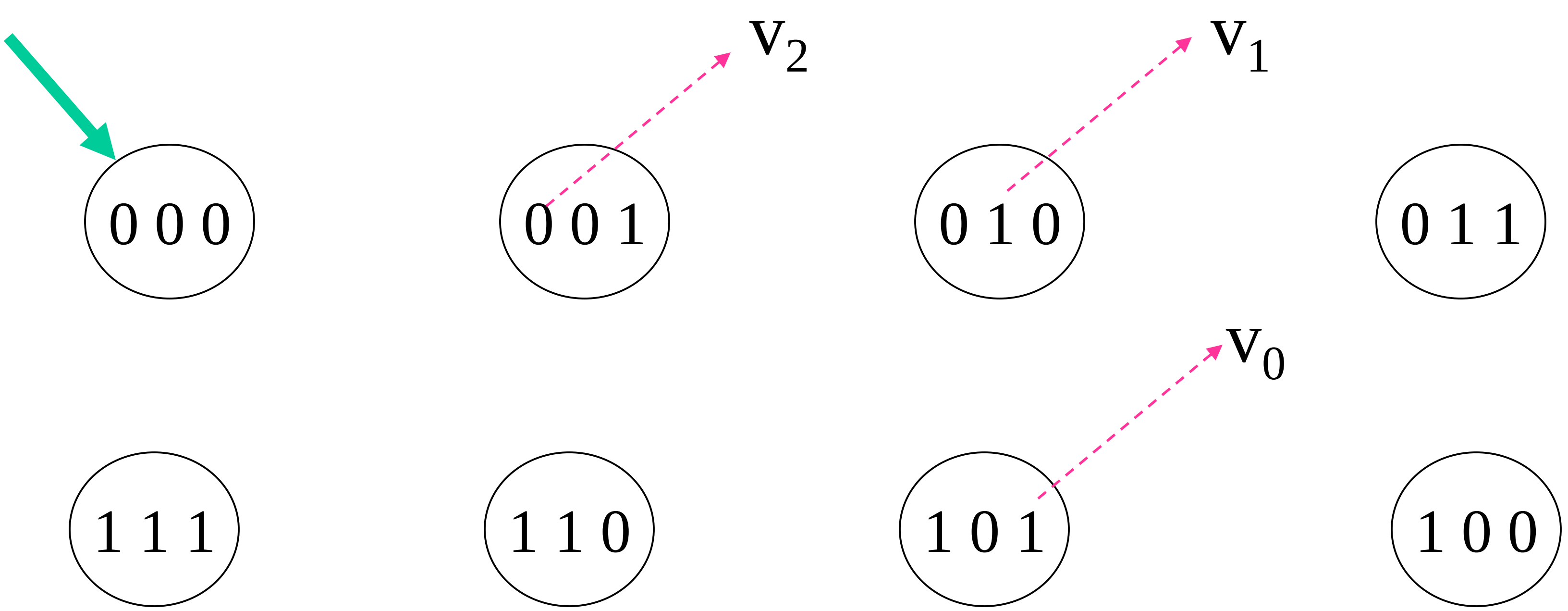
State Predicates



A set of states can be picked out by a propositional formula:

$\mathbf{X} = \mathbf{v_2} \vee \mathbf{v_0}$ is the set {100, 101, 110, 111, 001, 011}

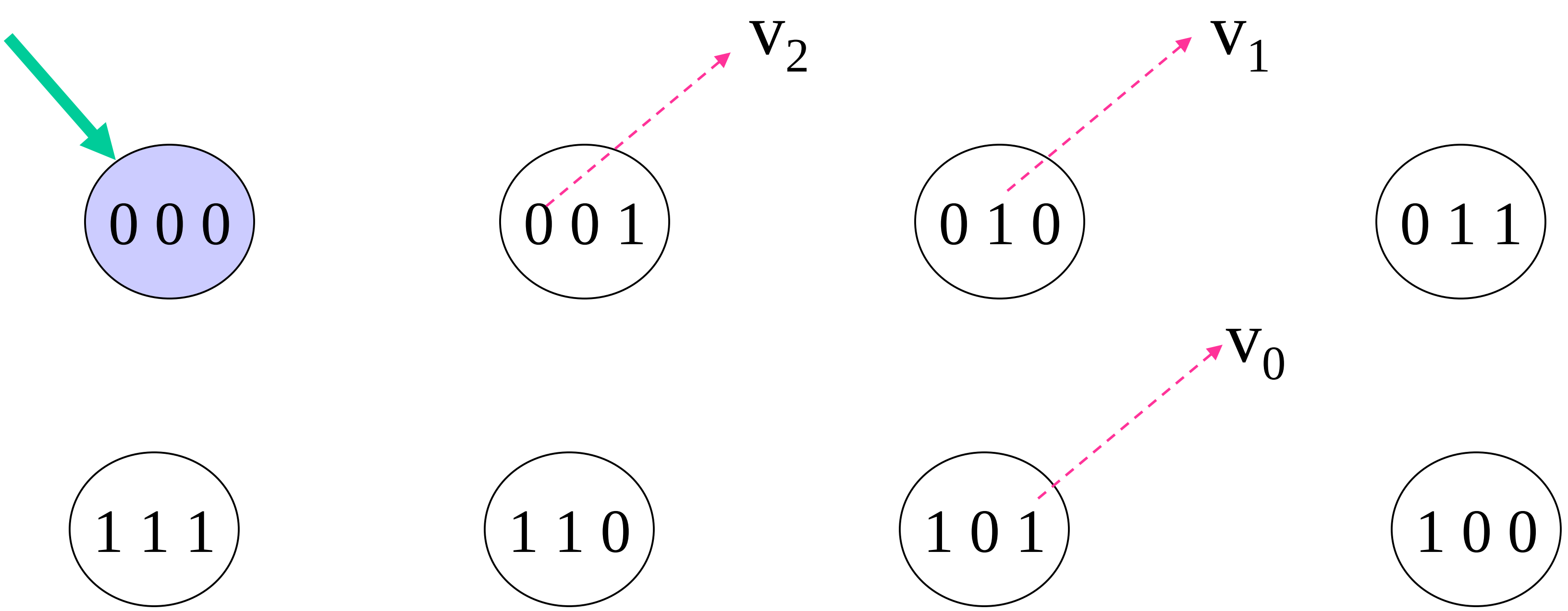
Initial States Predicate



A set of states can be picked out by a formula;

$\mathbf{S_0} = \neg\mathbf{v_2} \wedge \neg\mathbf{v_1} \wedge \neg\mathbf{v_0}$

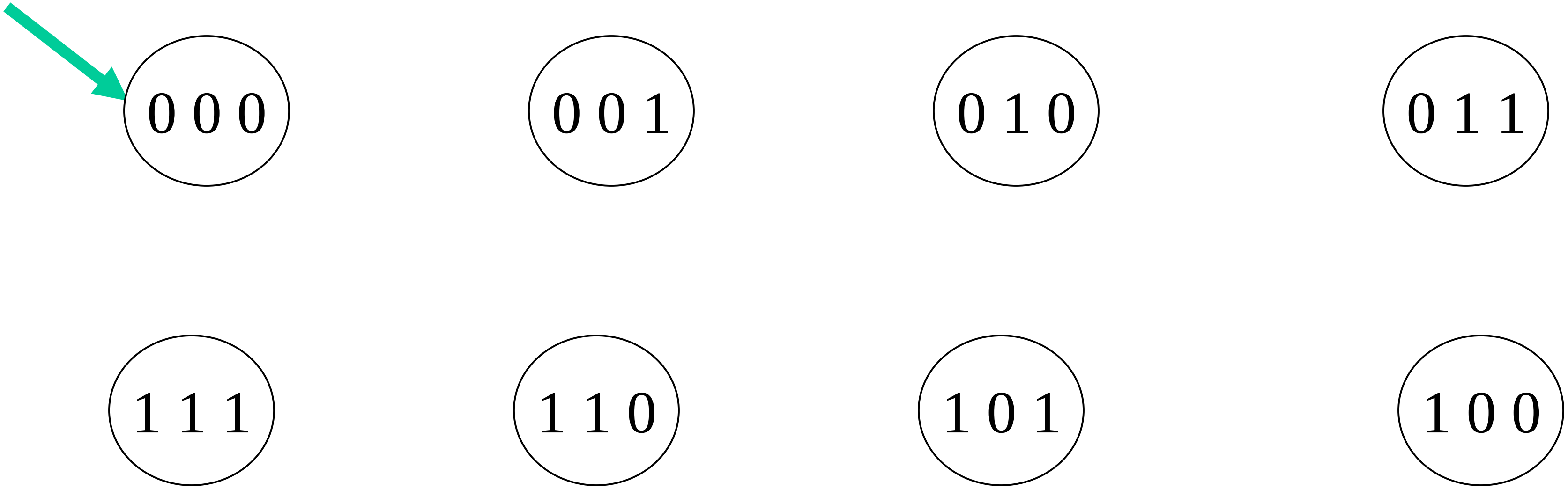
Initial States Predicate



A set of states can be picked out by a formula;

$\mathbf{S_0} = \neg \mathbf{v_2} \wedge \neg \mathbf{v_1} \wedge \neg \mathbf{v_0}$ therefore $\mathbf{X_1} = \{ \mathbf{S_0} \} = \{ 000 \}$

Transition relation predicate

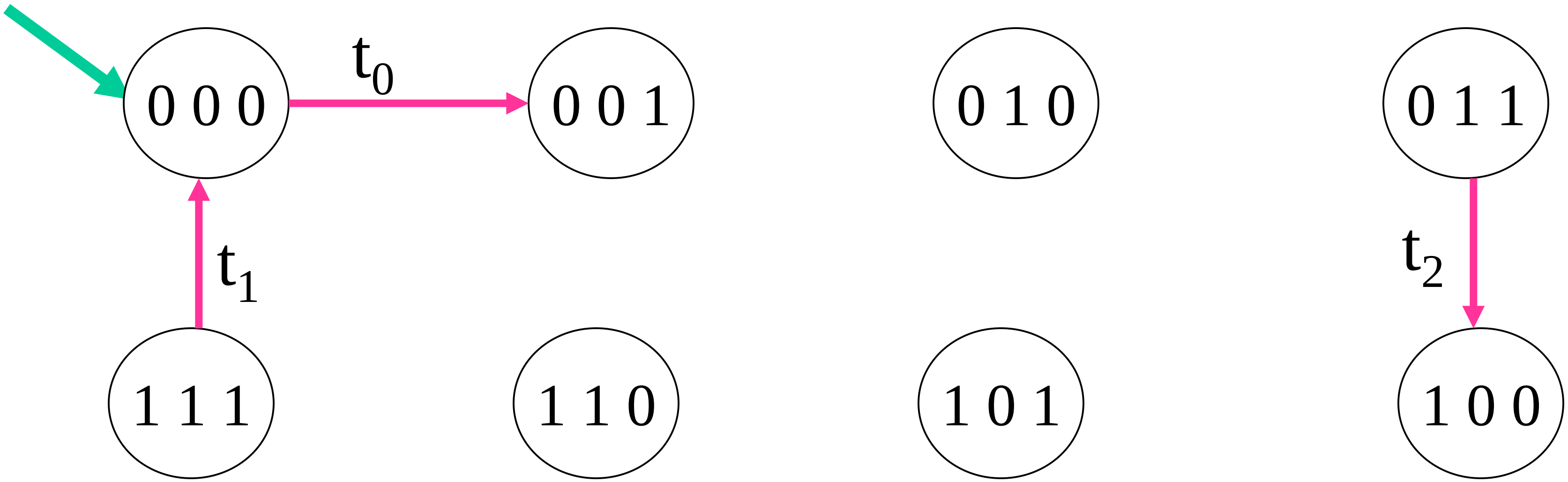


A set of *transitions* can also be picked out by a formula.

$$\mathbf{R_2} = \mathbf{v_2'} \Leftrightarrow (\mathbf{v_0} \wedge \mathbf{v_1}) \oplus \mathbf{v_2}$$

$\mathbf{v_2}$ – current value $\mathbf{v_2'}$ – next value

Transition relation predicate

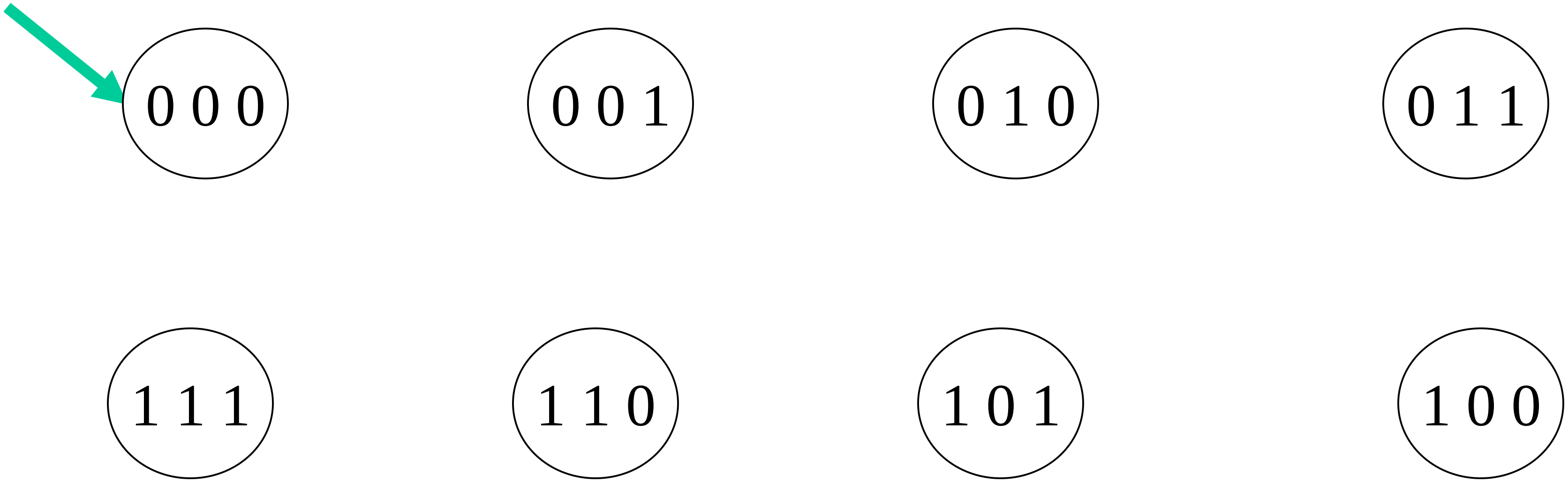


A set of transitions can also be picked out by a formula.

$\mathbf{R_2} = \quad \mathbf{v_2'} \Leftrightarrow (\mathbf{v_0} \wedge \mathbf{v_1}) \oplus \mathbf{v_2}$ $\quad\quad$ $\mathbf{v_2}$ – current value $\quad$ $\mathbf{v_2'}$ – next value

$\boxed{\{\mathbf{t_0}, \mathbf{t_1}, \mathbf{t_2}\} \subseteq \mathbf{R_2}}$

Transition relation predicate

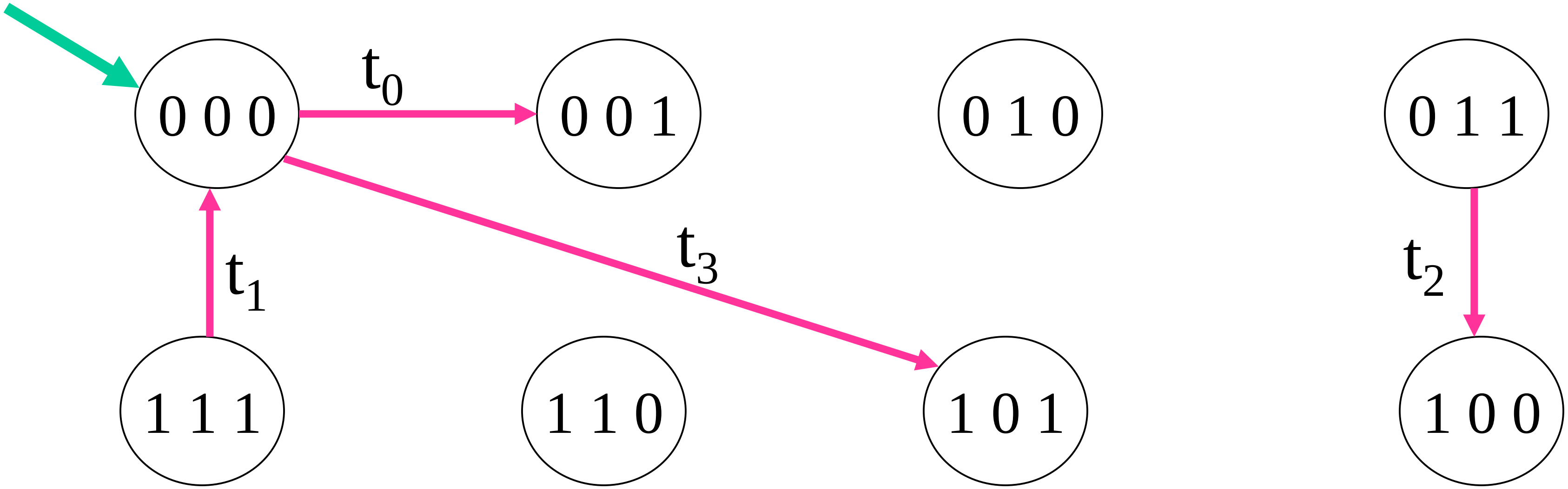


R = **Formula**($\mathbf{v_2}$, $\mathbf{v_1}$, $\mathbf{v_0}$, $\mathbf{v_2}$', $\mathbf{v_1}$', $\mathbf{v_0}$')

Not all formulae will define subsets of transitions.

You must pick the right formula .

Transition relation predicate



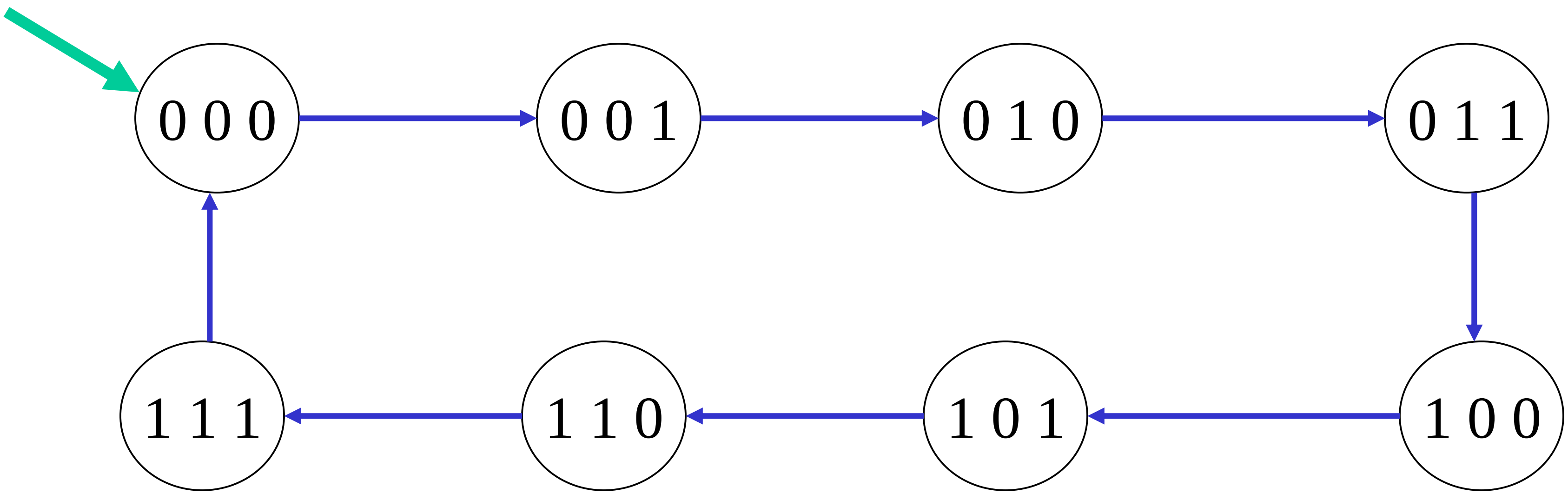
$\mathbf{R_0}$ = $\mathbf{v_0' \neq v_0}$ $\mathbf{v_0}$ – current value $\mathbf{v_0'}$ – next value

$\mathbf{R_0}$ = $\{(000) \longrightarrow (101)$,……..}

But this is not a transition!

$\mathbf{\{t_0, t_1, t_2, t_3\} \subseteq R_0}$ but $\mathbf{t_3 \not\in R_2}$

Transition relation predicate



$\mathbf{R_0} = \mathbf{v_0'} \neq \mathbf{v_0}$ $\mathbf{v_i}$ – current value $\mathbf{v_i'}$ – next value

$\mathbf{R_1} = \mathbf{v_1'} = (\mathbf{v_0} \oplus \mathbf{v_1})$

$\mathbf{R_2} = \mathbf{v_2'} = (\mathbf{v_0} \wedge \mathbf{v_1}) \oplus \mathbf{v_2}$

$\mathbf{R} = \mathbf{R_0} \wedge \mathbf{R_1} \wedge \mathbf{R_2}$

Symbolic Representation of Transition Systems

- $\{v_1, v_2, \ldots, v_n\}$--- System variables.
- $D_1, D_2, \ldots, D_n$ --- The corresponding domains.
- $D = \bigcup D_i$
- $s : \{v_1, v_2, \ldots, v_n\} \longrightarrow D$ such that
  $s(v_1) \in D_1 \ldots..$
- **S** --- The set of states.

Initial States

- $S_0(v_1, v_2, \ldots, v_n)$ is a FO formula describing the set of initial states.
- Atomic formula
  - **$v = d$** where v is is a system variable and d is a constant symbol interpreted as a member of the domain of v.

Example:

- "$\mathbf{S_0}$ is the set of all states where the **pc = 0** and ***input*** is a power of 2"
- $(pc = 0) \wedge n \geq 0 \wedge (input = EXP(n))$

Transition relation

- $R(v_1,v_2,\ldots v_n,v_1',v_2',\ldots,v_n')$ is a FO formula involving the ***current variables*** $v_1,v_2,\ldots,v_n$ (the system variables) and the ***next variables*** v_1', $v_2',\ldots,v_n'$.

- $(d_1, d_2,\ldots,d_n) \longrightarrow (d_1', d_2',\ldots,d_n')$ iff $R(v_1,v_2,\ldots v_n,v_1',v_2',\ldots,v_n')$ is true under the valuation $v_1 = d_1,\ldots,v_n = d_n$, $v_1' = d_1',\ldots v_n' = d_n'$.

Synchronization: no interaction

The system model is just the ***cartesian product*** of the simpler modules.

Let $TS_1,\ldots,TS_n$ be n automata (or **TSs**), where
$TS_i = \langle S_i, A_i, R_i, s_{i0}\rangle$

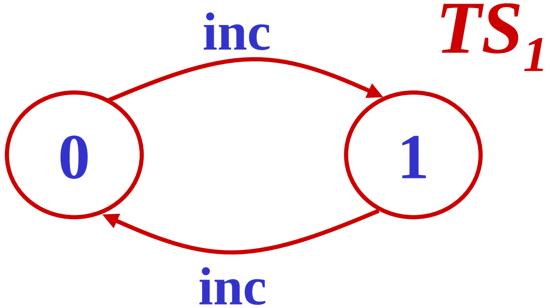
The system is then defined as $TS=\langle S,A,R,s_0\rangle$ where

$$S = S_1 \times S_2 \times \ldots \times S_n$$

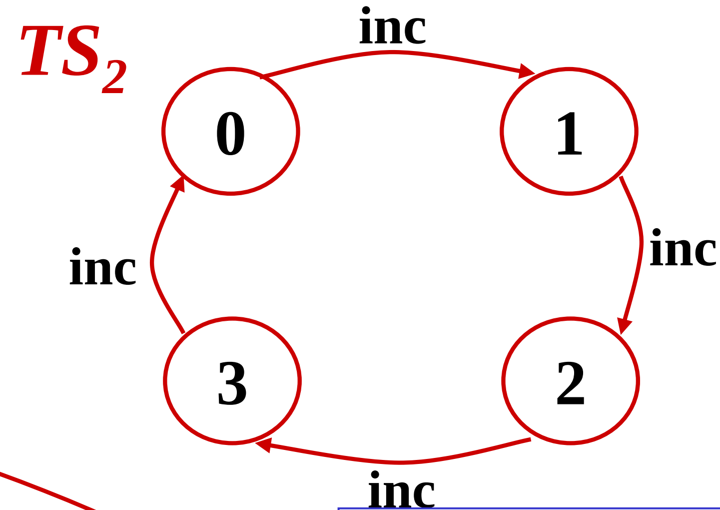
$$A = A_1 \cup \{-\} \times A_2 \cup \{-\} \times \ldots \times A_n \cup \{-\}$$

$$R = \{(\langle s_1,\ldots,s_n\rangle,\langle a_1,\ldots,a_n\rangle,\langle s'_1,\ldots,s'_n\rangle) \mid \textit{forall } i,\ a_i \neq - \textit{ and } (s_i,a_i,s'_i) \in R_i,\ \textit{or } a_i = - \textit{ and } s'_i = s_i\}$$

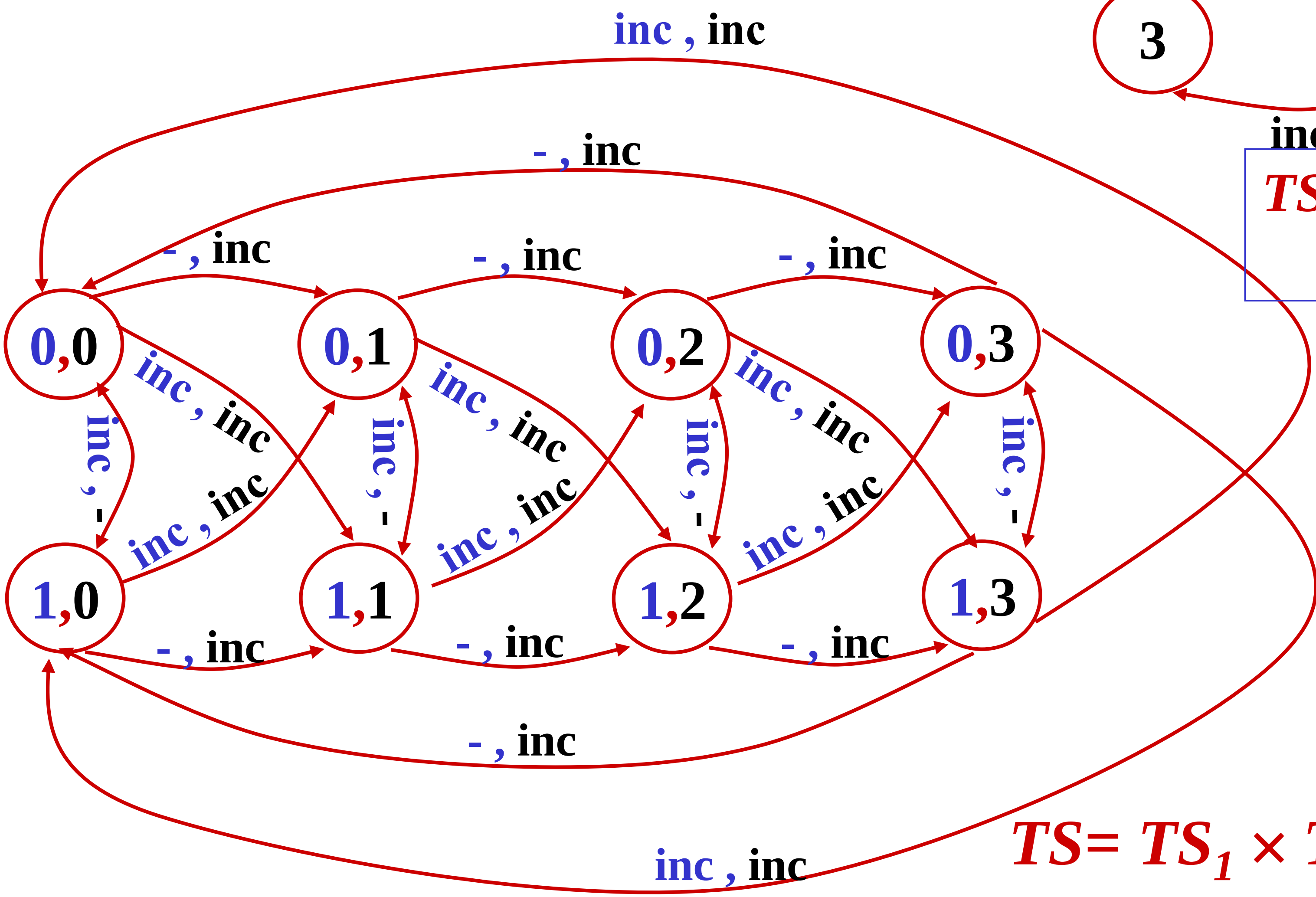
$$s_0 = \langle s_{10}, s_{20}, \ldots, s_{n0}\rangle$$



TS_1 counter
modulo 2



TS_2: counter
modulo 4



$TS = TS_1 \times TS_2$

Synchronization: interaction

To allow for interaction, or synchronization on specific actions we can introduce a Synchro-nization Set (to inhibit undesired transitions) :

- ***Synchronization set is just a subset of the composite actions:***

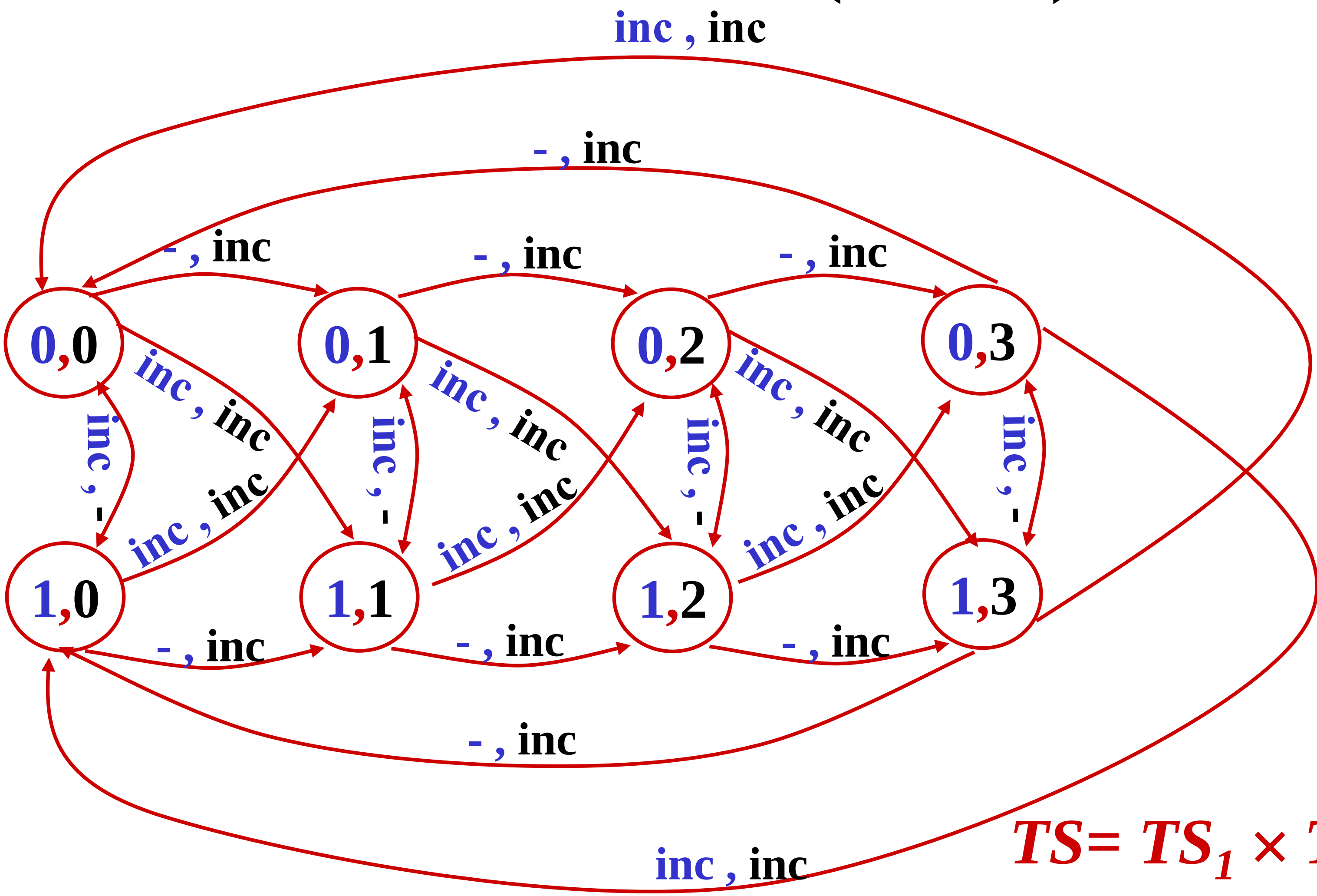
$$Sync \subseteq A_1 \cup \{-\} \times A_2 \cup \{-\} \times \ldots \times A_n \cup \{-\}$$

- ***Then we will have to define the possible transitions as:***

$$R = \{(\langle s_1,\ldots,s_n\rangle, \langle a_1,\ldots,a_n\rangle, \langle s'_1,\ldots,s'_n\rangle) \mid (a_1,\ldots,a_n) \in Sync \text{ and forall } i,\ a_i \neq - \text{ and } (s_i,a_i,s'_i) \in R_i, \text{ or } a_i = - \text{ and } s'_i = s_i\}$$

Free synchronization (Asynchronous systems):

$$Sync = \{inc, -\} \times \{-, inc\} = \{(-,-), (inc,-), (-,inc), (inc,inc)\}$$



$TS = TS_1 \times TS_2$

Free synchronization

Asynchronous systems:

$$\textit{Sync} = \{inc,-\} \times \{-,inc\}$$

$$R(V,V') = \bigwedge_{i\in I}\left(R_i(v_i,v_i') \vee v_i'=v_i\right)$$

Free synchronization

Asynchronous systems:

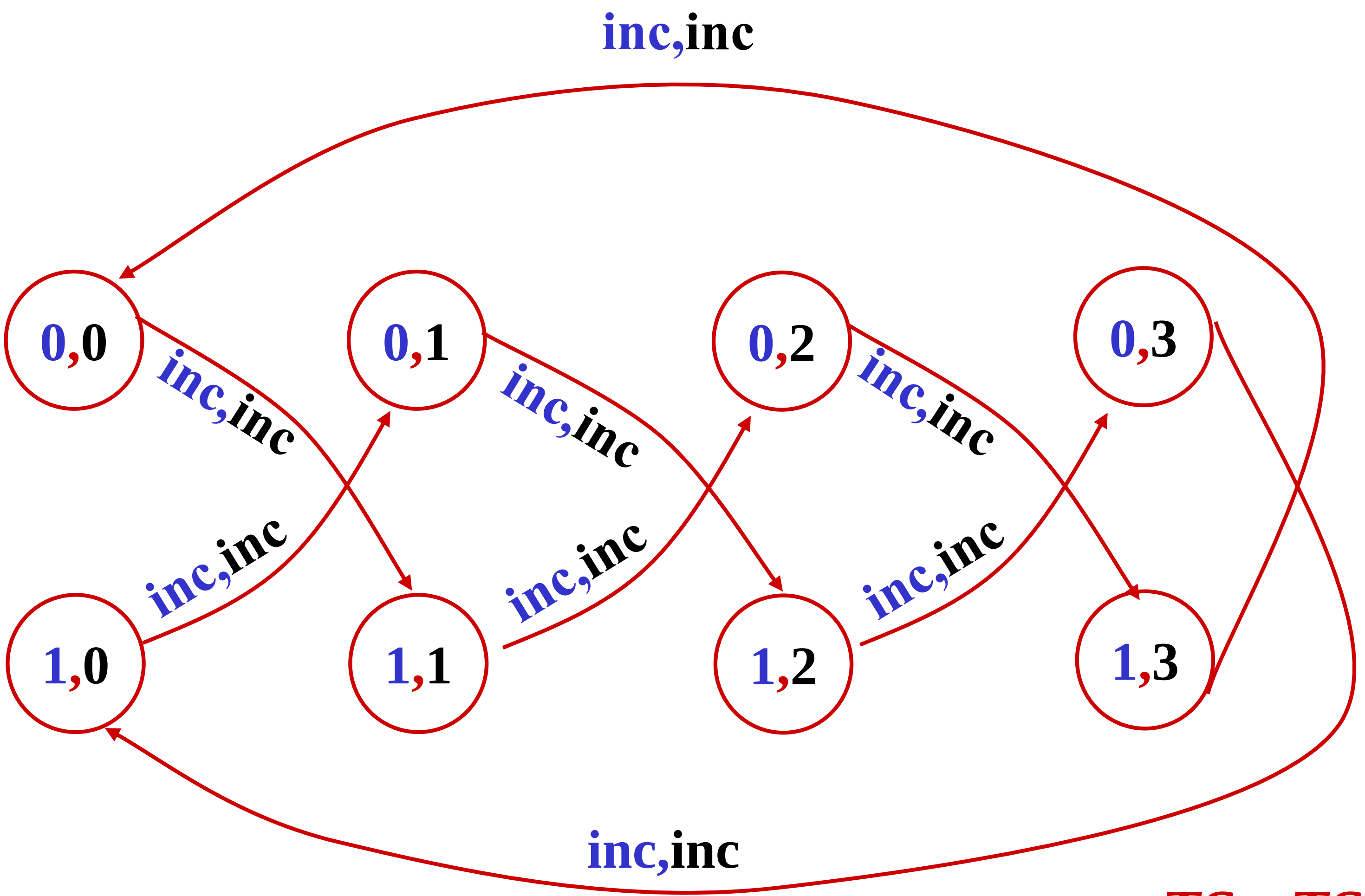
$Sync = \{inc,-\} \times \{-,inc\} \ \boxed{\setminus \ \{(-,-)\}}$

$$R(V,V') = \bigwedge_{i \in I}\Big(R_i(v_i,v_i') \vee v_i'=v_i\Big) \qquad \boxed{\wedge \neg \bigwedge_{i \in I} (v_i'=v_i)}$$

if one wants to *discard* the situation where *no components act*

Synchronization on all actions (Synchronous systems)

Sync = {(*inc*,*inc*)}



$TS = TS_1 \times TS_2$

Synchronous systems

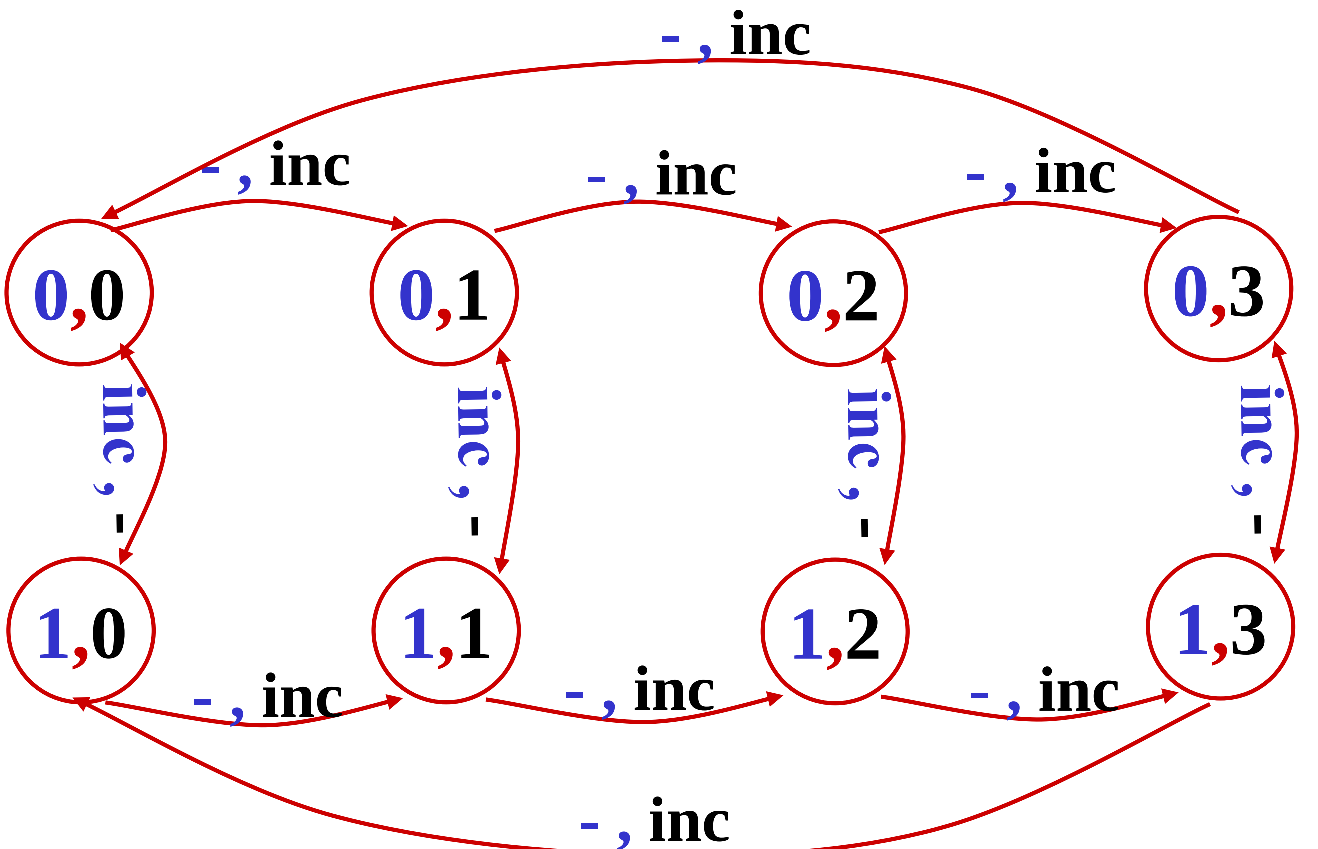
Synchronous systems:

***Sync* = {(*inc*,*inc*)}**

$$R(V,V') = \bigwedge_{i \in I} R_i(v_i,v_i')$$

Asynchronous systems with interleaving (only one component acts at any time):

$Sync = \{(-,inc),(inc,-)\}$



$TS = TS_1 \times TS_2$

Asynchronous systems: Interleaving

***Asynchronous systems**: only one component acts at any time.*

***Sync** = {(-,inc),(inc,-)}*

$$R(V,V') = \bigvee_{i \in I}\left(R_i(v_i,v_i') \wedge \bigwedge_{j \neq i} \mathrm{same}(v_j)\right)$$

Concurrent programs

- Many systems to be verified can be viewed as concurrent programs
  - operating system routines
  - cache protocols
  - communication protocols
- **P = cobegin ($P_1 \| P_2 \| \ldots \| P_n$) coend**
- **$P_1, P_2, .. P_n$ ---** Sequential Programs.
- ***Program variables*** set $\mathbf{V = V_1 \cup \ldots \cup V_n}$ (set $\mathbf{V_i}$ for program **i**)
- ***Program counters*** set **PC** (one for each program)
- ***Usually interleaving semantics is assumed***

Program Statements

A program P is a sequence of statements of the following form:

- skip
- v:= Expr (Expr an arithmetical expression)
- wait(Cond) (Cond an boolean expression)
- lock(v) (v a varible: semaphore)
- unlock(v) (v a varible: semaphore)
- $Statm_1$; $Statm_2$; … ; $Statm_n$ (sequential composition)
- IF Cond THEN $Statm_1$ ELSE $Statm_2$ ENDIF
- WHILE Cond DO Statm DONE
- COBEGIN (P_1 || P_2 || …|| P_n) COEND

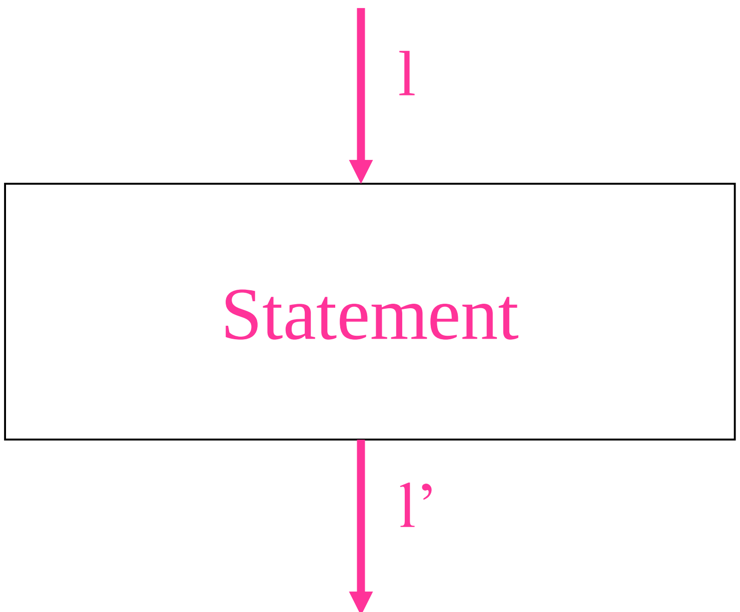
Transition relation of a program

- ***$R(v_1,v_2,...v_n,v_1',v_2',...,v_n')$*** is a formula involving the ***current variables*** $v_1,v_2,...,v_n$ (the system variables) and the ***next variables*** $(v_1', v_2',...,v_n')$.

- **$(d_1, d_2,..,d_n) \longrightarrow (d_1', d_2',..,d_n')$** iff ***$R(v_1,v_2,...v_n,v_1',v_2',...,v_n')$*** is true under the valuation **$v_1 = d_1,...,v_n = d_n, v_1' = d_1',..v_n' = d_n'$**.

Sequential Programs: the transition predicate C

General Structure

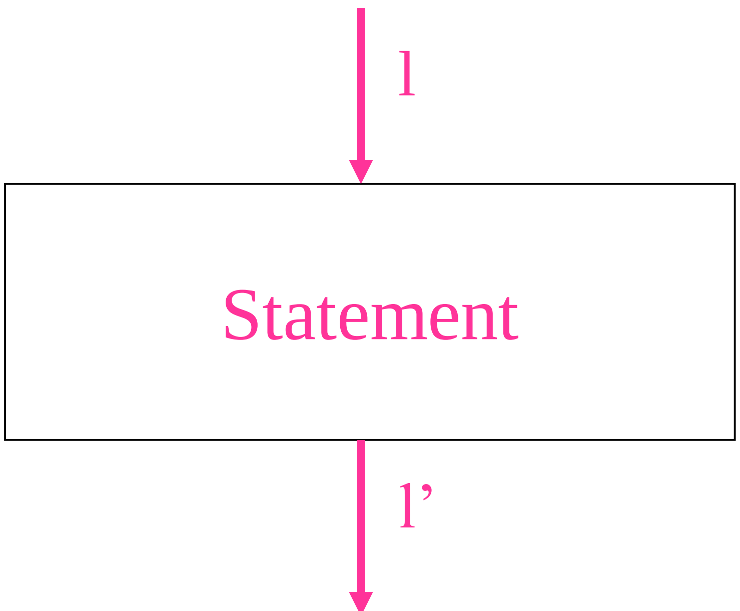


C(l, statement, l’)

C is essentially a *translation function* taking a *label*, a *program statement* and a *label* and giving the *FOL* formula specifying the transition relation induced by the statement.

Sequential Programs: the transition predicate C

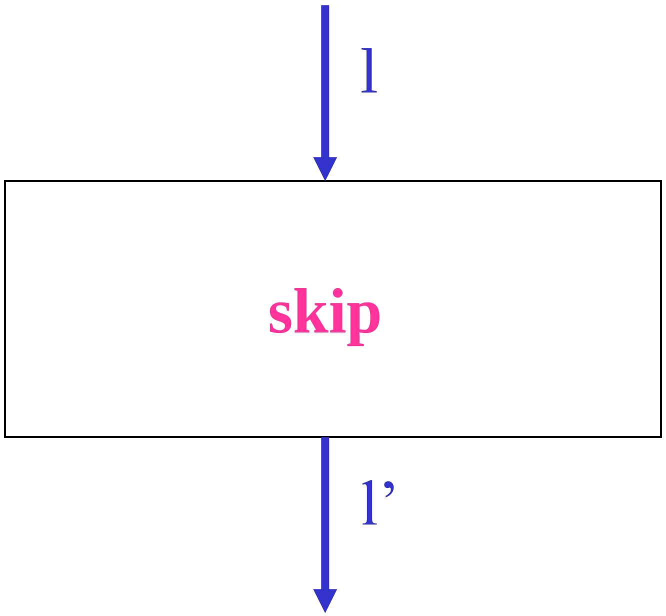
General Structure



C(l, statement, l')

$s \rightarrow s'$ is a transition of P iff the pair (s,s') satisfies the formula $C(l_{init}, P, l_{end})$, where l_{init} and l_{end} are the initial and final nodes of the CFG of P.

Skip

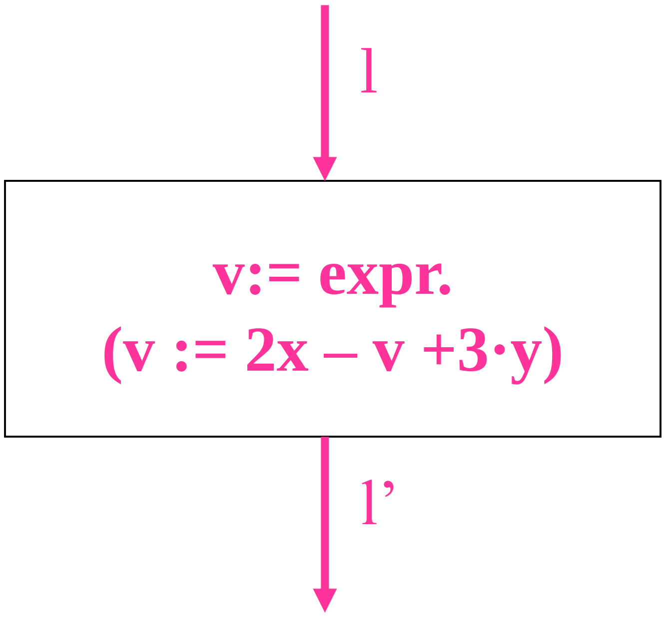


C(l, skip, l')

$$pc = l \wedge pc' = l' \wedge \text{same}(V) \wedge \text{same}(PC - \{pc\})$$

[for $Y = \{y_1, y_2, \ldots, y_m\}$,
$\text{same}(Y) \stackrel{\text{def}}{=} y_1' = y_1 \wedge y_2' = y_2 \wedge y_m' = y_m$ **]**

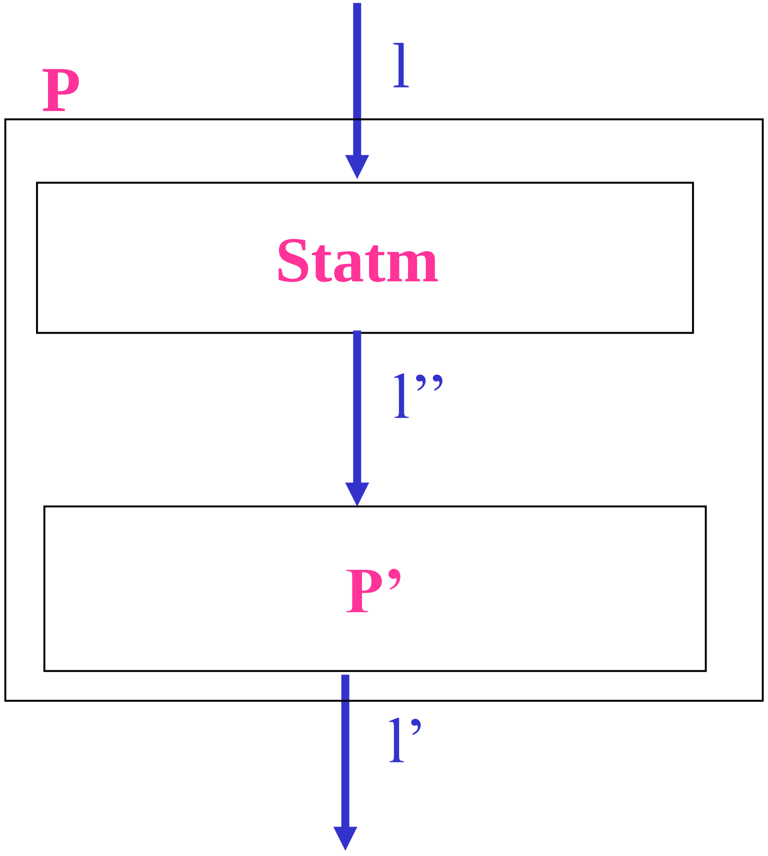
Assignments



C(l, v:=expr., l’)

$$pc = l \wedge pc' = l' \wedge v' = expr. \wedge$$
$$\wedge\ same\ (V - \{v\}) \wedge same\ (PC - \{pc\})$$

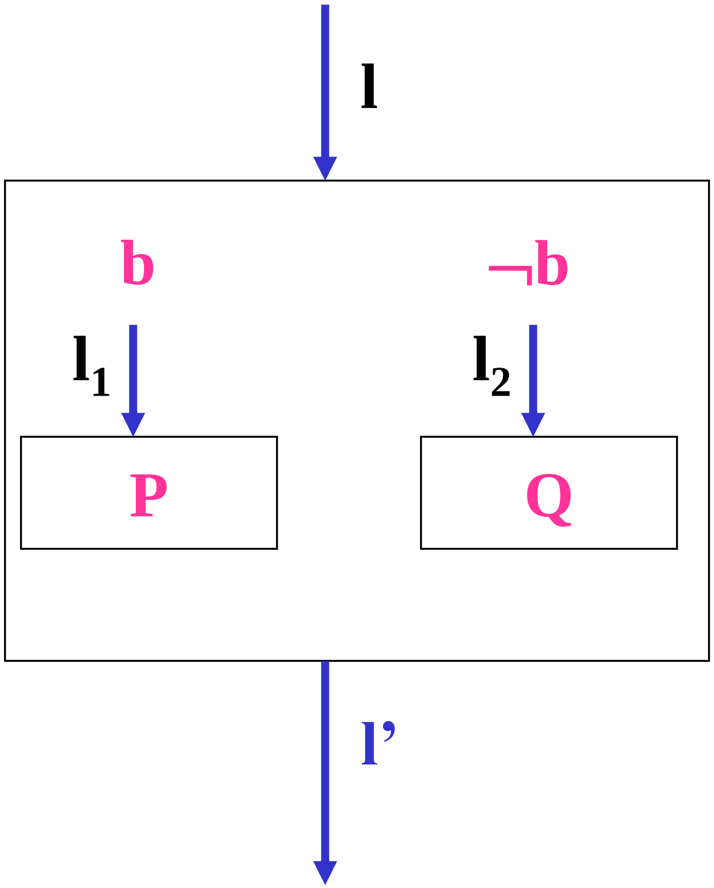
Sequential composition



$\boxed{C(l, \text{Statm} ; P', l')}$

$C(l, \text{Statm}, l'') \vee$
$C(l'', P', l')$

Conditional statement



$C(l, \text{IF-THEN-ELSE}(b,l_1,l_2), l')$

$(pc = l \wedge pc' = l_1 \wedge b \wedge same(V) \wedge$
$same(PC - \{pc\}) \vee$

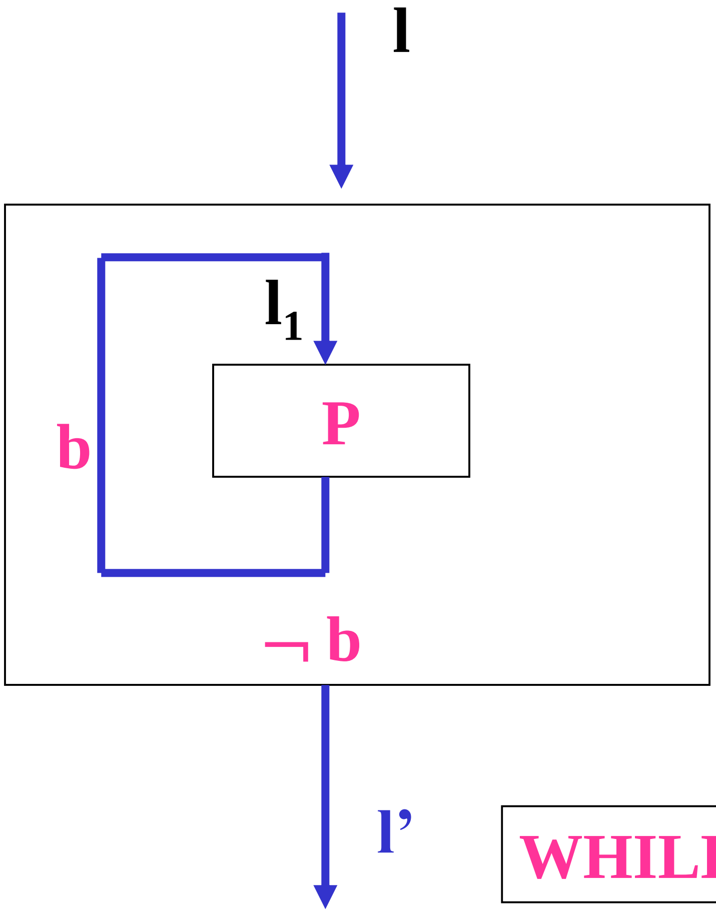
$(pc = l \wedge pc' = l_2 \wedge \neg b \wedge same(V) \wedge$
$same(PC - \{pc\}) \vee$

$C(l_1, P, l') \vee$

$C(l_2, Q, l')$

IF b THEN P ELSE Q FI

While statement



$C(l, \text{WHILE}(b, l_1), l')$

$(pc = l \wedge pc' = l_1 \wedge b \wedge \text{same}(V) \wedge$
$\text{same}(PC - \{pc\}) \vee$

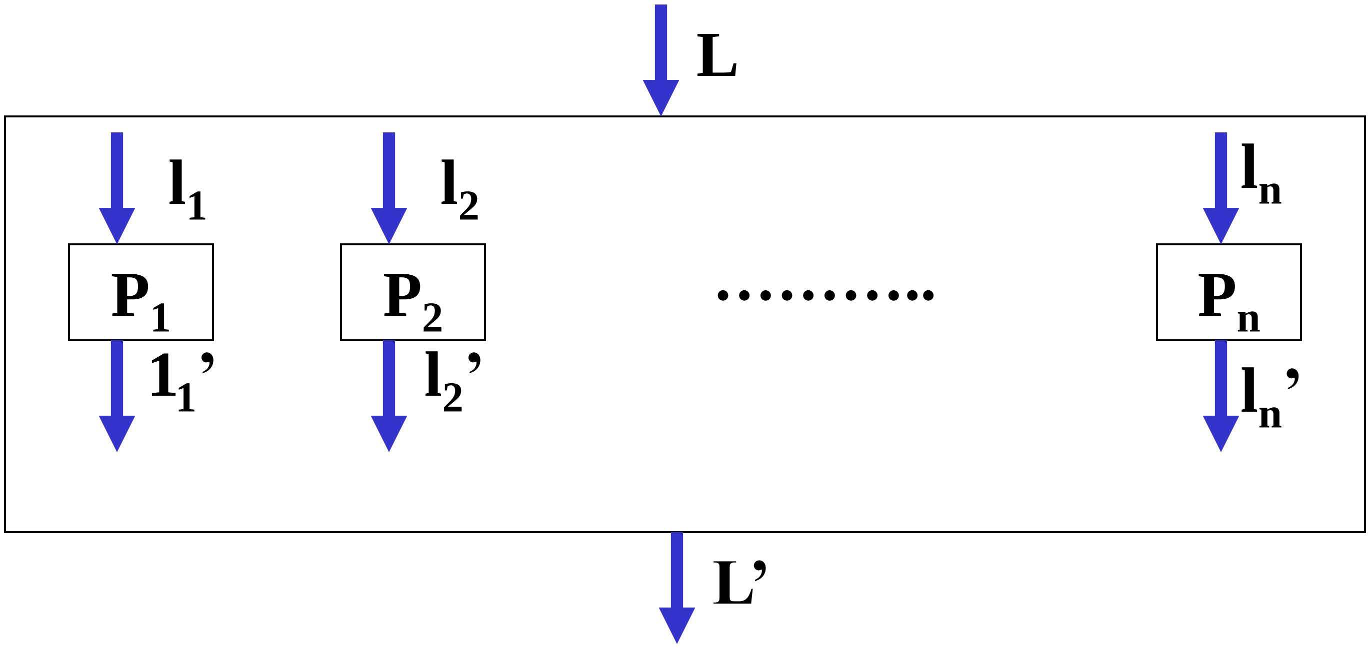
$(pc = l \wedge pc' = l' \wedge \neg b \wedge \text{same}(V) \wedge$
$\text{same}(PC - \{pc\}) \vee$

$C(l_1, P, l)$

WHILE b DO P END_WHILE

Concurrent programs

- **P = cobegin (P_1 || P_2 || …|| P_n) coend**
- **P_1, P_2,..P_n ---** Sequential Programs.



Concurrent programs

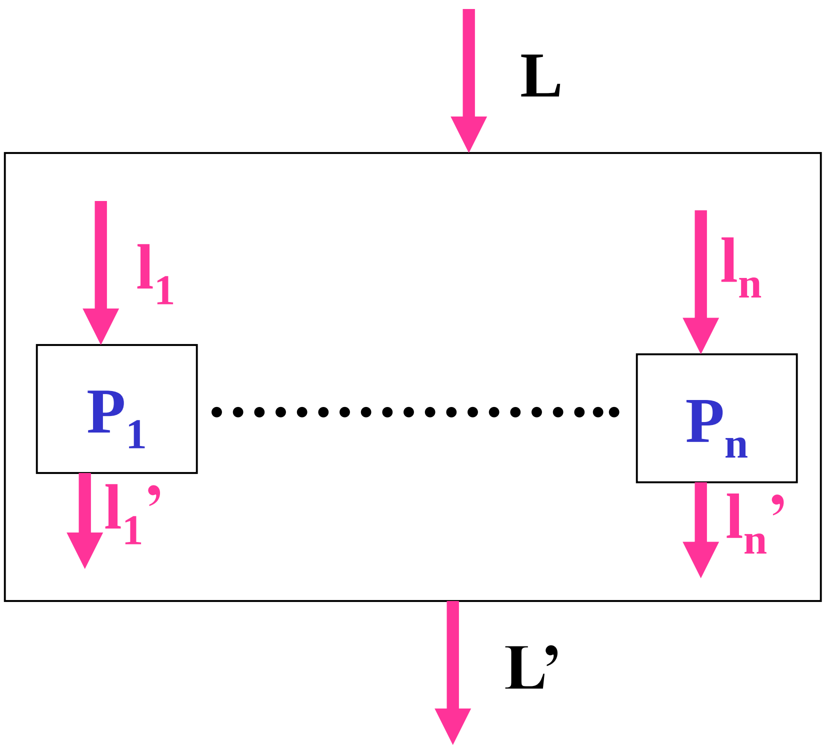
- **P** = **cobegin (**$P_1 \| P_2 \| \ldots \| P_n$**) coend**
- $P_1, P_2, \ldots P_n$ --- ***Sequential Programs***.
- $C(l_1, P_1, l_1')$ --- The transitions of program P_1 (defined ***inductively*** on the structure of P_1!).
- V_i ---- The set of variables of program P_i.
- Programs may ***share*** variables!
- pc_i – The program counter of program P_i.

Concurrent programs

- **pc** ---- the program counter of the ***concurrent program***; it could be part of a larger program!
- $\perp$ denotes an ***undefined*** program counter value.
- $S_0(V, PC)$ = **pre(V)** $\wedge$ **(pc=L)** $\wedge$

  $$(pc_1=\perp) \wedge \ldots\ldots \wedge (pc_n=\perp)$$

The Transition Predicate



C(L, P, L')

$(pc = L \wedge pc_1' = l_1 \wedge \ldots \wedge pc_n' = l_n \wedge$
$\wedge\ pc' = \perp \wedge same(V))$

$\vee$

$(C(l_1, P_1, l_1') \wedge Same\ (V - V_1)$
$\wedge\ Same(PC \setminus \{pc_1\}))$

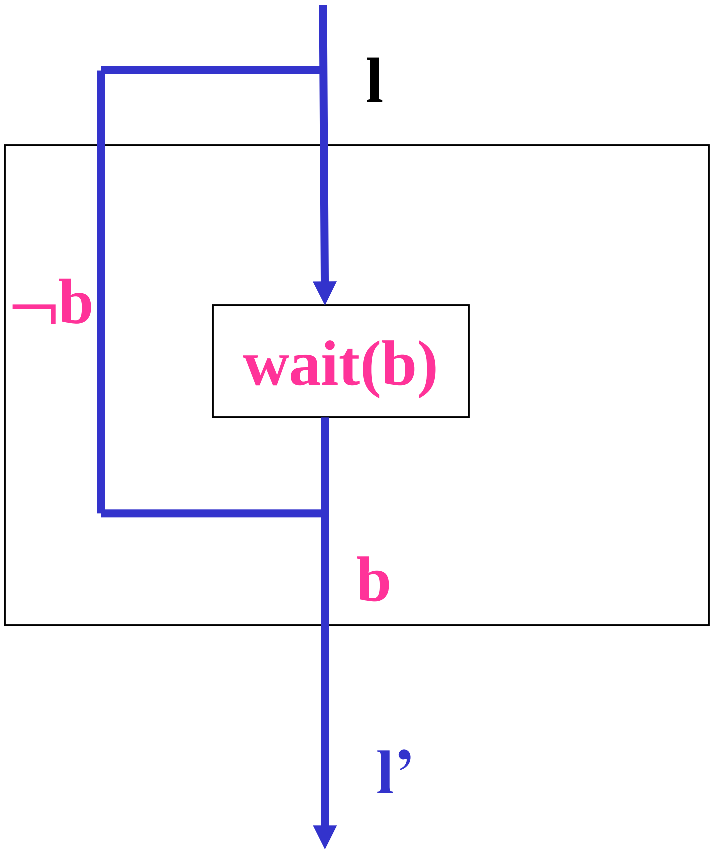
$\vee \ldots \vee$

$C(l_n, P_n, l_n') \wedge Same\ (V - V_n)$
$\wedge\ Same(PC \setminus \{pc_n\}))$

$\vee$

$(pc = \perp \wedge pc_1 = l_1' \wedge \ldots \wedge pc_n = l_n' \wedge$
$\wedge\ pc' = L' \wedge$
$pc_1' = \perp \wedge \ldots pc_n' = \perp \wedge same(V))$

The Transition Predicate

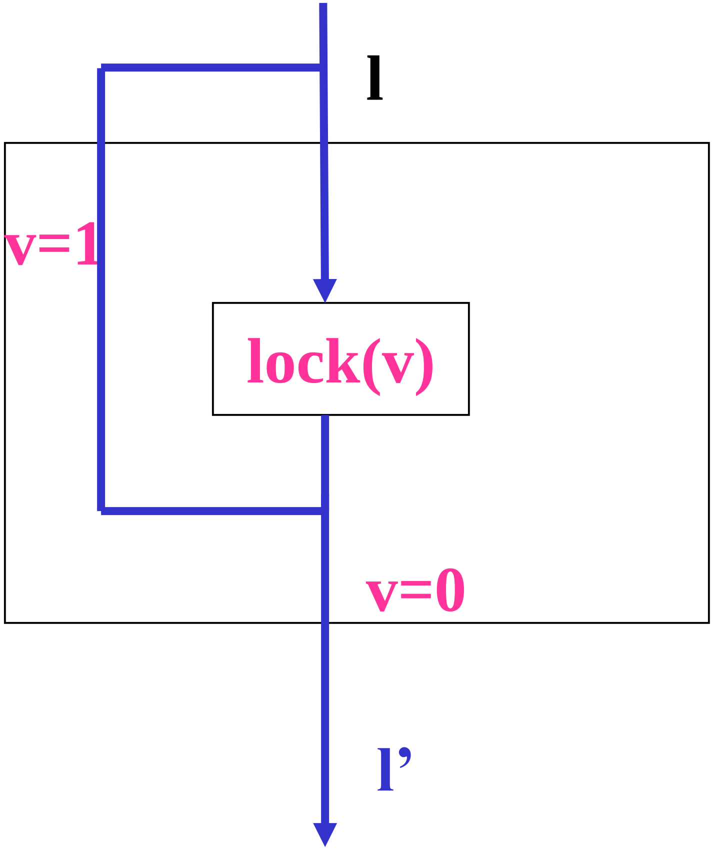


C(l, wait(b) , l')

$$(pc_i = l \wedge pc_i' = l \wedge \neg b \wedge same(V_i))$$
$$\vee$$
$$(pc_i = l \wedge pc_i' = l' \wedge b \wedge same(V_i))$$

Repeatedly tests the boolean expression **b** until it is true.
When **b** becomes **true** proceeds to the next step.

The Transition Predicate



C(l, lock(v) , l')

$$(pc_i = l \wedge pc_i' = l \wedge v = 1 \wedge same(V_i))$$

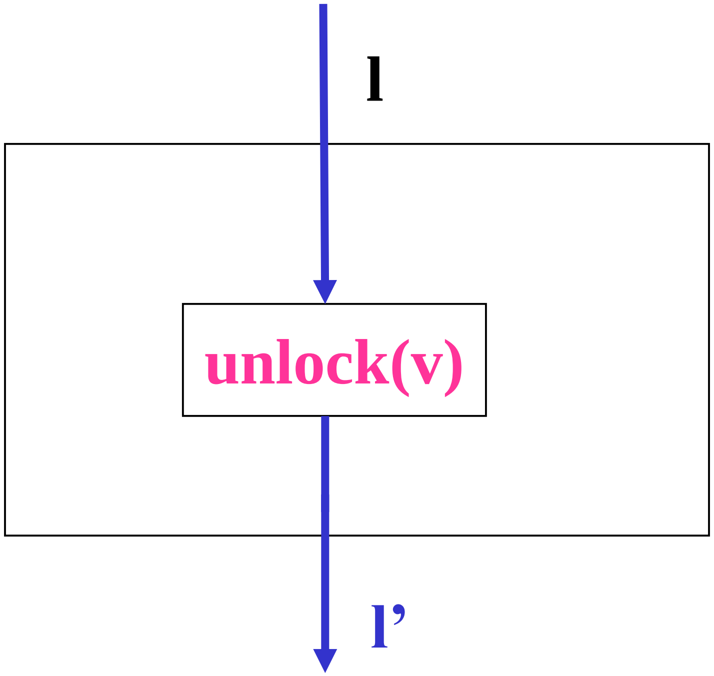
$$\vee$$

$$(pc_i = l \wedge pc_i' = l' \wedge v = 0 \wedge v' = 1 \wedge same(V_i \setminus \{v\}))$$

Similar to **wait** with boolean expression **v=0**, but when the condition becomes **true**, **v** is updated to **1** and it proceeds to next step.

The Transition Predicate

C(l, unlock(v) , l’)



$$(pc_i = l \wedge pc_i' = l' \wedge v' = 0 \wedge same(V_i \setminus \{v\}))$$

Simply sets variable **v** to **0**, thus, possibly, enabling other processes to trigger their **lock (**or **wait**) transition to enter critical regions.

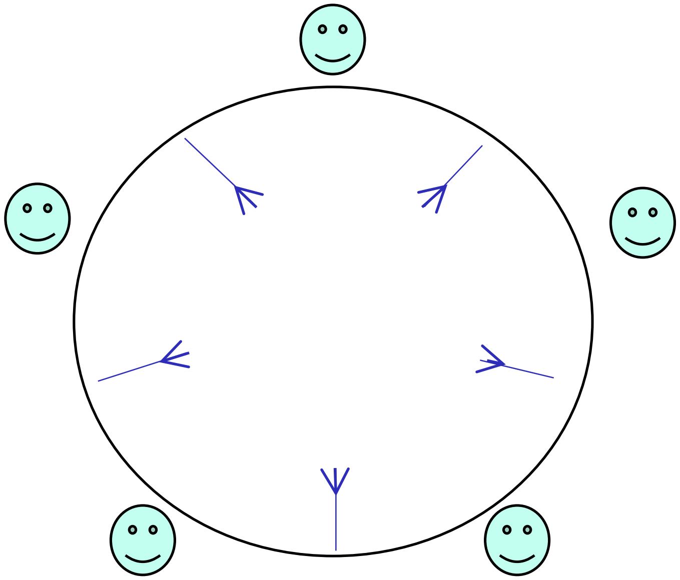
Summary

- System variables
- Domain of values
- States
- Initial state predicate
- Transition predicate
- pc values (for programs)
- Synchronization mechanisms

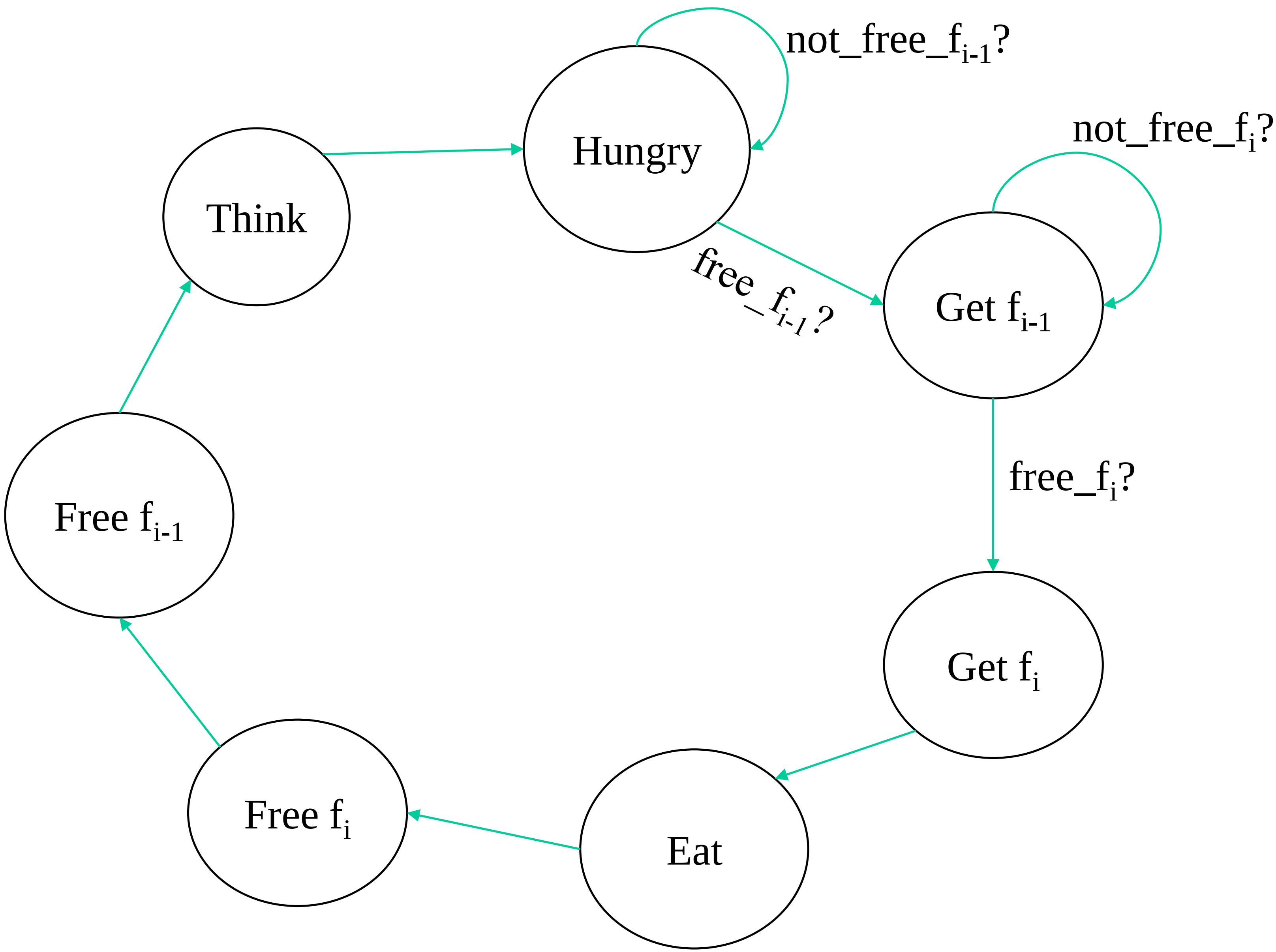
Example: shared resurces

"Dining Philosophers"

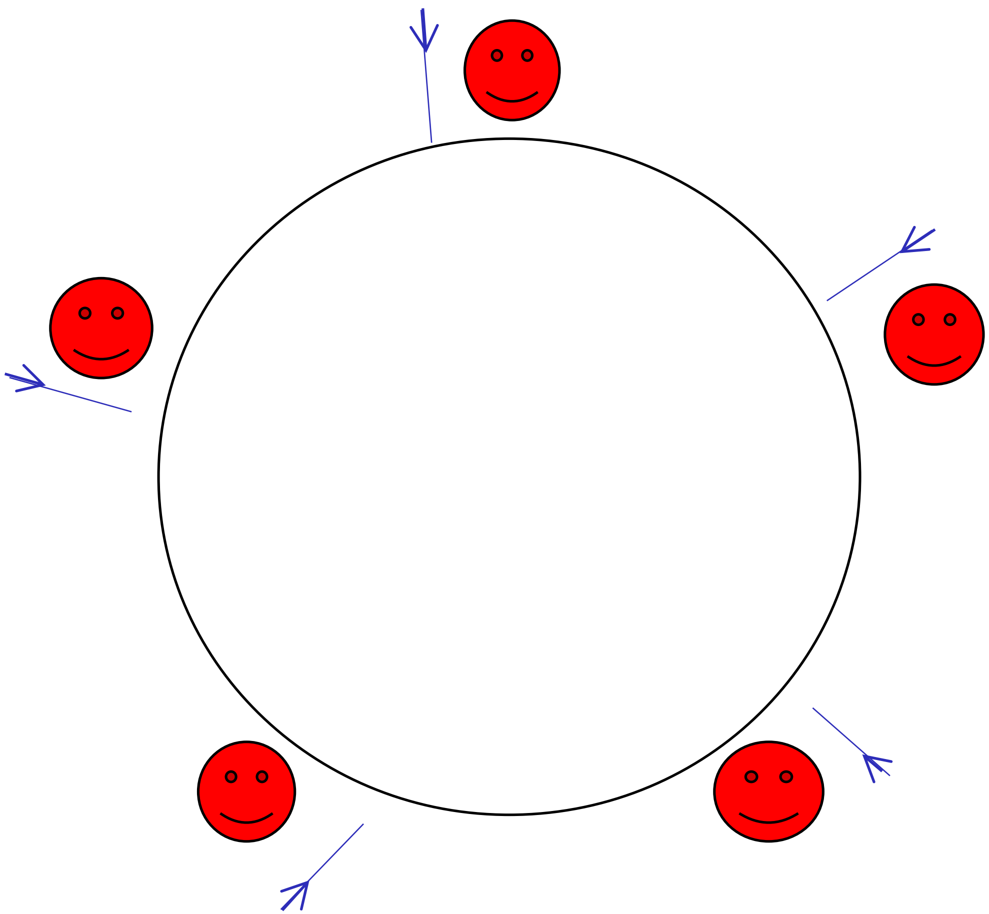
- Five philosophers sit around a table;
- Next to each philosopher is a fork (5 philosophers and 5 forks);
- Philosophers think most of the time and, when hungry, they can eat as long as they can grab two forks.



Possible philosopher's automaton



A problem: deadlock

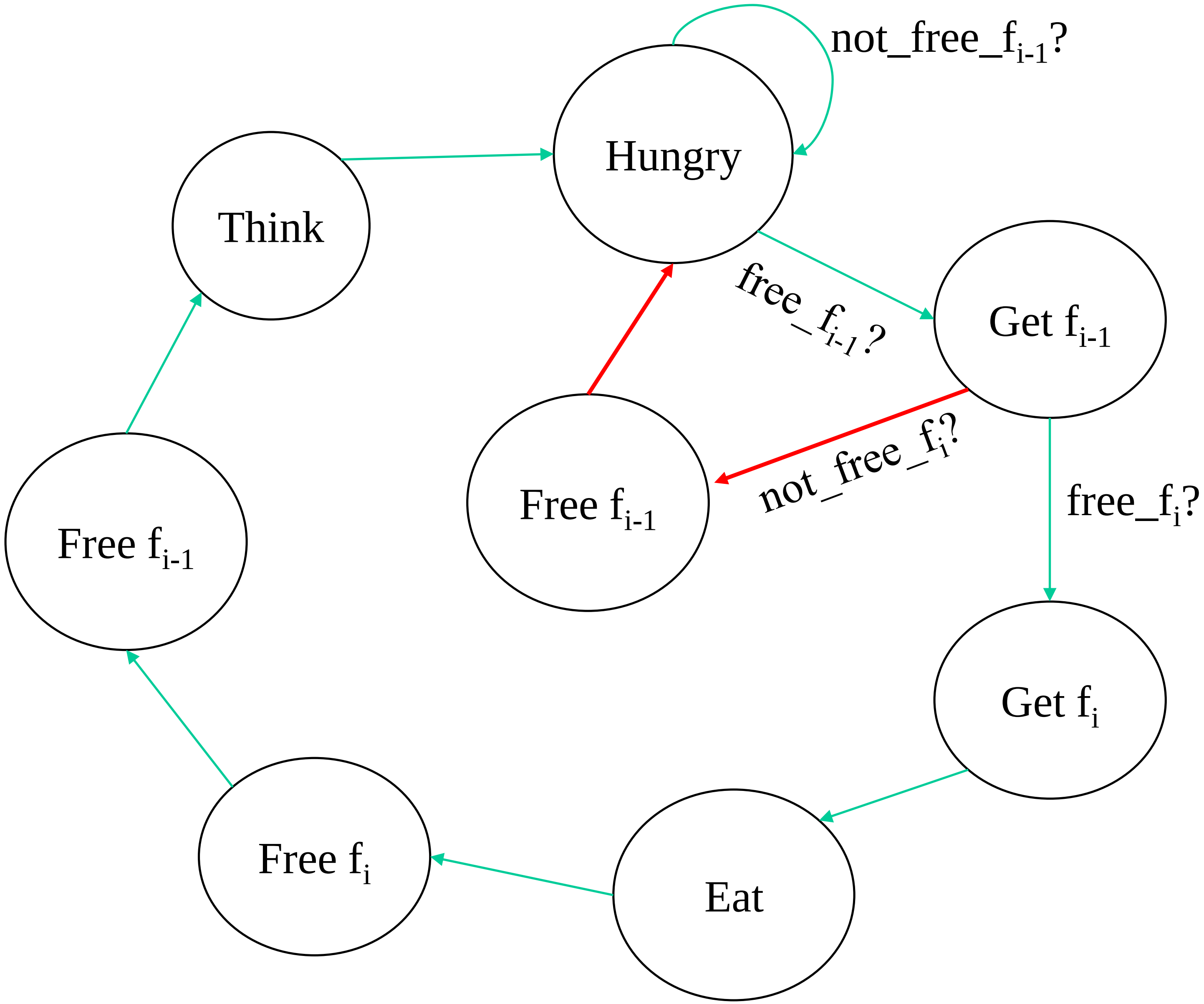


Problems

"Dining Philosophers"

- Possible problems:
  - ***Deadlock***: System state where no further action is possible (global state change).
  - ***Starvation***: When one system component is prevented to access the resurce.
  - ***Livelock***: When no component is "blocked" but the system, as a whole, cannot progress.

Alternative solution: no deadlock



Fairness

Dining Philosophers

- A possible solution to deadlock:
  - ***Pick up left fork only if both are present***

  System assumptions:

  - ***weak fairness***: transitions ***continuously enabled*** will ***eventually*** be executed (e.g., each philosopher will stop eating)
  - ***strong fairness***: transitions enabled ***infinitely often*** will ***eventually*** be executed (e.g., if 2 forks are available infinitely often, the phisolopher will be able to eat).

Starvation

Dining Philosophers

- Possible solution
  - ***Pick up left fork only if both are present***

Assumption:

- ***strong fairness***: transitions enabled ***infinitely often*** will ***eventually*** be executed (e.g., if 2 forks are available infinitely often, the philosopher will be able to eat).

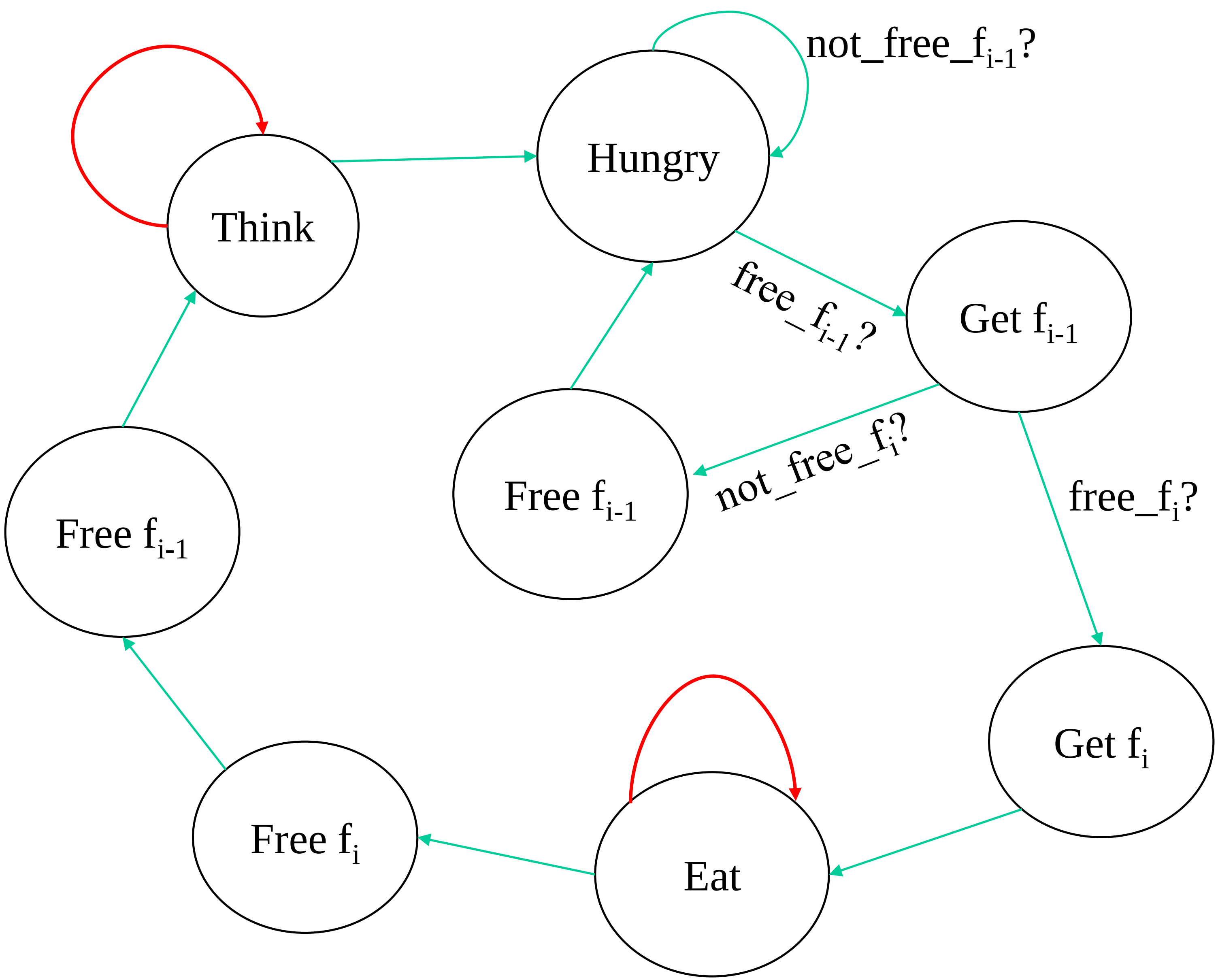
Strong fairness is not sufficient to avoid ***starvation***

Why? Think to the case of 4 philosophers!

Sol.(?): ***Prevent consecutive forks pick ups by each philosopher***.

Still suffers from ***starvation*** with 5 philosophers! ***Why***?

Non Determinismo

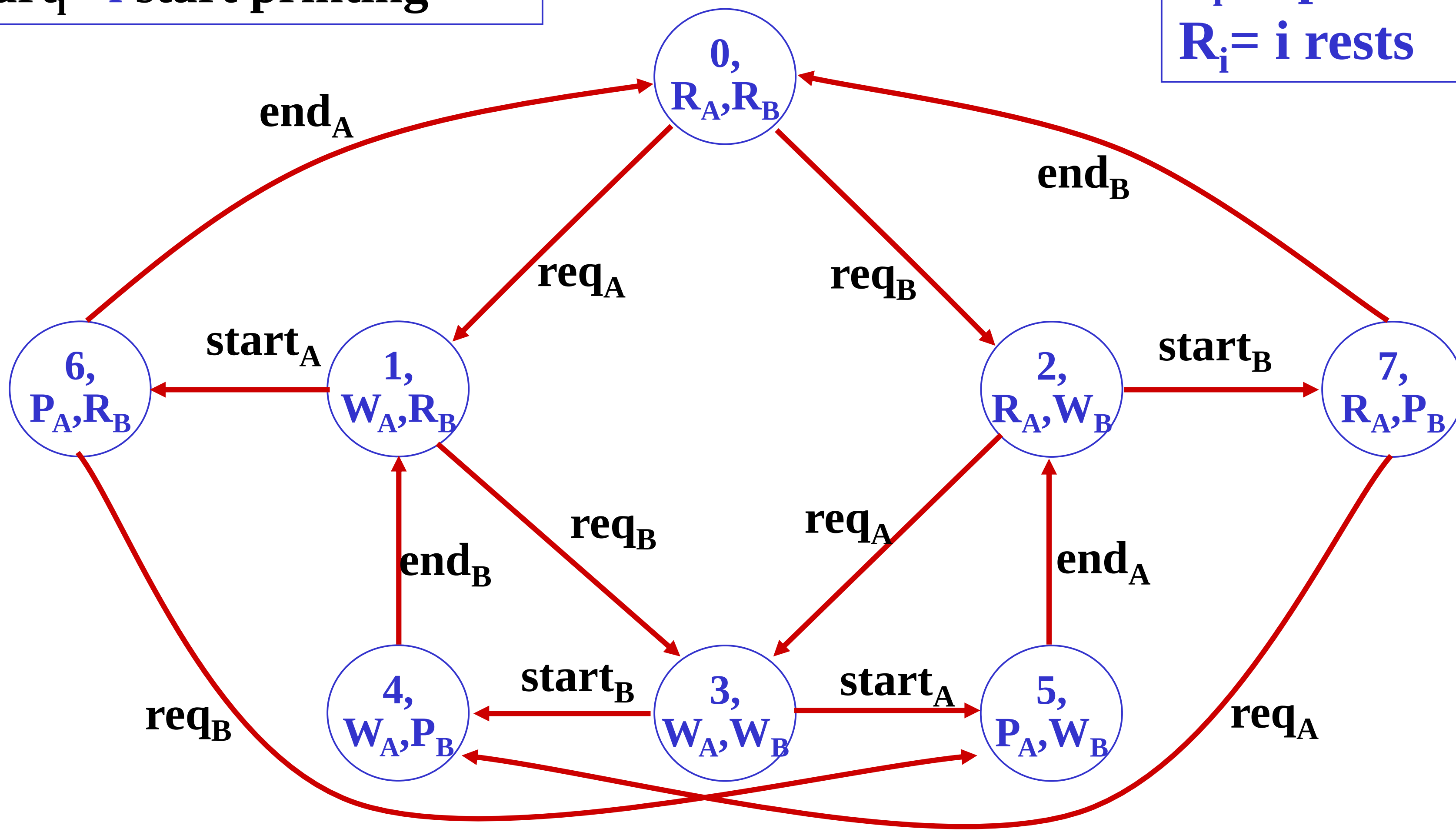


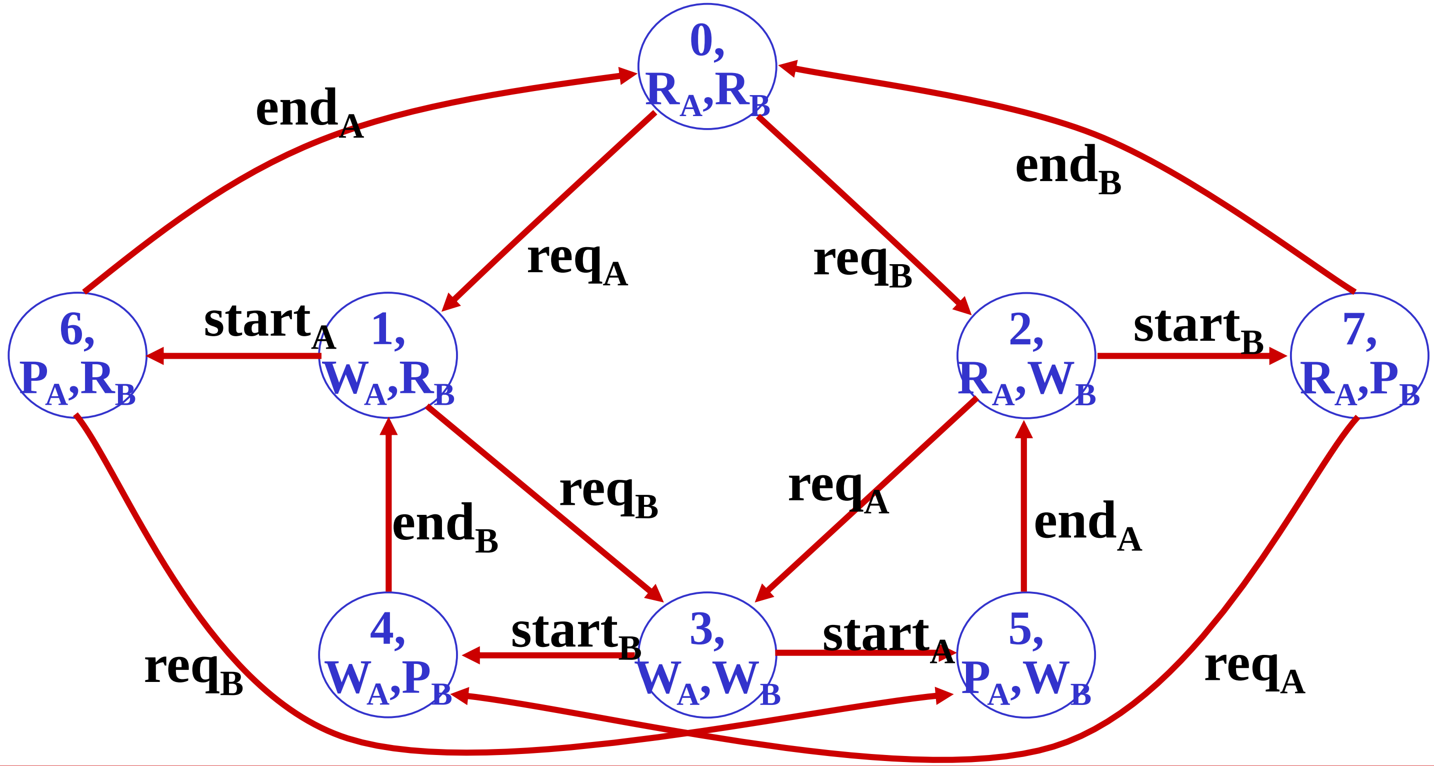
Example: a print manager

$\mathbf{end_i}$ = i ends printing
$\mathbf{req_i}$ = i requests printing
$\mathbf{start_i}$ = i start printing

AP

W_i = i waits
P_i = i prints
R_i = i rests





- $S = \{0,1,2,3,4,5,6,7\}$
- $A = \{end_A, end_B, req_A, req_B, start_A, start_B\}$
- $R = \{(0,req_A,1), (0,req_B,2), (1,req_B,3), (1,start_A,6), (2,req_A,3), (2,start_B,7), (3,start_A,5), (3,start_B,4), (4,end_B,1), (5,end_A,2), (6,end_A,0), (6,req_B,5), (7,end_B,0), (7,req_A,4),\}$
- $L = \{0 \rightarrow \{R_A,R_B\}, 1 \rightarrow \{W_A,R_B\}, 2 \rightarrow \{R_A,W_B\}, 3 \rightarrow \{W_A,W_B\}, 4 \rightarrow \{W_A,P_B\}, 5 \rightarrow \{P_A W_B\}, 6 \rightarrow \{P_A,R_B\}, 7 \rightarrow \{R_A P_B\}\ \}$

Properties of the printing systems

1. Every state in which $\mathbf{P_A}$ holds, is preceded by a state in which $\mathbf{W_A}$ holds
2. Any state in which $\mathbf{W_A}$ holds is followed (possibly not immediately) by a state in which $\mathbf{P_A}$ holds.

- The first can easily be checked to be true
- The second is ***false*** (e.g. 0134134134…) - in other words the system is ***not fair***.

Transition Relation

- V = {x, y, z}
- Program : {x, y, z, pc}

  l_0 : begin

  l_1 : statement$_1$

  l_2 : statement$_2$

  ….

  l_5 : if even(x) then x = x/2 else x = x –1

  l_6 : ….

Transition Relation

- **V = {x, y, z}**
- Program : **{x, y, z, pc}**

  l_5 : if even(x) then x = x/2 else x = x –1

  l_6 : ….

- **φ (x, y, z, pc, x', y', z', pc')**
- **$pc = l_5 \wedge pc' = l_6 \wedge (\exists n. (x = 2n) \supset x' = x/2) \wedge$**
  $(\neg\exists n. (x = 2n) \supset x' = x\text{-}1) \wedge same(y, z)$

Notice that the formula above is equivalent to:

- **$pc = l_5 \wedge pc' = l_6 \wedge$**
  $((\exists n.(x=2n) \wedge x'=x/2) \vee (\neg\exists n.(x=2n) \wedge x'=x\text{-}1)) \wedge$
  $same(y, z)$
- **where same(y, z) stands for $y' = y \wedge z' = z$**

Transition Relation

- In a similar fashion , we can specify the transition relation formulae for :
  - Assignment statement
  - While statements
  - etc.etc.
  - See the text book!