Tecniche di Specifica e di Verifica

Modeling with Transition Systems

An example

The Dining Philosophers

- Possible problems:
  - ***Deadlock***: system state where no action can be taken (no meaningful transition possible)
  - ***Starvation***: When a system component is prevented access to resources.
  - ***Livelock***: When the system components are not blocked but the system as a whole does not progress (e.g., some components are prevented to take specific actions).

Fairness

The Dining Philosophers

- A possible solution to deadlock:
  - ***pick up right fork only if both are present***

  Useful assumptions on the system:

  - ***weak fairness***: any phil. trans. ***continuously*** enabled will ***eventually*** fire (e.g. eating philosophers will finish)
  - ***strong fairness***: any phil. trans. enabled ***infinitely often*** will ***eventually*** occur (e.g. if 2 fork available infinitely often, phil. will eventually eat).

Starvation

The Dining Philosophers

- Possible solution:
  - ***pick up fork only if both are present***

  Assumptions:

  - ***strong fairness***: any phil. trans. enabled infinitely often, will eventually occur (if 2 fork available infinitely often, philosopher will eventually eat).

strong fairness is not enough to prevent ***starvation***

Why? Think of the case with 4 philosophers!

Sol.(?): Try ***preventing consecutive eating***.

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

Outline

- The model – Transition systems
- Some features
  - Paths
  - Computations
  - Branching
- First order representation

Transition systems

- A transition system (***Kripke structure***) 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**.

Notions and Notations

- **TS = (S, S_0, R)**
- **(s, s') $\in$ R   R(s, s')   s $\rightarrow$ s'**
- A (finite) ***path*** from **s** is a sequence

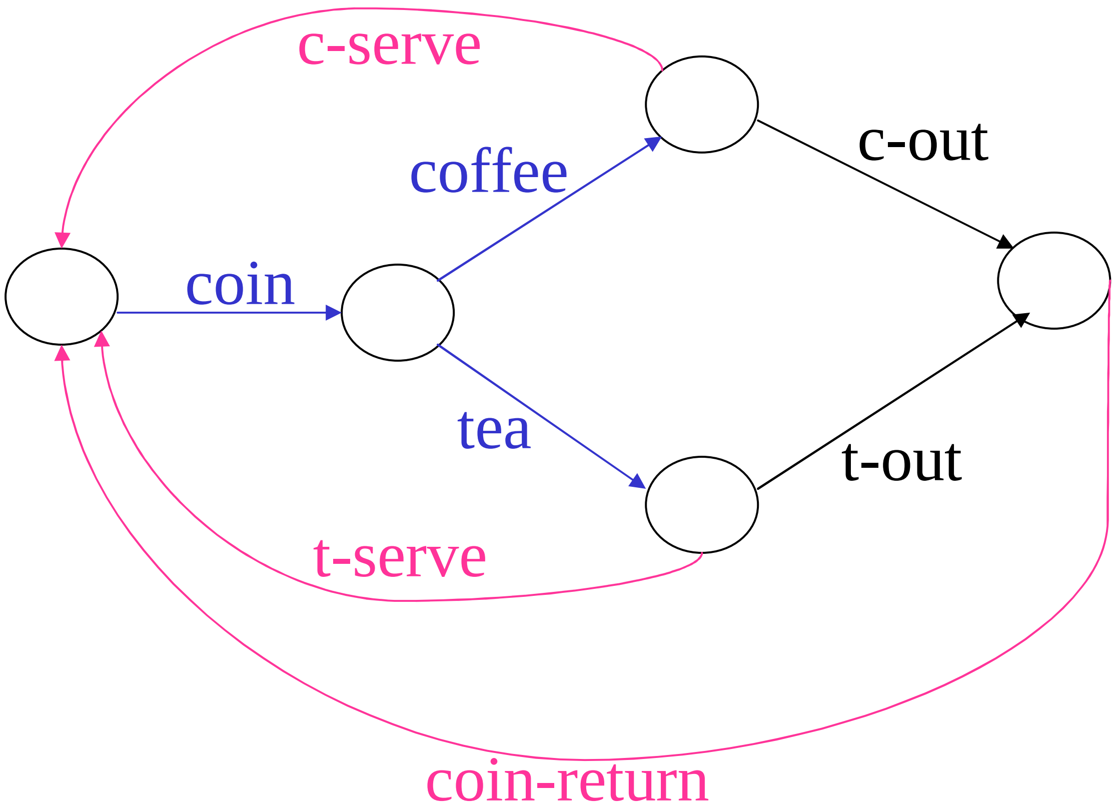
  $s_1, s_2, \ldots, s_n$

  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*** …..

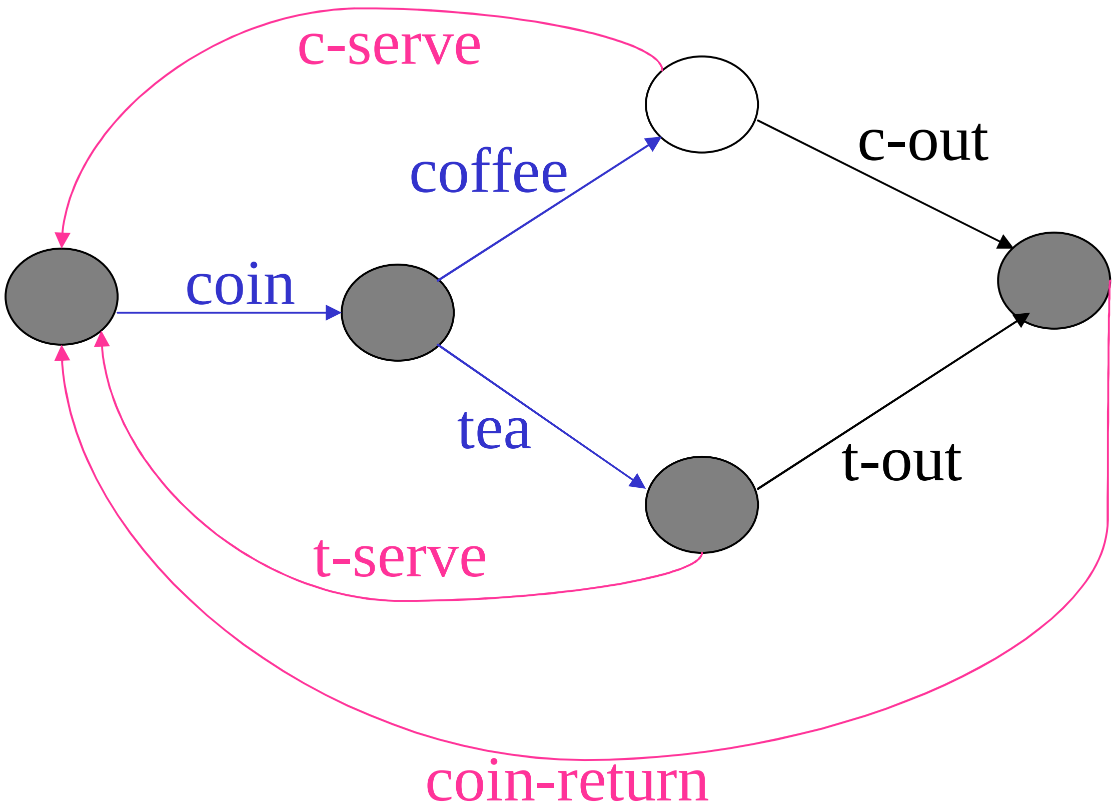
Labeled transition systems

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

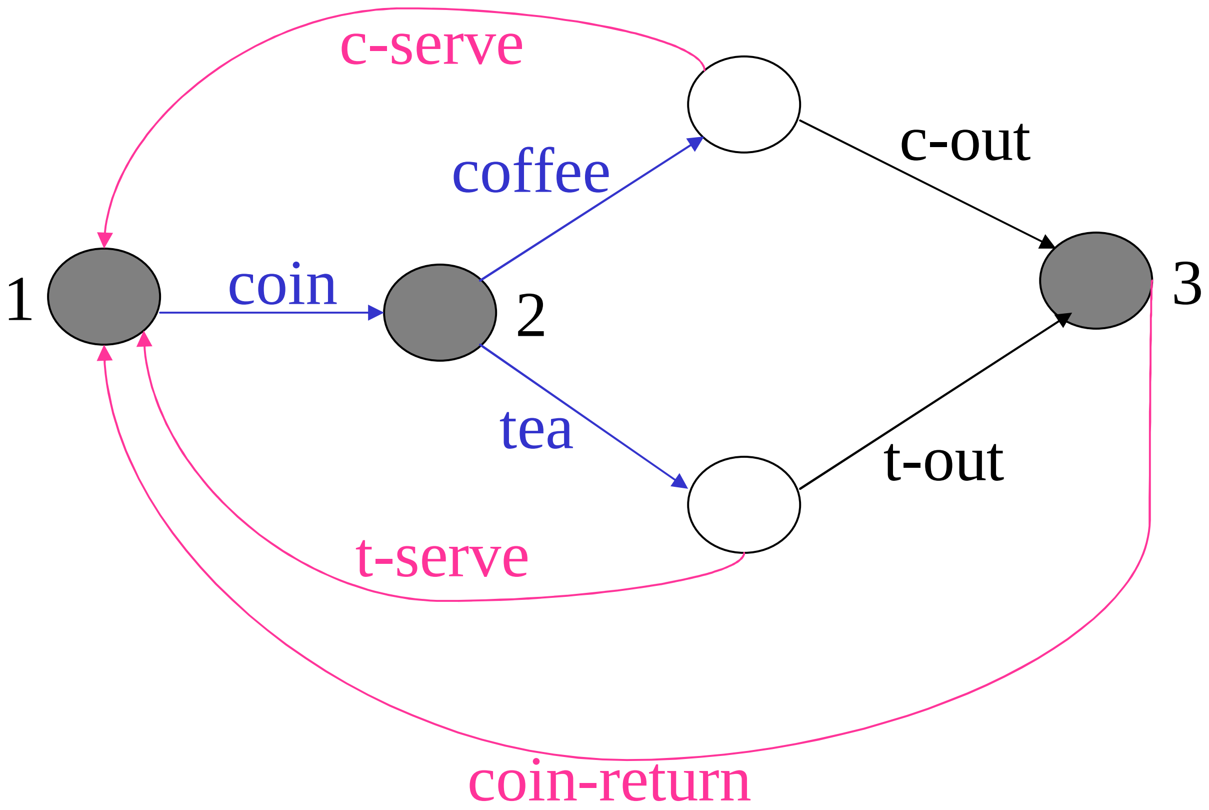
A vending machine



A path



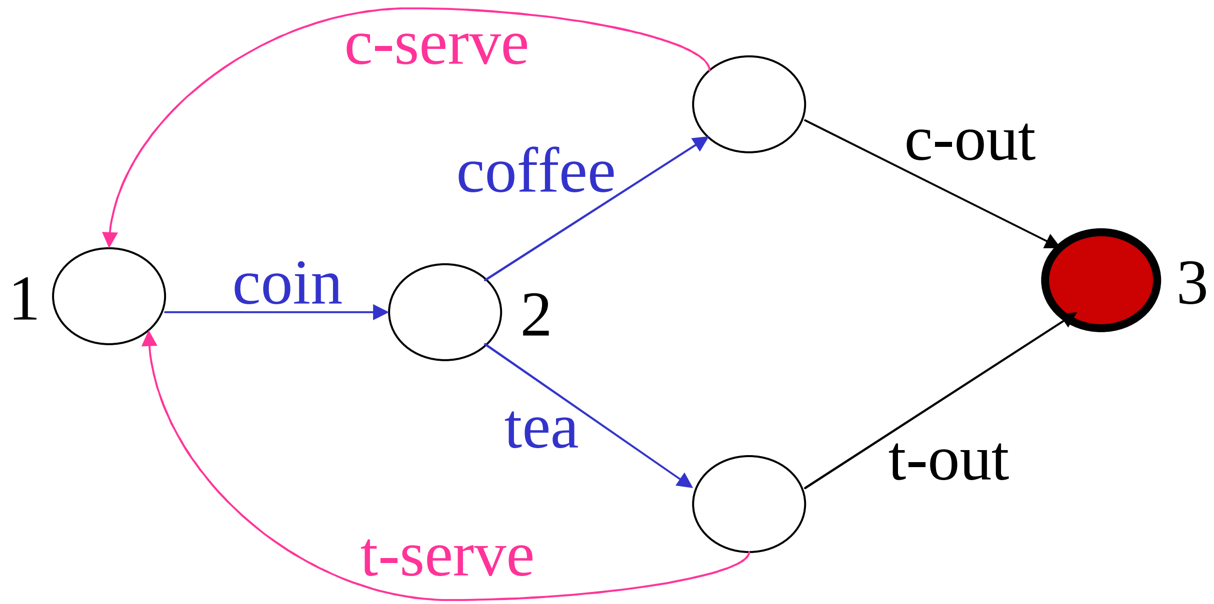
A non-path



1 2 3 No!

3 1 2 yes!

A non-total transition relation



State space

- The ***state space*** of a system (e.g. program) is the set of ***all its possible states***.
- For example, if **V={a, b, c}** and the variables range over the naturals, 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>

…

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.
- Is a:=a+1 atomic?
- 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

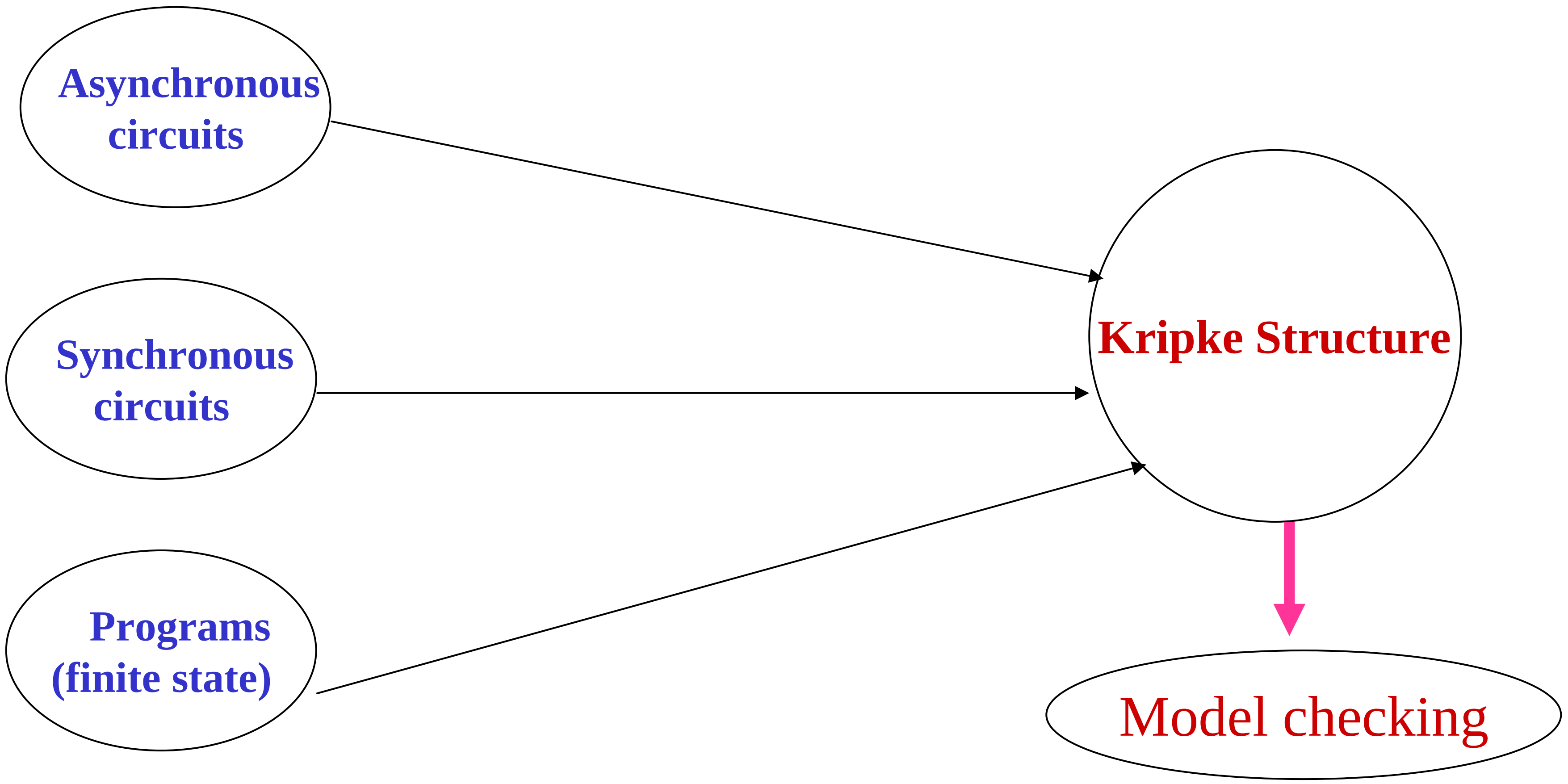
P2:load R2,a
inc R2
store R2,a

- **a may also be 1**

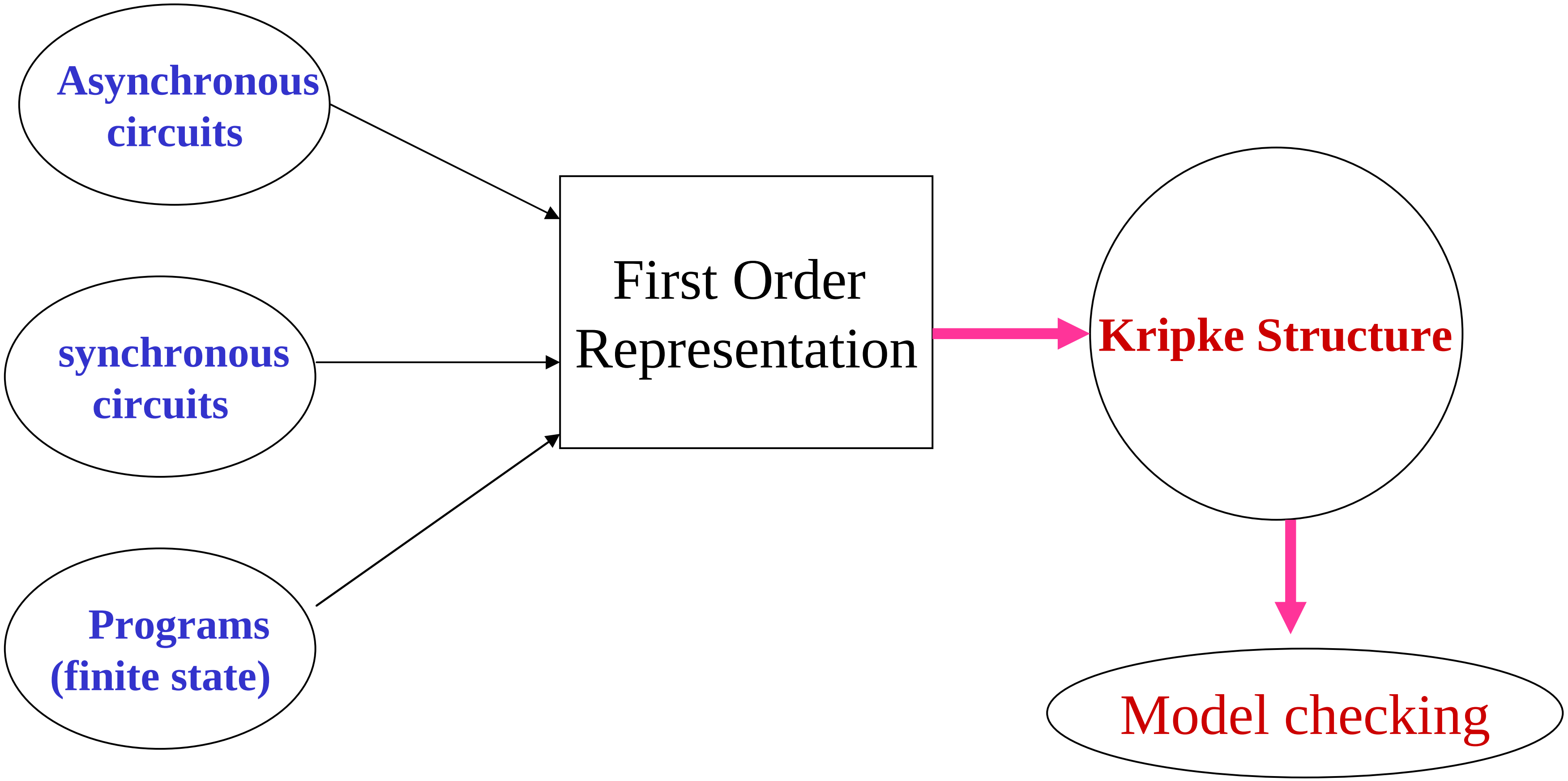
The common framework

- 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***

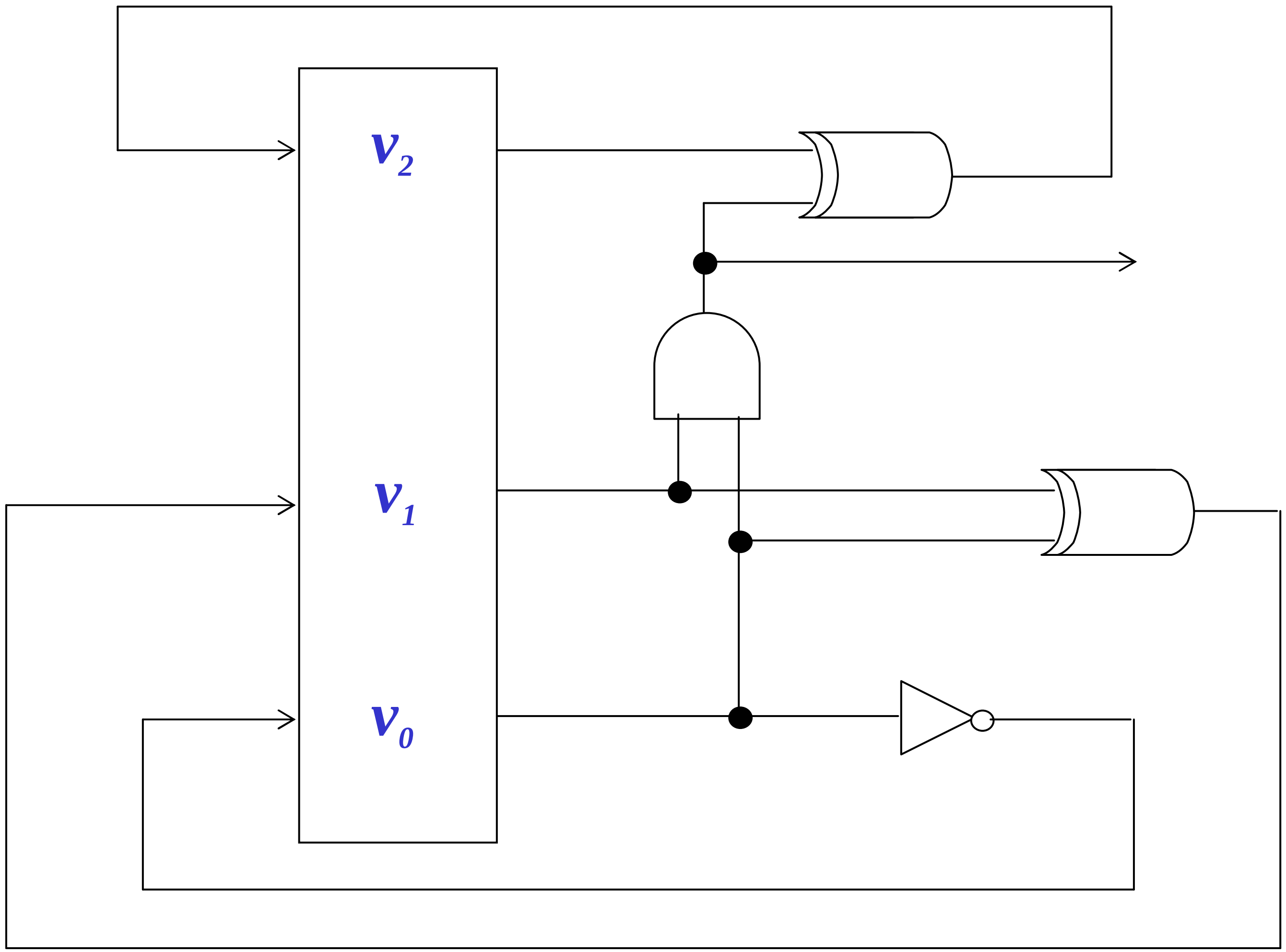
The inefficient way



The efficient way



Synchronous counter modulo 8

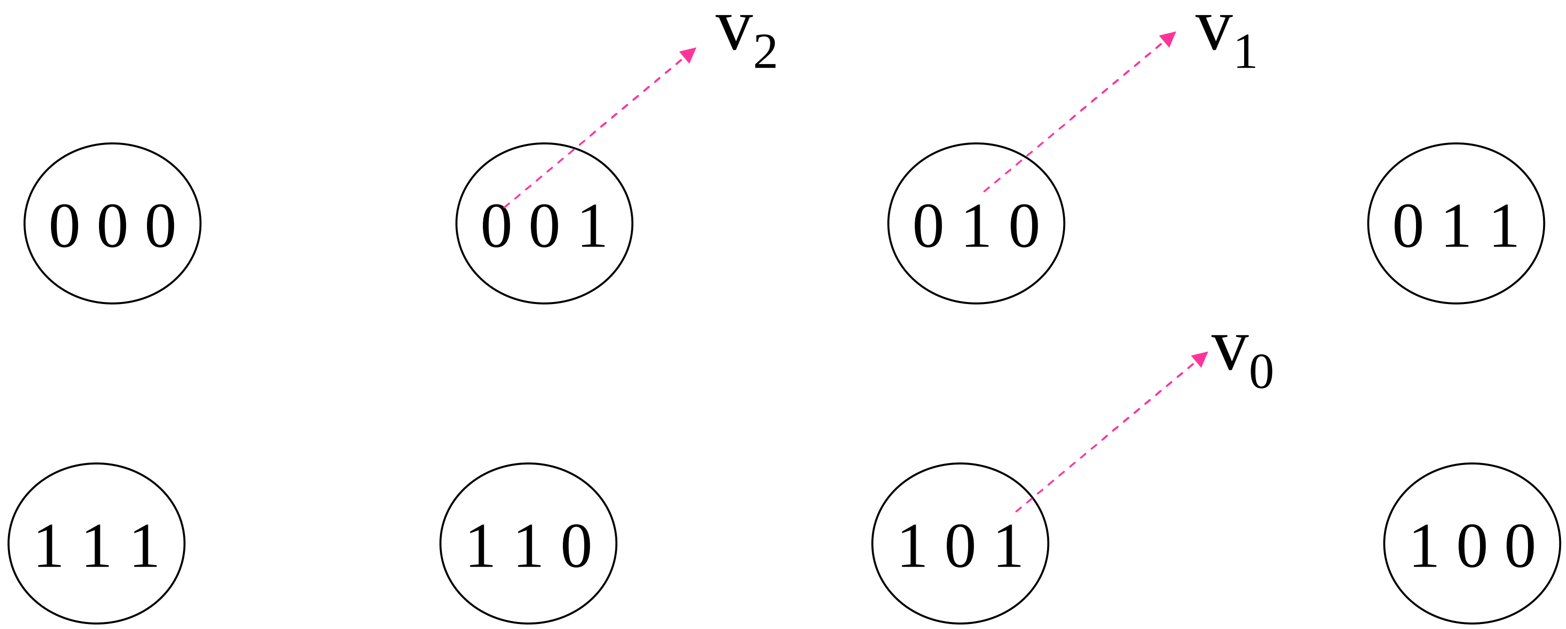


The mod-8 counter

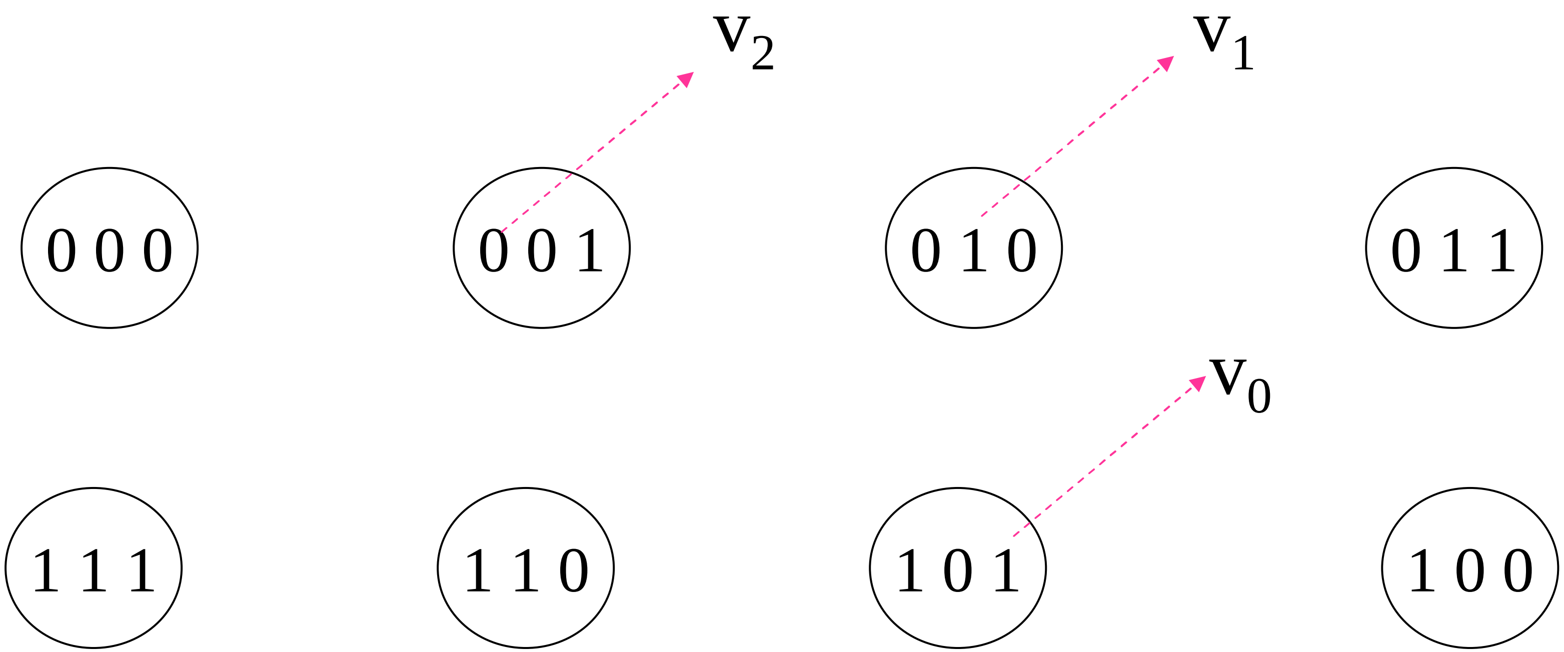
- System variables : $V = \{v_2\ v_1\ v_0\}$
- Domain of v_2 is $\{0, 1\}$

  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.
  - **$s : V \rightarrow B$**
  - **$s(v_0) = 0$ $s(v_1) = 1$ $s(v_2) = 1$**
  - This specifies the state **s** = (1 1 0) !

A mod-8 counter: states



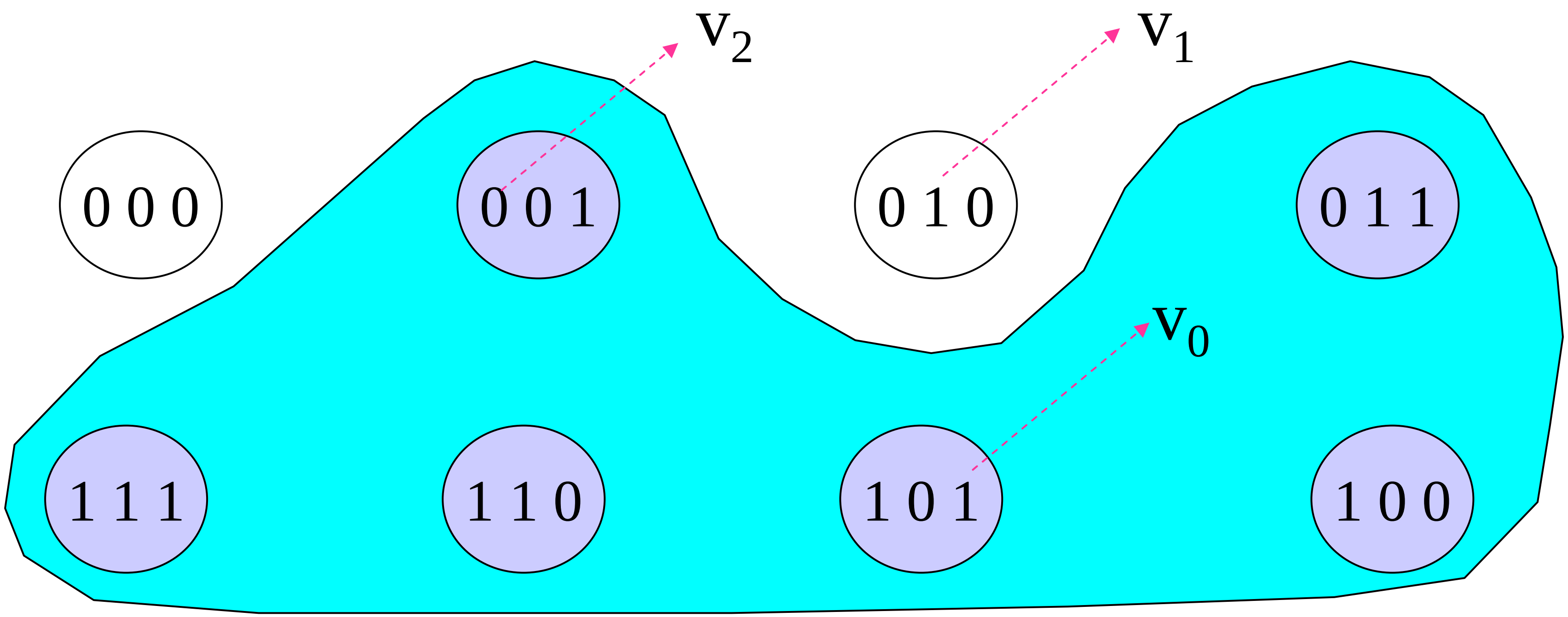
State Predicates



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

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

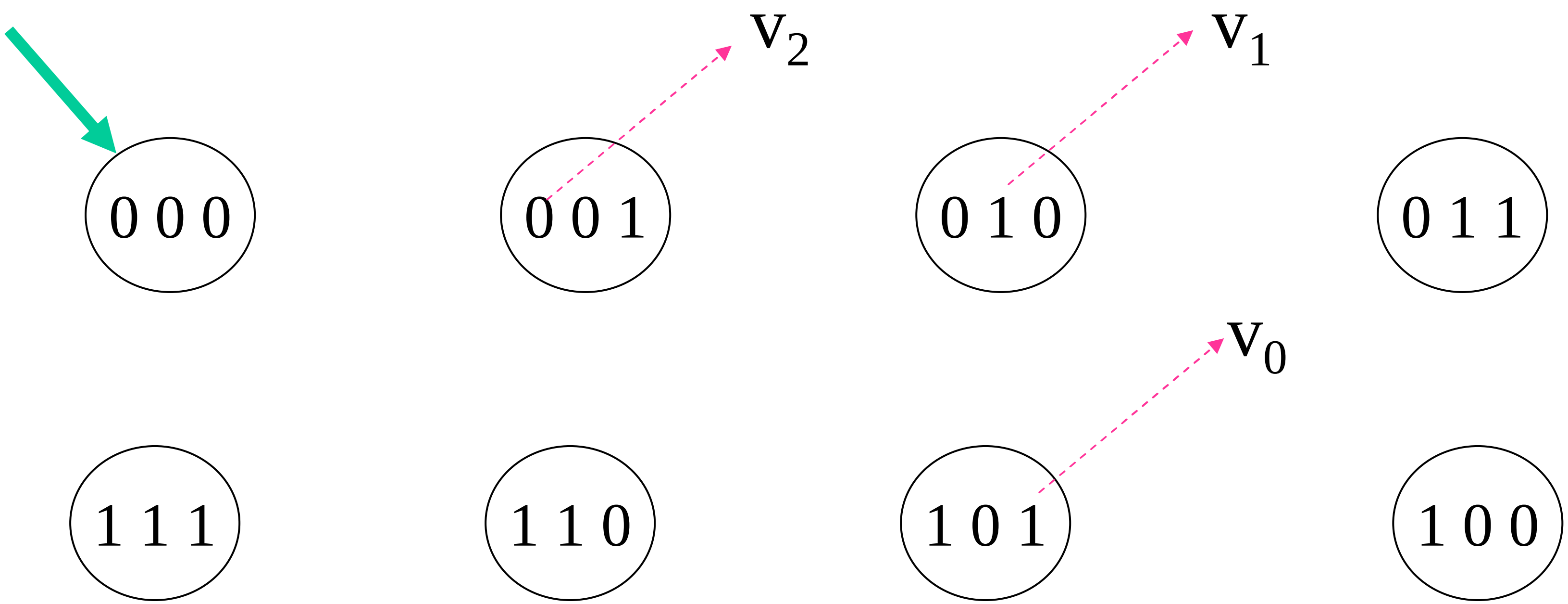
State Predicates



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

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

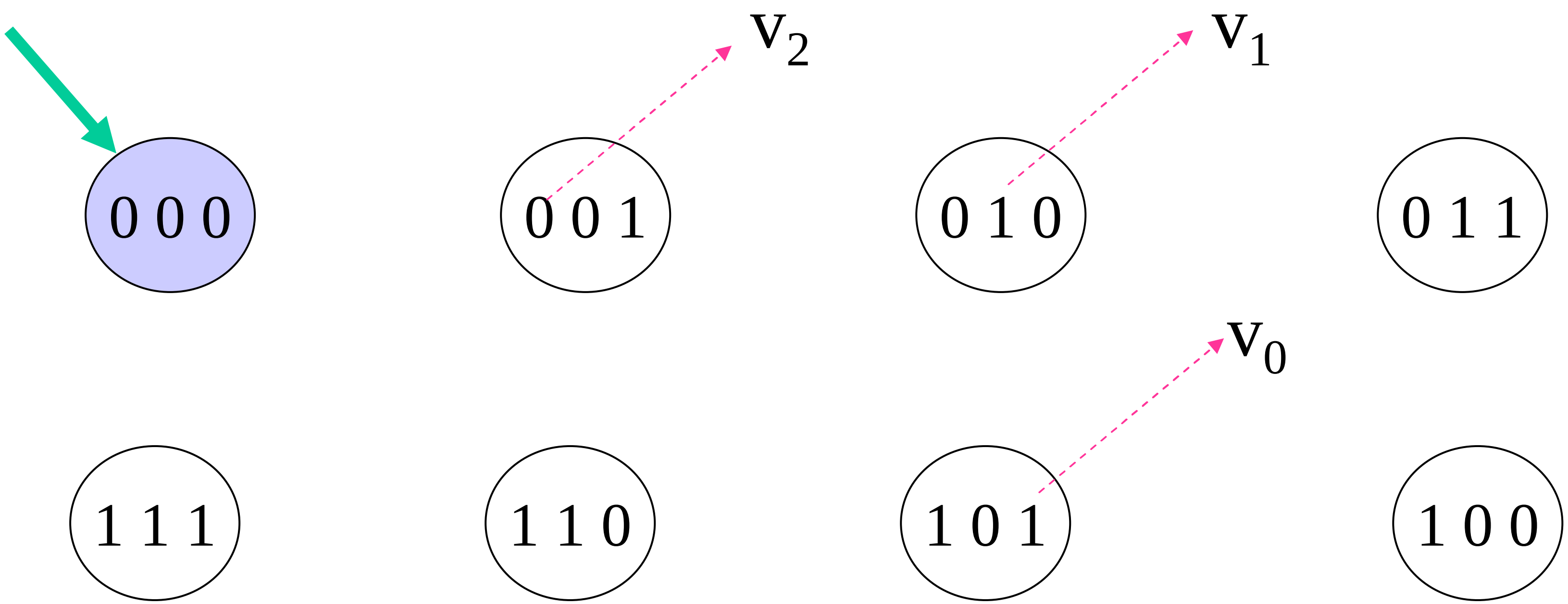
Initial States Predicate



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

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

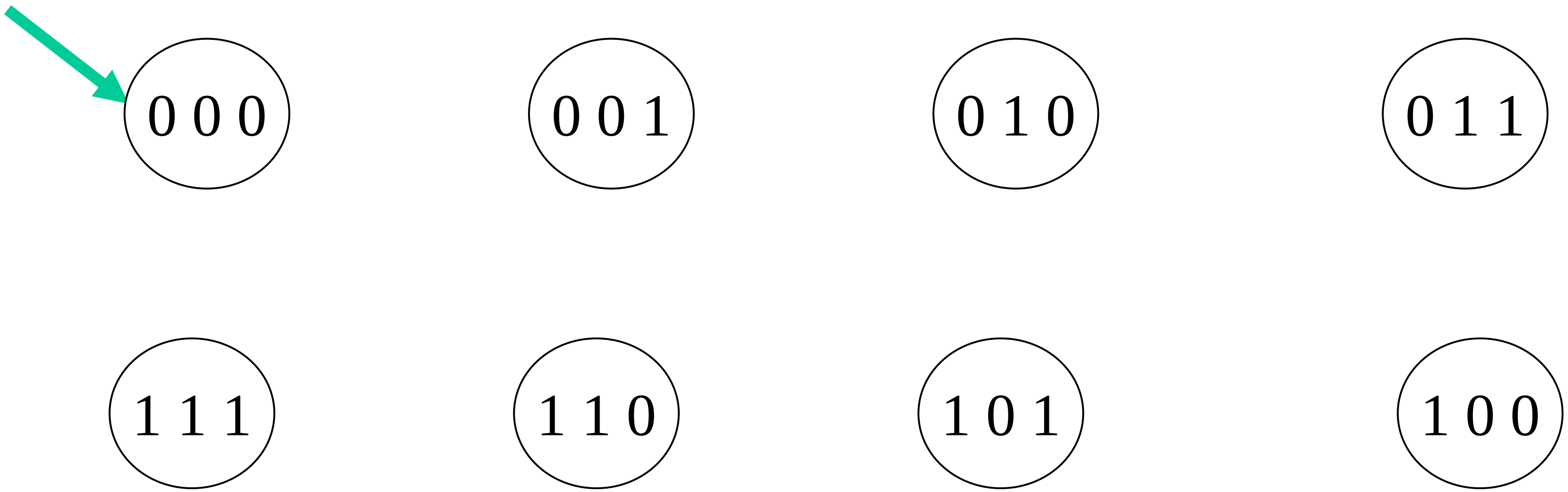
Initial States Predicate



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

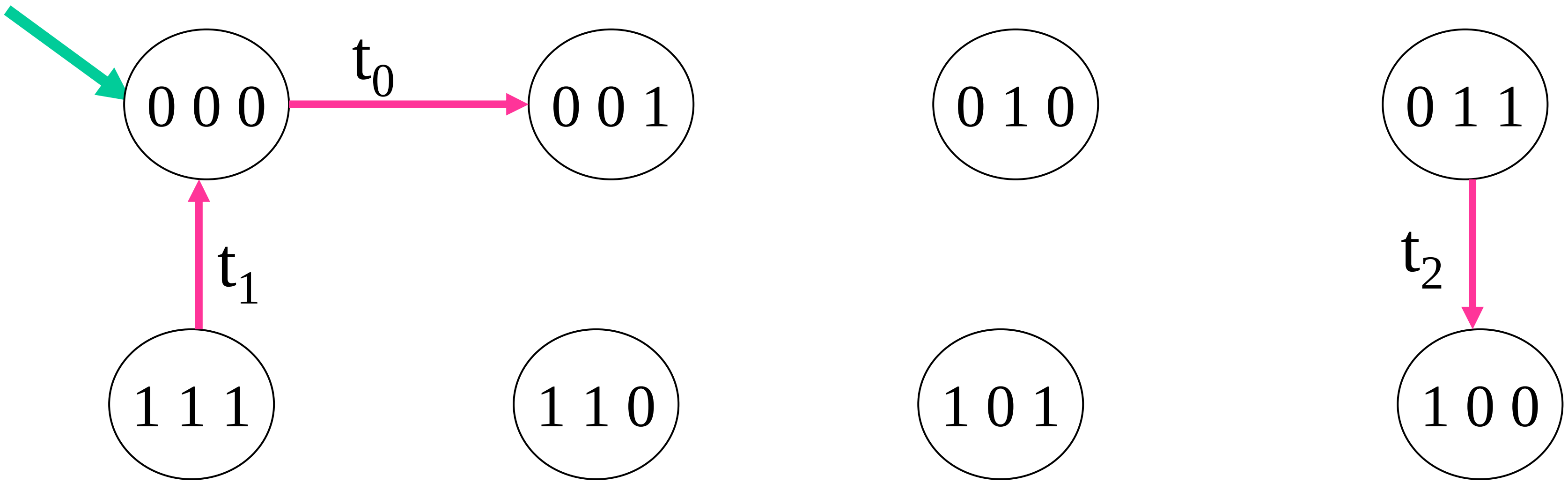


A set of *transitions* can also be picked out by a formula.

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

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

Transition relation predicate

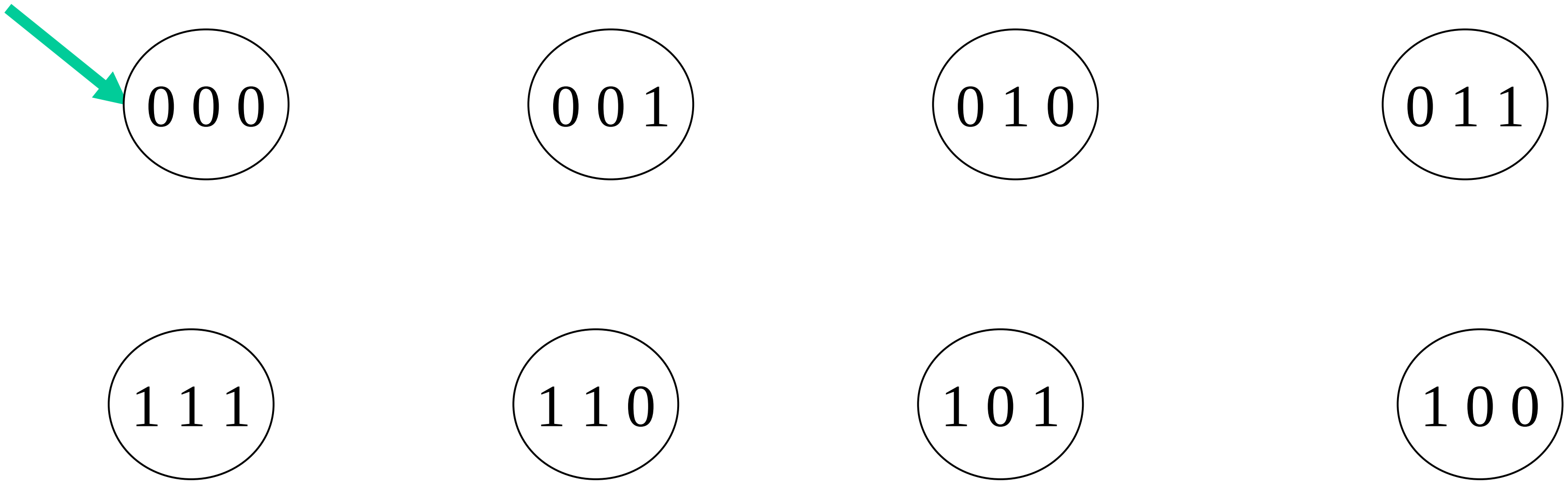


A set of transitions can also be picked out by a formula.

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

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

Transition relation predicate

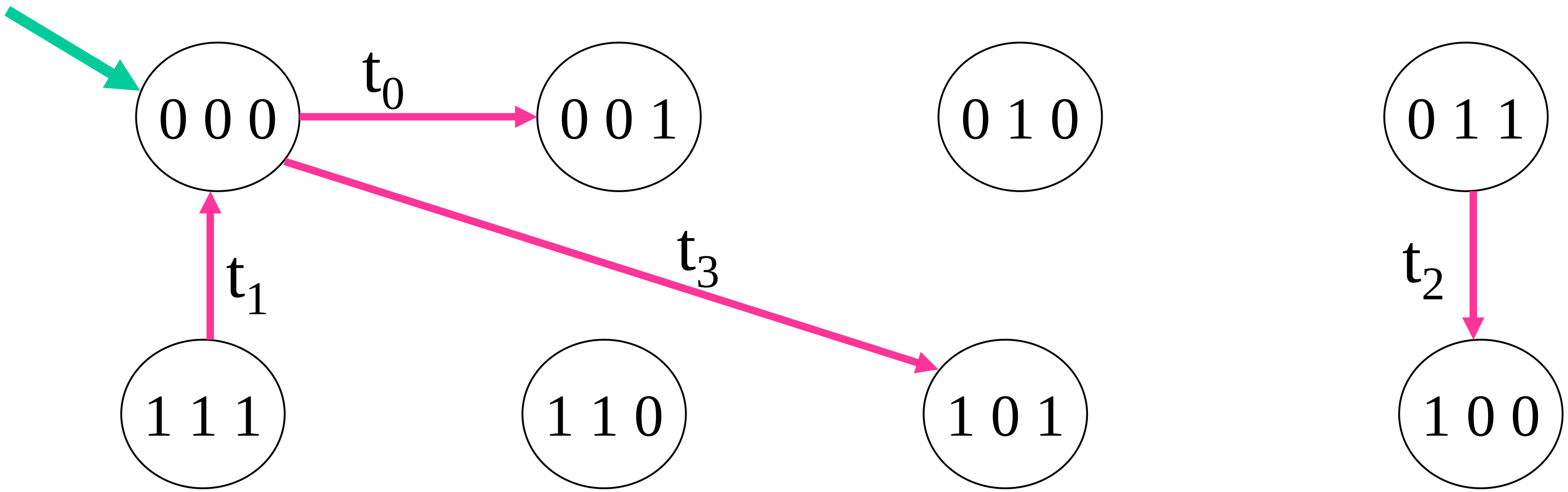


R = **Formula**(v_2, v_1, v_0, v_2', v_1', v_0')

Not all formulae will define subsets of transitions.

You must pick the right formula .

Transition relation predicate



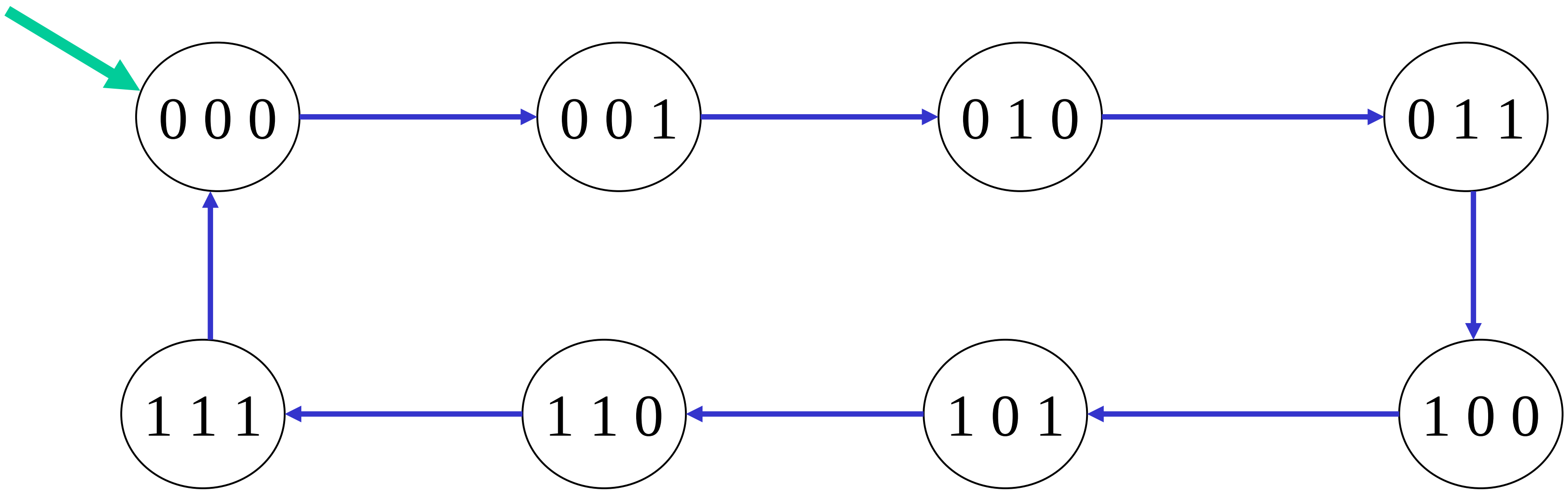
$\mathbf{R_0}$ = $\mathbf{v_0'} \neq \mathbf{v_0}$ $\mathbf{v_0}$ – current value $\mathbf{v_0'}$ – next value

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

But this is not a transition!

$\mathbf{\{t_0, t_1, t_2, t_3\} \subseteq R_0}$ but $\mathbf{t_3 \notin R_2}$

Transition relation predicate



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

$R_1 = v_1' = (v_0 \oplus v_1)$

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

$R = R_0 \wedge R_1 \wedge R_2$

Summary of Predicates

- System variables $\mathbf{v_0}, \mathbf{v_1}, \mathbf{v_2}, \ldots\ldots, \mathbf{v_n}$.
- Each $\mathbf{v_i}$ has a domain of values
  - Boolean , {a,b,c,..}, {5,8,0,7}…
  - We require that each domain be ***finite***.
- A state is a function **s** which assigns to each system variable a value in its domain.
- The set of states is ***finite*****.**

Summary

- Predicates can be used to pick out –succinctly- sets of states (useful for identifying initial states).
- **X** = **Formula**($\mathbf{v_0}$, $\mathbf{v_1}$, $\mathbf{v_2}$,…,$\mathbf{v_n}$)
- But this works well only when all domains are boolean.
- In general, we can use ***first order formulae***.

Summary

- A set of transitions can also be picked out using predicates.
- **T** = **Formula**(v_0, v_1,…, v_n, v_0', v_1',…,v_n')
- **T** is the set of all transitions

$$(v_0, v_1,\ldots,v_n) \longrightarrow (v_0', v_1',\ldots,v_n')$$

such that Formula (above!) is satisfied.

- Not all (state or transition) formulas will be legitimate.

Why use formulae?

- ***Formulae*** allow us to compactly describe a system and its dynamics
- It's easy to go from a "***logical***" description to ***Kripke structures***.
- Once we have a ***Kripke structure***, we are in business.
- We can use
  - ***Temporal Logics*** to specify properties
  - ***Model checking*** to verify these properties.

First Order Logic

- The general structure :
  - **Syntax**
    - Formulae
  - **Semantics**
    - When is a formula true?
    - Models
      - Interpretations
      - Valuations

Syntax

- **Terms**
  - Variables
  - Functions symbols, constant symbols
- **Atomic formulas**
  - Relation symbols, equality, terms
- **Formulas**
  - Atomic formulas
  - Propositional connectives
  - ***Existential and universal quantifiers***

Syntax

- (individual) variables --- **x, y, v_3, v'**,…
  - System variables in our context
- Function symbols : $\mathbf{f}^{(n)}$
  - **n** is the arity of **f**.
  - Add $^{(2)}$
  - Next $^{(1)}$
- Function symbols will capture the functions used in the programs, circuits, …

Constant symbols

- Apart from variables, it will also be convenient to have constant symbols.
  - ***zero*** , ***five***, ….
- Variables can be assigned different values but a constant symbol is assigned a fixed value.

Terms

- Terms are used to point at values.
- Any variable **v** is a term.
  - **x , v , v''**
- Any constant symbol ***c*** is a term.
- Suppose ***f*** is a function symbol of arity ***n*** and $t_1, t_2, \ldots, t_n$ are terms, then $f(t_1, t_2, \ldots, t_n)$ is a also term.

Terms

- Let Plus be a function symbol of arity 2.
- $\mathbf{v_1}$, $\mathbf{v_2}$, **Plus(**$\mathbf{v_2}$**, Plus(**$\mathbf{v_1}$**,** $\mathbf{v_1}$**))** are terms.
  – the semantics of the last term is intuitively

    $\mathbf{v_2 + 2v_1}$
- Let weird_op be a function symbol of arity 3
- Then

  Plus(weird_op(**v**, Plus(**$\mathbf{v_1}$**, **$\mathbf{v_2}$**), ***five***), Plus(**v**, **v”**))

  is a term.

Predicates

- Relation (predicate) symbols :
  - ***P*** which also has an arity
  - ***Greater-Than*** has arity 2
  - ***Prime*** has arity 1
  - ***Middle*** has arity 3 -- ***Middle*(t₁, x, t₂)**
    - intuitively, **x** lies between t_1 and t_2
- ***Equal*** has arity 2
  - will be denoted as **=**
  - It is a "**constant**" relation symbol.

Atomic formulas.

- If t_1 and t_2 are terms then $=(t_1, t_2)$ is an atomic formula.
  - also written $t_1 = t_2$
- Suppose P has arity **n** and $t_1, t_2, \ldots, t_n$ are terms.
- Then $P(t_1, t_2, \ldots, t_n)$ is an atomic formula.

Atomic formulas

- ***Greater-Than***(***five***, ***zero***)
- ***Greater-Than***(***two***, ***four***)
- ***Prime*****(Plus(v_1, v”))**
- **Plus(v,Zero) = weird_op(v,v,four)**
- **v =** ***Greater_Than*****(v_1,v_2)** is ***not*** an atomic formula !

Terms and Predicates

- A ***term*** is meant to denote a domain value.
  - **It makes no sense to talk about a term being true or false**.
- An ***atomic formula*** may be ***true*** or ***false*** (depends on the interpretation).
  - **It does not make sense to associate a domain value with an atomic formula**.

Formulas

- Every atomic formula is a formula.
- If φ is a formula then $\neg\varphi$ is a formula.
- If φ and φ' are formulas then $\varphi \vee \varphi'$ is a formula.
- $\varphi \wedge \varphi'$ abbreviates: $\neg(\neg\varphi \vee \neg\varphi')$
- $\varphi \supset \varphi'$ abbreviates : $\neg\varphi \vee \varphi'$
- $\varphi \equiv \varphi'$ abbreviates : $(\varphi \supset \varphi') \wedge (\varphi' \supset \varphi)$

Formulas

- If φ is a formula and $\mathbf{x}$ is a variable then $\exists \mathbf{x}.\, \varphi$ is a formula.
- $\forall \mathbf{x}.\, \varphi$ abbreviates : $\neg\exists \mathbf{x}.\, \neg\varphi$
- These are ***existential*** and ***universal*** quantifiers.
- The power of first order logic comes from these operators!

Semantics

- **Models** :
  - ***Domain of interpretation***
  - ***Interpretation***
    - For the function, constant and relation symbols.
      - ***Fixed for all formulas***.
    - For the individual variables, on a “per formula” basis.
      - ***Valuations***.

Semantics

- ***Domain***
  - Each variable will have its domain of values.
  - We pretend all these domains are the same.
  - Or rather, a big enough "universe" that will contain all these domains.
- Fix **D** the universe of values.

Semantics

Interpretation function I

- Assign a concrete function to each function symbol (of the same arity!)
- Assign a concrete member of **D** to each constant symbol.
- Assign a concrete relation to each relation symbol (of the same arity!).

Semantics

- Assume we have fixed an interpretation for all function symbols, constant symbols and relational symbols.
- Let φ be a formula. Fix a ***valuation*** (or ***assignment***) **V** which assigns a member of **D** to each variable.
- **V** : **Var** $\longrightarrow$ **D**

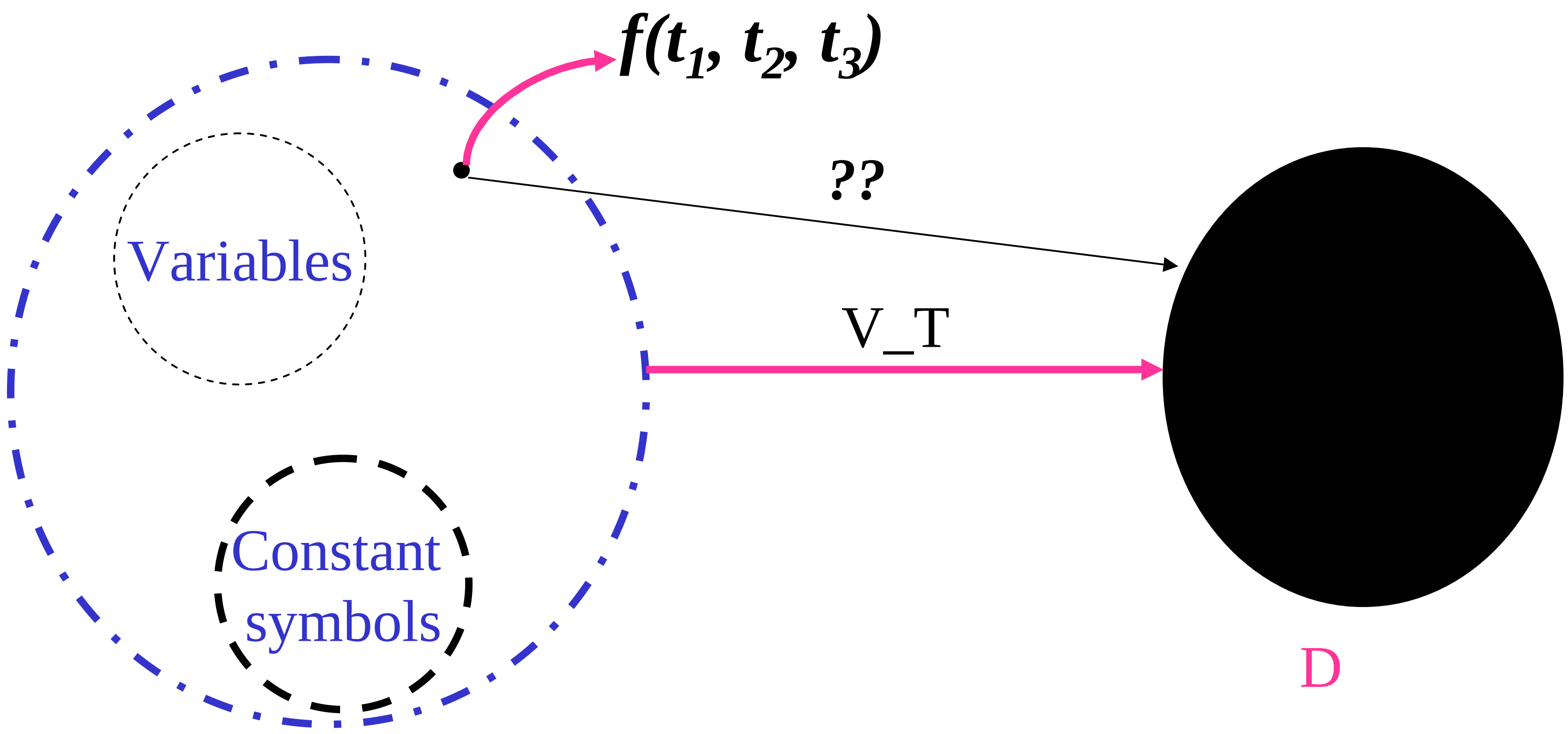
Lift V to All Terms

- We have :
  - An ***interpretation*** for the function symbols and constant symbols.
  - An ***assignment*** **V** : **Var** ⟶ **D**
- Using these, we can construct (uniquely!)

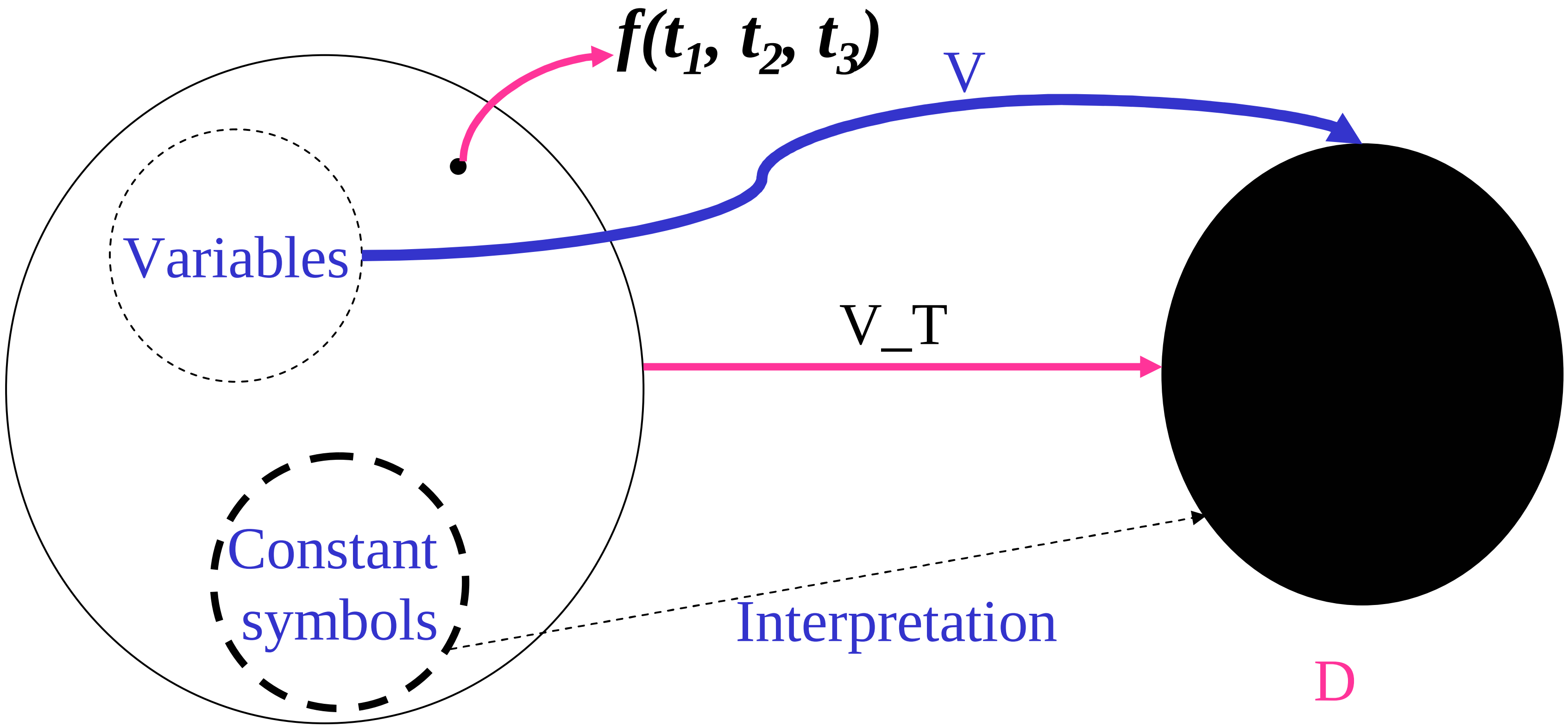
  V_T : Terms ⟶ **D**

  the interpretation of terms!

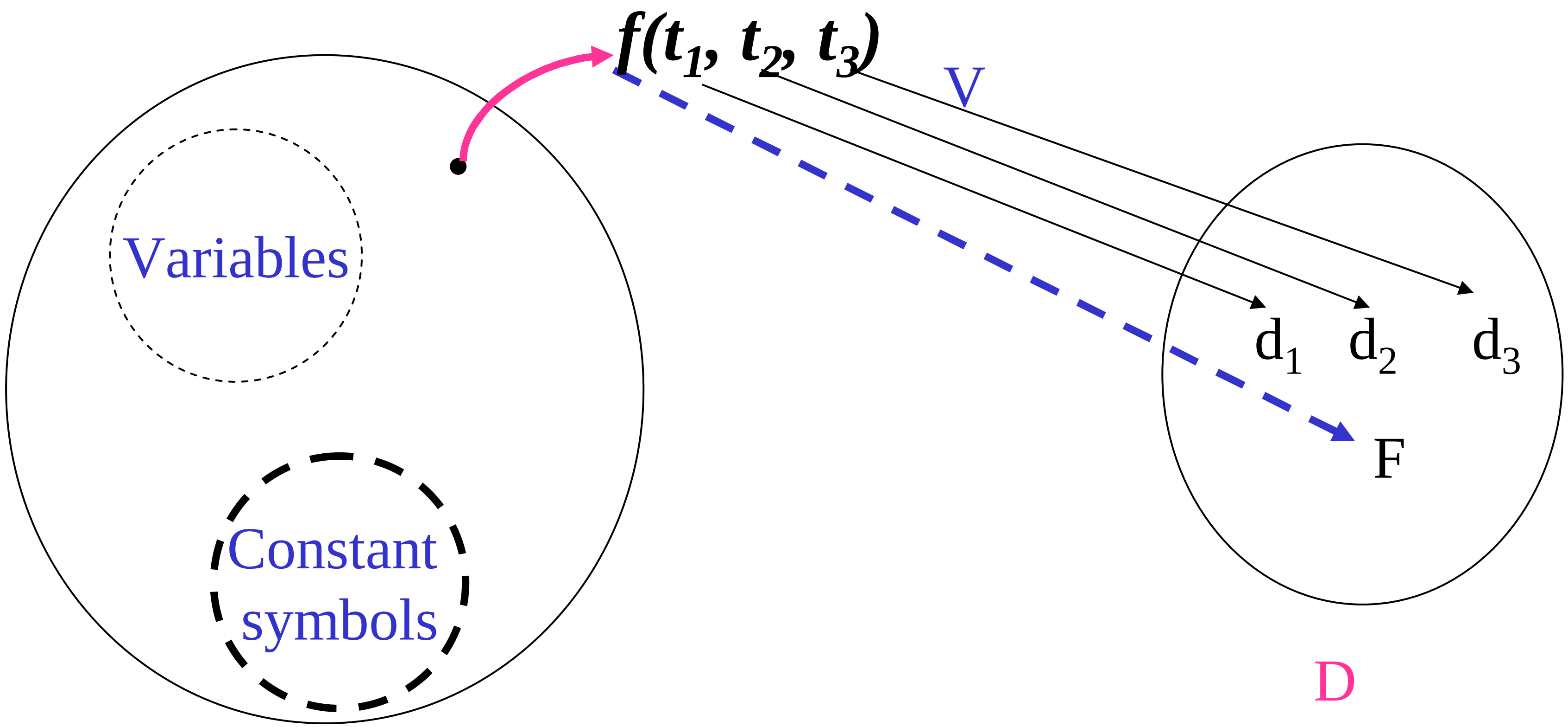
Constructing V_T



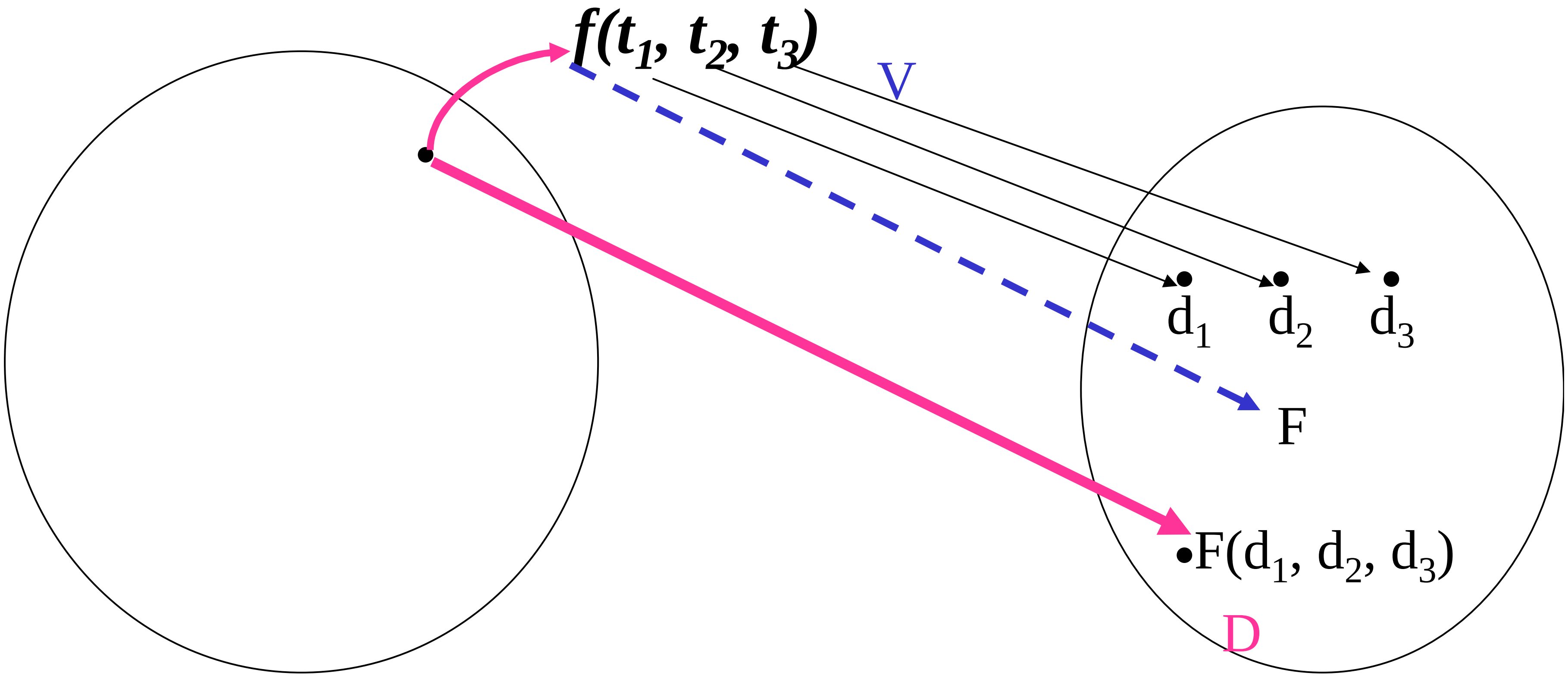
Constructing V_T



Constructing V_T



Constructing V_T



Semantics

- Let φ be a formula. Fix a valuation **V** which assigns a member of **D** to each variable.
- So we now have **V_T** that assigns a member of **D** to each term.
- φ is satisfied under **V** (and the interpretation we have fixed, for all formulae) if :

Semantics

- Suppose ***P(t₁, t₂,.., tₙ)*** is an atomic formula

  and **V_T**(t_1) = d_1, ….**V_T**(t_n) = d_n

  and PCON is the relation assigned to symbol ***P*** by our interpretation **I**.
- Then $P(t_1, t_2,.., t_n)$ is satisfied under **V** iff

  PCON(d_1, d_2,…,d_n) holds in **D**, that is:

  $(d_1, d_2, \ldots, d_n) \in PCON \subseteq \mathbf{D} \times \mathbf{D} \times \ldots \times \mathbf{D}$

Semantics

- Suppose φ is of the form $\neg\varphi'$.

  Then φ is satisfied under **V** iff φ' is **not** satisfied under **V**.

- Suppose φ is of the form $\varphi_1 \vee \varphi_2$

  Then φ is satisfied under **V** iff φ_1 is satisfied under **V** **or** φ_2 is satisfied under **V**.

Semantics

- ***Greater-Than***(**Plus**(***v***, ***3***), **Multi**(***x***, ***2***))

  t_1 t_2

- V(v) = 2 V(x) = 1

  $V\_T(t_1) = 5$ $V\_T(t_2) = 2$

  $(5, 2) \in \; > \; \subseteq$ Integers × Integers

- V'(v) = 1 V'(x) = 6 and $V'\_T(t_1) = 3$ $V'\_T(t_2) = 12$

  $(3, 12) \notin \; > \; \subseteq$ Integers × Integers

- Under **V** the atomic formula is true, but under **V'** the atomic formula is not.

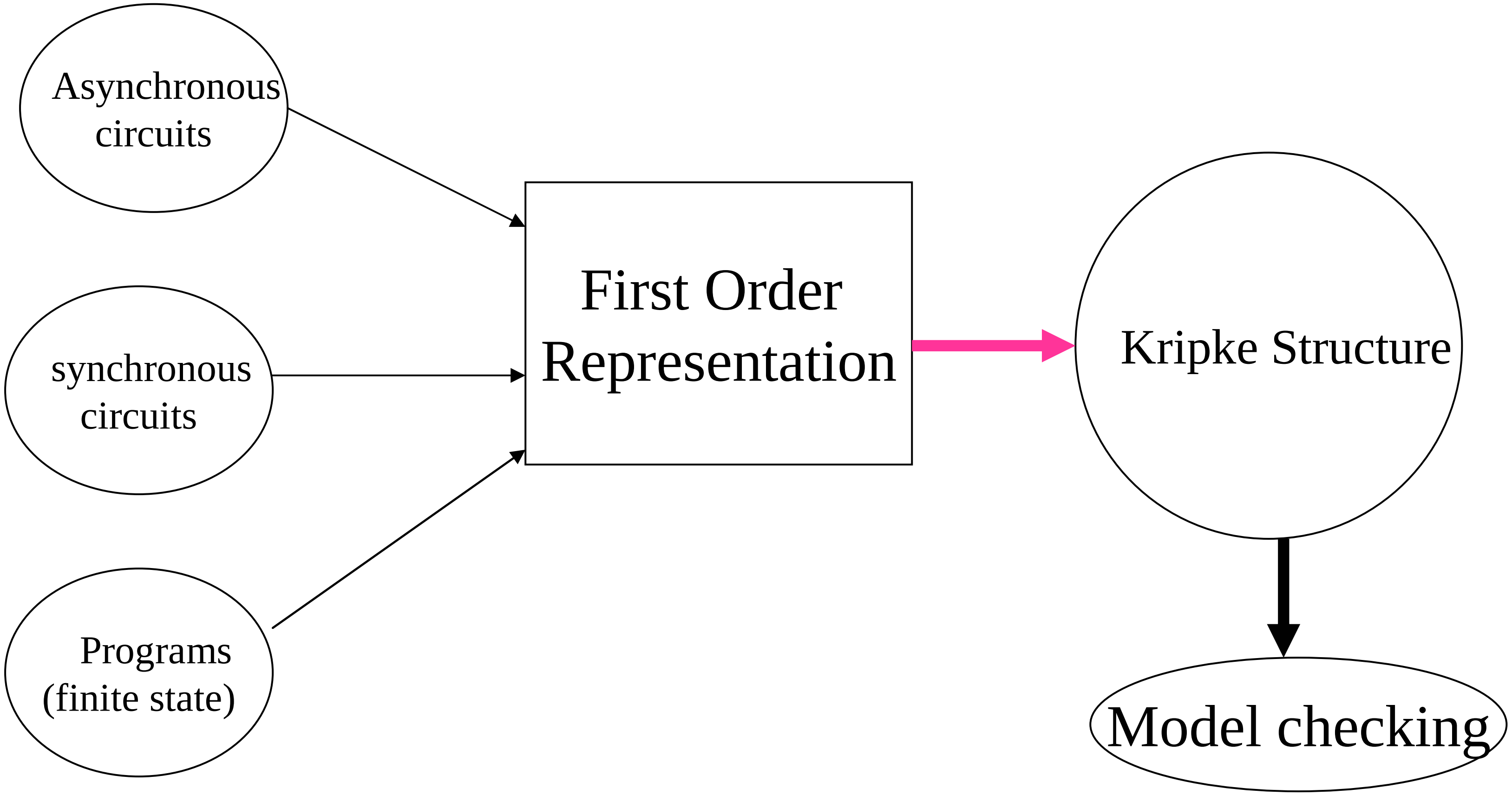
Semantics

- The only case left is when φ is of the form **$\exists$x.** φ**’**
- φ is satisfied under **V** iff there is a valuation **V’** such that φ**’** is satisfied under **V’** and **V’** is required to meet the condition:
  - **V’** is exactly **V** for all variables except **x**.
  - To **x** , **V’** can assign ***any value*** of its choosing.

Semantics

- Whether **$\exists x.\ \varphi$** is true or not under **V**
  - does not depend on what **V** does on **x** !
- **$\exists x.\ 2x = y$** is true under **V(y) = 4**, **V(x) = 1**!
- Because, we can find **V'**, with **V'(y) = 4** but **V'(x) = 2**.
- One says **x** is ***bound*** in the formula and **y** is ***free***.

The efficient way



First Order Representation to 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
  $s(v_1) \in D_1$ …..
- **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:

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

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 **$v_1 = d_1,\ldots,v_n = d_n, v_1' = d_1',\ldots v_n' = d_n'$**.

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

Notice that the formula above is equivalent to:

- **$pc = l_5 \wedge pc' = l_6 \wedge ((\exists n.(x=2n) \wedge x'=x/2) \vee (\neg\exists n.(x=2n) \wedge x'=x-1)) \wedge same(y, z)$**
- **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!

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

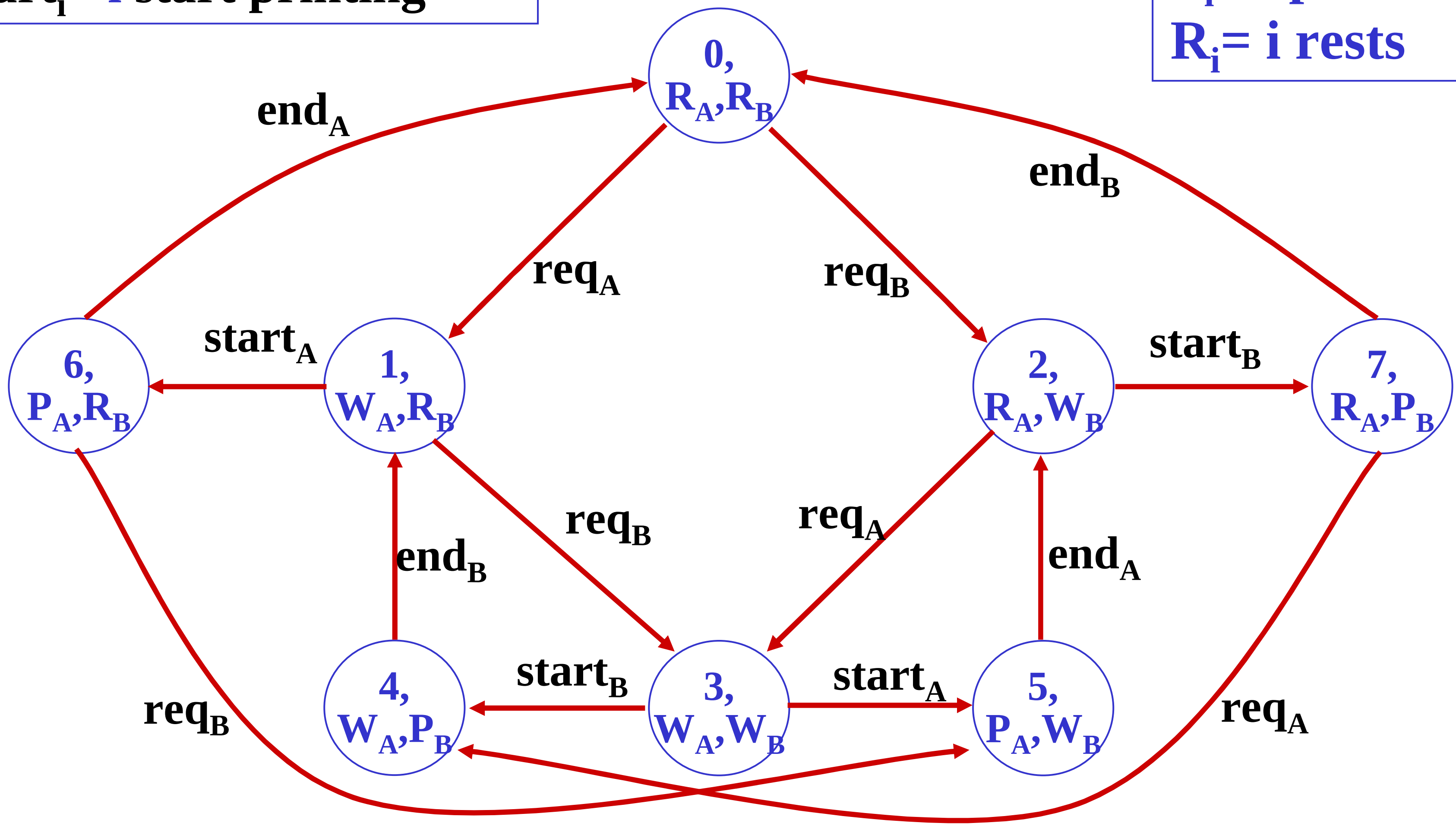
- **AP** - set of elementary property
- **<S,A,R,s_0,L>**
- **S** - set of states
- **A** - set of transition labels
- **R $\subseteq$ S×A×S** - (labeled) transition relation
- **L** - interpretation mapping **$L:S \longrightarrow 2^{AP}$**
- **In *FO representation* we would need two sets of variables: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

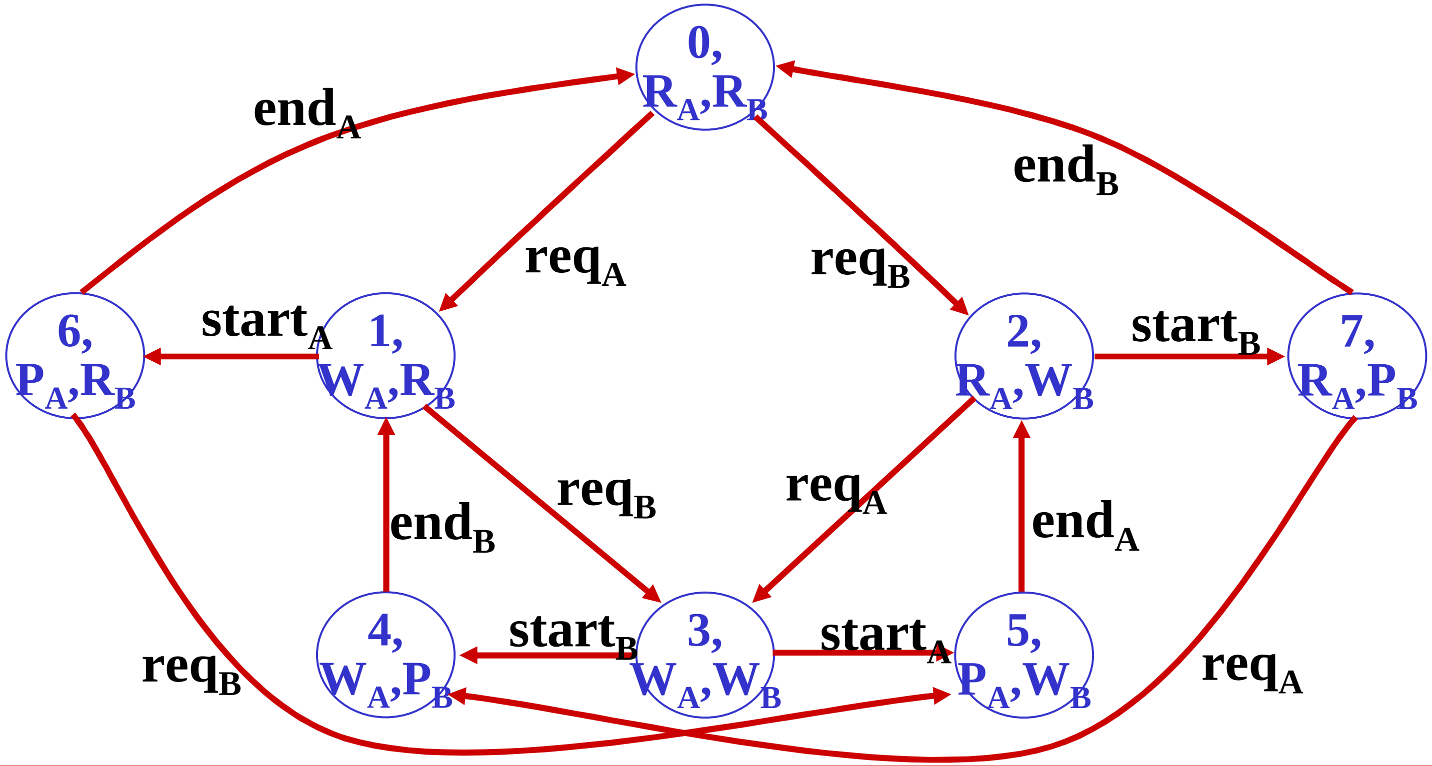
Example: a print manager

end_i = i ends printing
req_i = i requests printing
$start_i$= 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***.

Synchronization

- Usually complex systems are composed of a number of smaller ***subsystems*** (***modules***)
- It is natural to model the whole system starting from the models of the subsystems.
- And then define how they cooperate.
- There are many ways to define cooperation (***synchronization***).

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

$TS_i = <S_i, A_i, R_i, s_{i0}>$

The system is then defined as $TS=<S,A,R,s_0>$ where

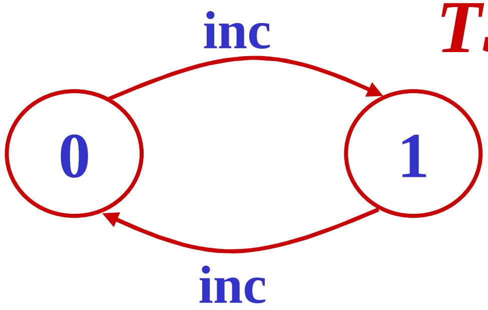
$$S = S_1 \times S_2 \times \ldots \times S_n$$

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

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

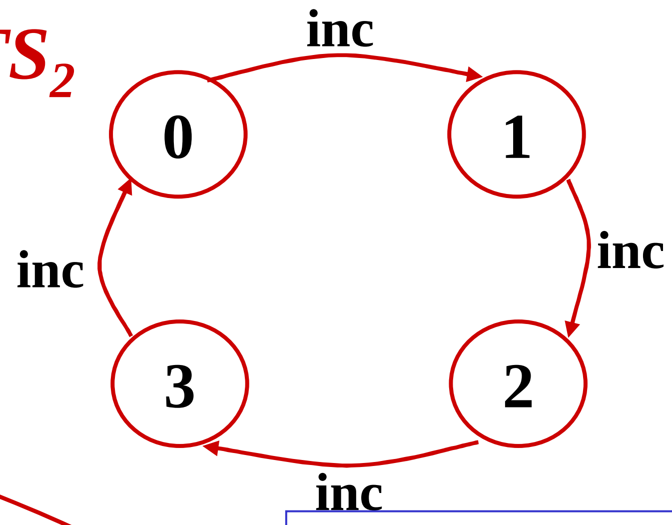
$$s_0 = <s_{10}, s_{20}, \ldots, s_{n0}>$$

TS_1

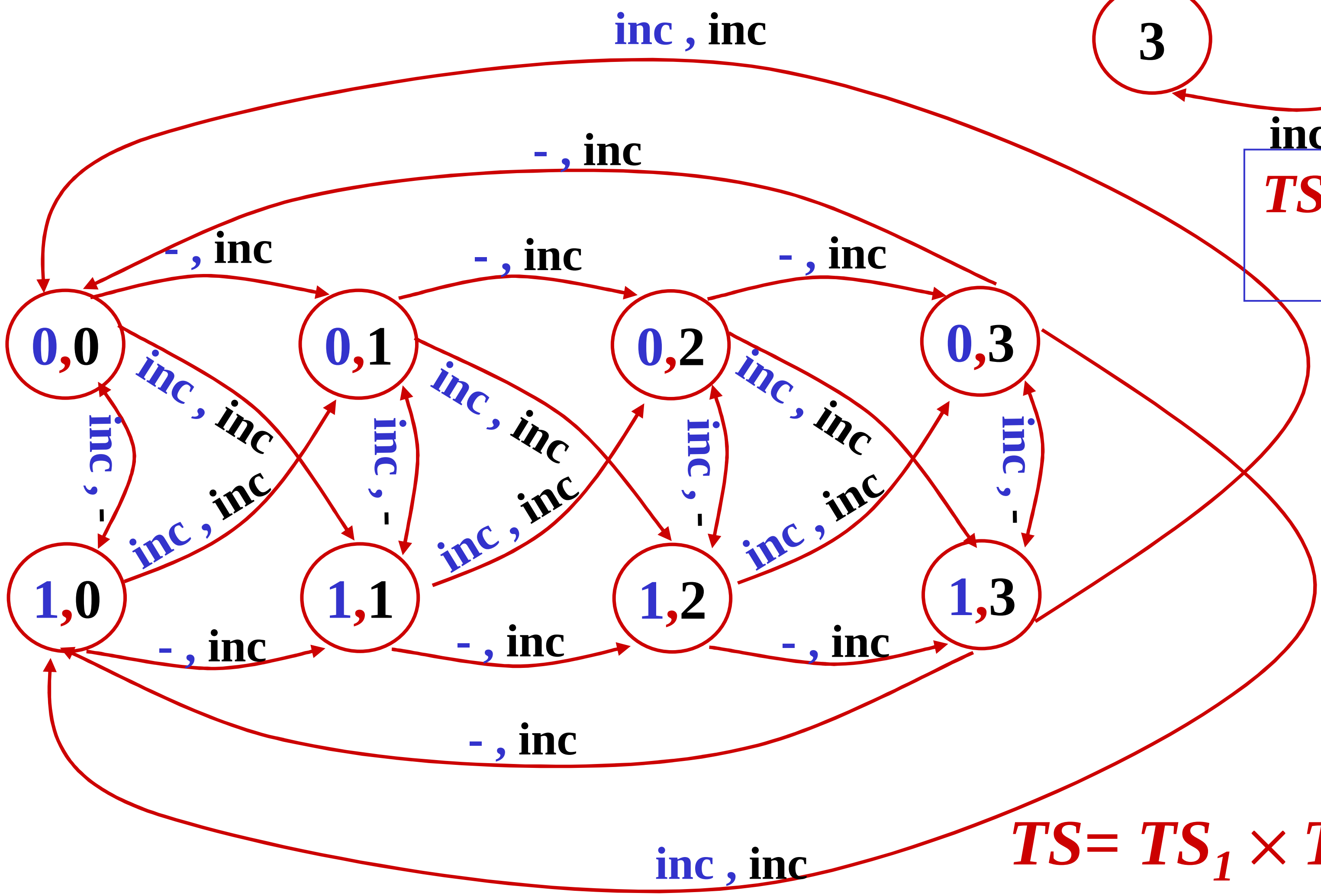


TS_1 counter modulo 2

TS_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:***

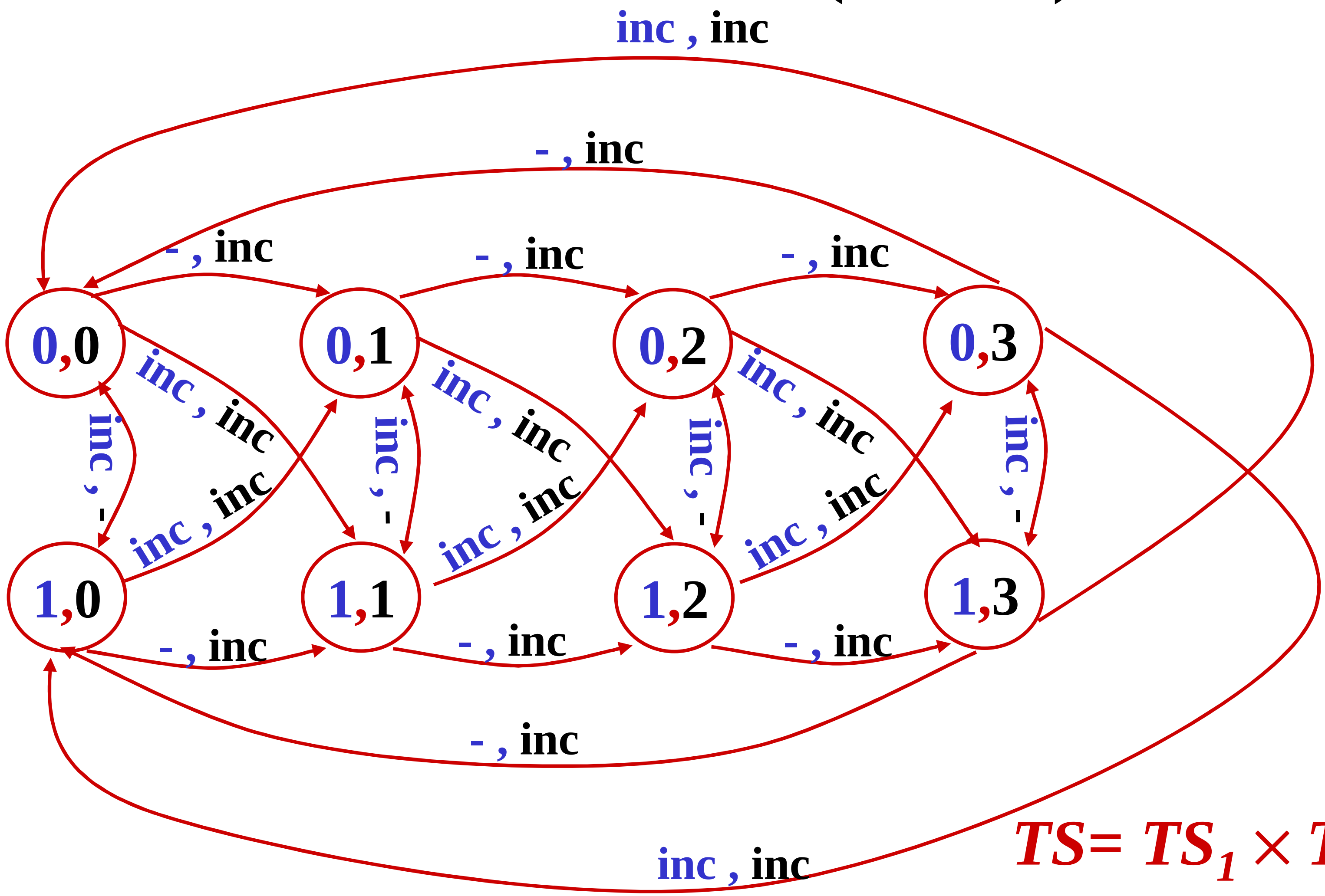
$$Sync \subseteq A_1 \cup \{-\} \times A_2 \cup \{-\} \times \ldots \times A_n \cup \{-\}$$

- ***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:

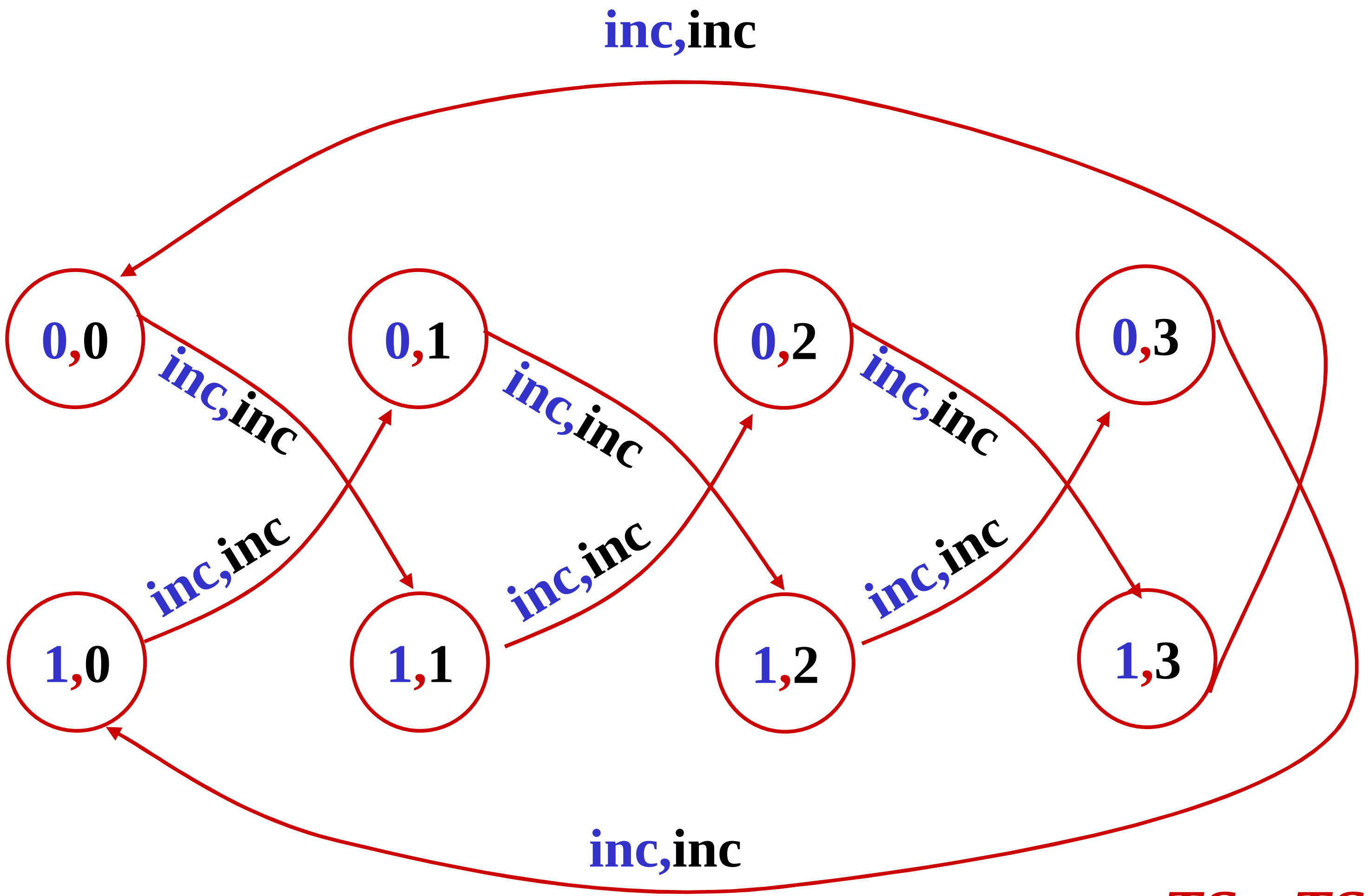
$Sync = \{inc, -\} \times \{-, inc\} \setminus \{(-,-)\}$

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

if one wants to *discard* the situation where *no component acts*

Synchronization on all actions (Synchronous systems):

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



***TS*= *TS*$_1$ × *TS*$_2$**

Synchronous systems

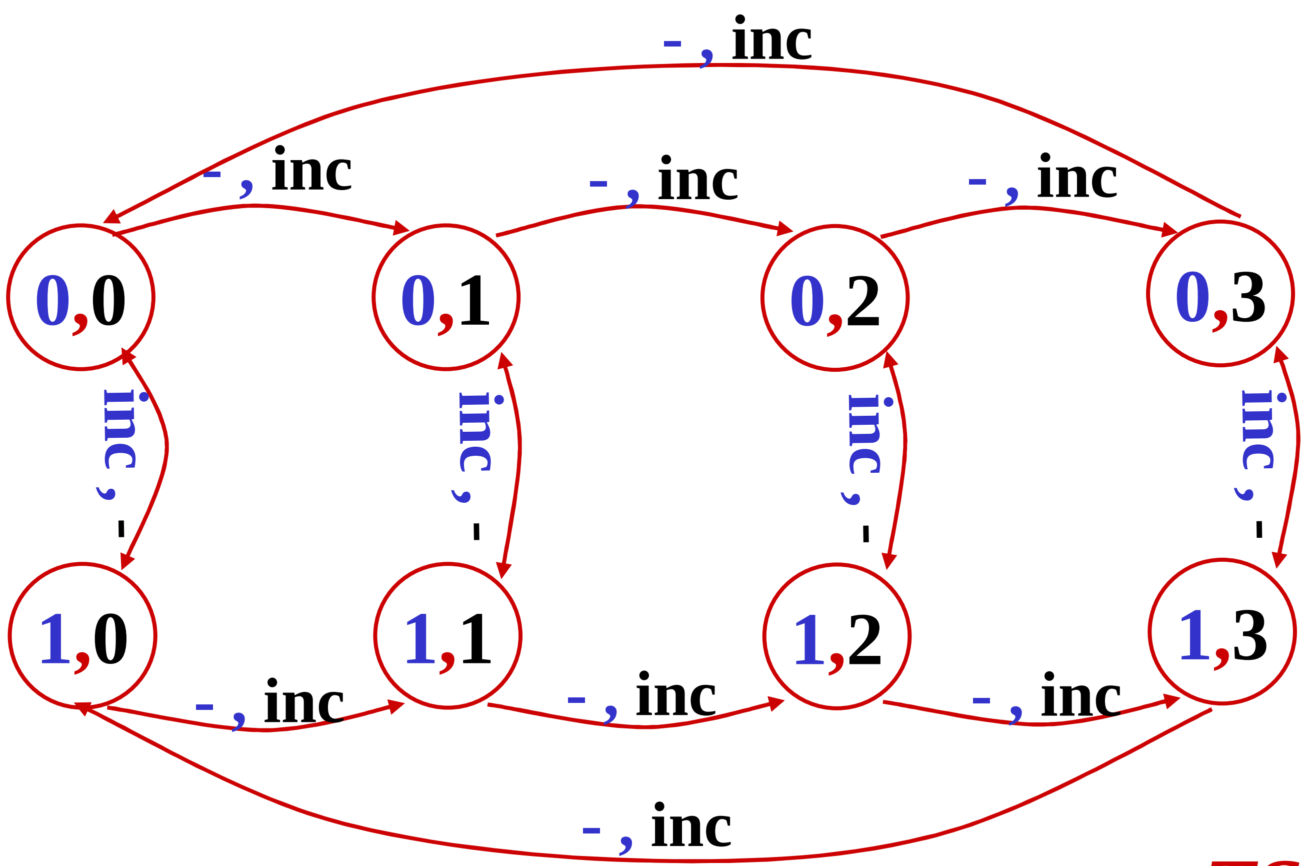
Synchronous systems:

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

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

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

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



TS= TS₁ × TS₂

Asynchronous systems: Interleaving

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

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

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

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 || …|| P_n) coend**
- **P_1, P_2,..P_n ---** Sequential Programs.
- ***Program variables*** set **$V = V_1 \cup \ldots \cup V_n$** (set **$V_i$** for program **i**)
- ***Program counters*** set **PC** (one for each program)
- ***Usually interleaving semantics is assumed***

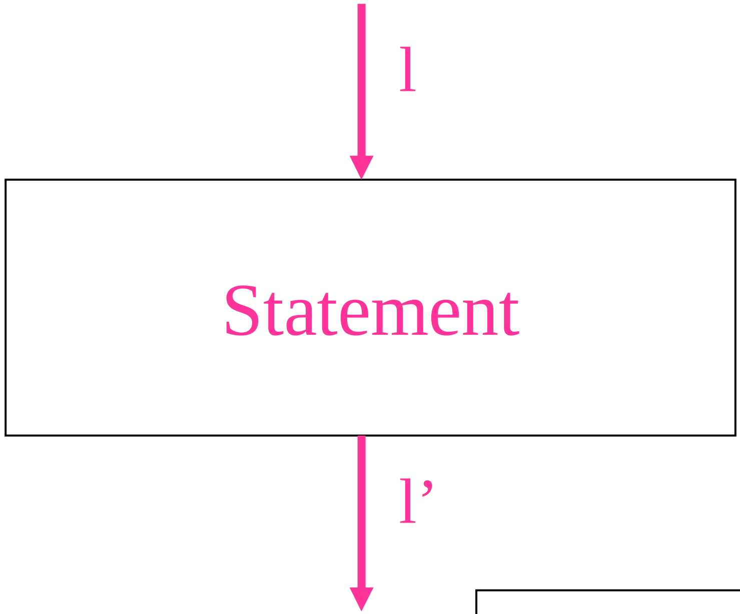
Program Statements

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**

Sequential Programs: the transition predicate C

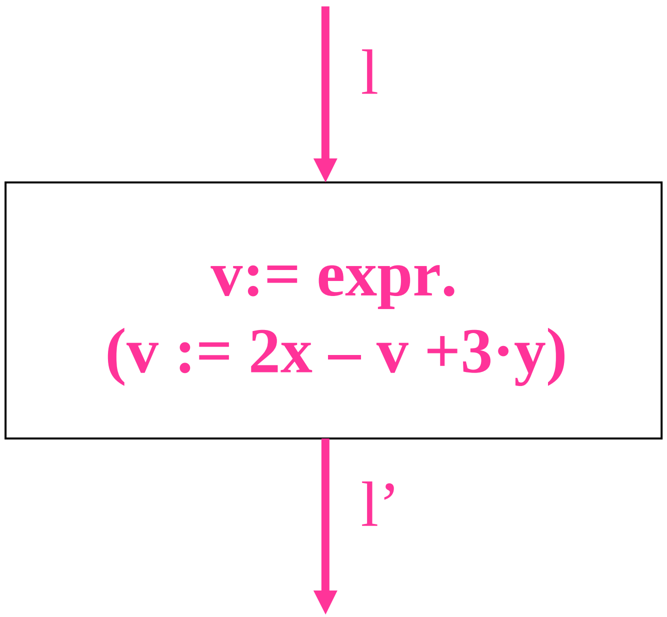
General Structure



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 for the statement.

Assignments



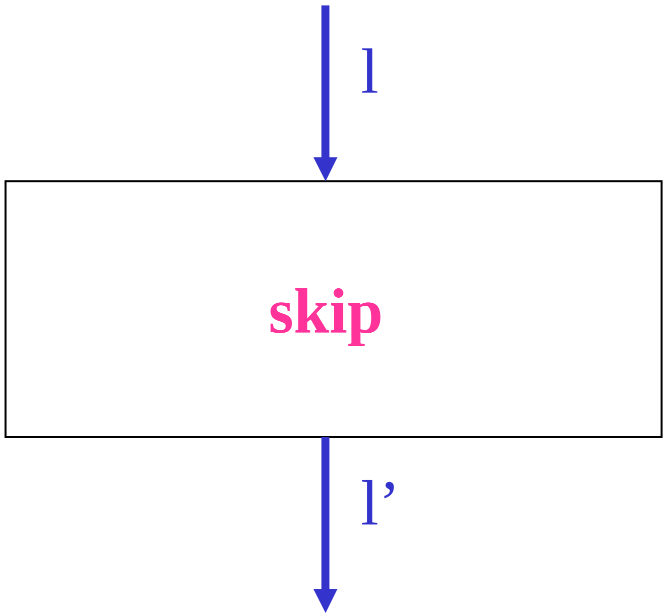
C(l, v:=expr., l')

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

[for $Y = \{y_1, y_2, ..y_m\}$,

$same(Y) \equiv y_1' = y_1 \wedge y_2' = y_2 \wedge \ldots \wedge y_m$]

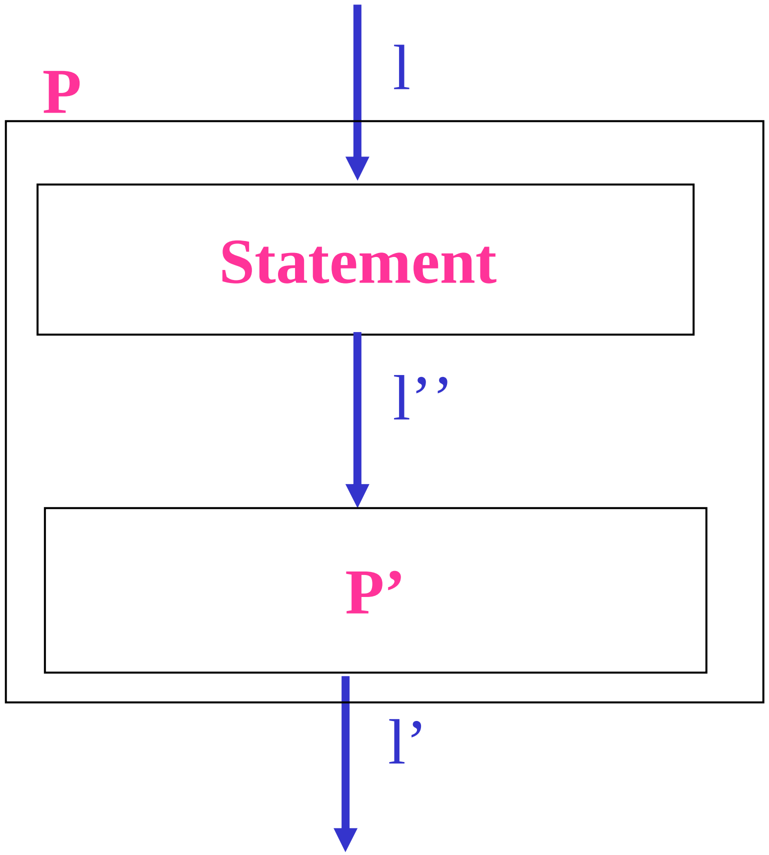
Skip



C(l, skip, l')

pc = l ∧ pc' = l' ∧ same (V) ∧ same (PC – {pc})

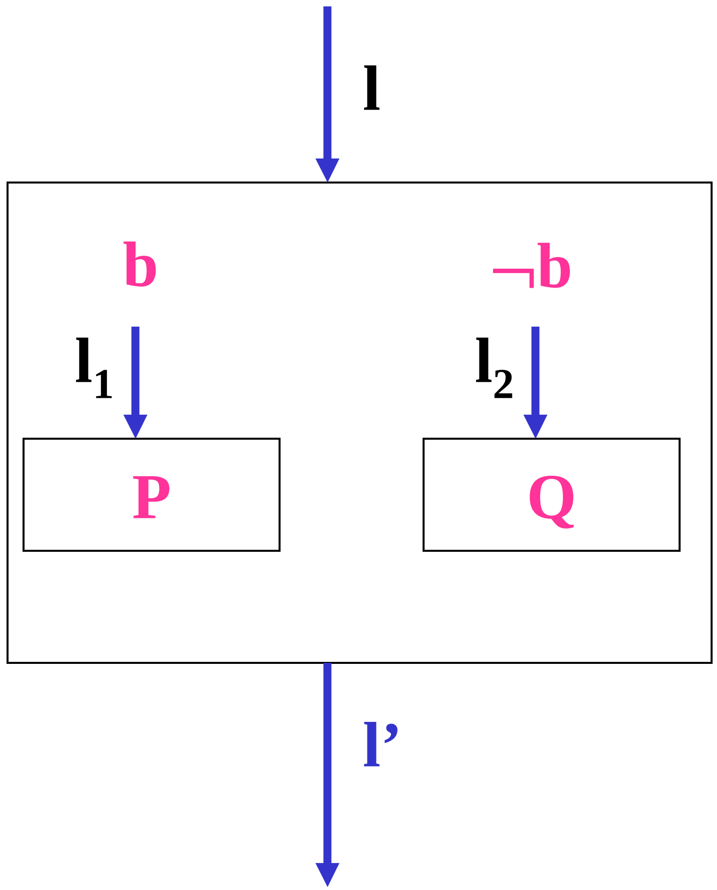
Sequential composition



C(l, **P** , l’)

C(l, **Statement** , l’’) ∨
C(l’’, **P’** , l’)

Conditional statement



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

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

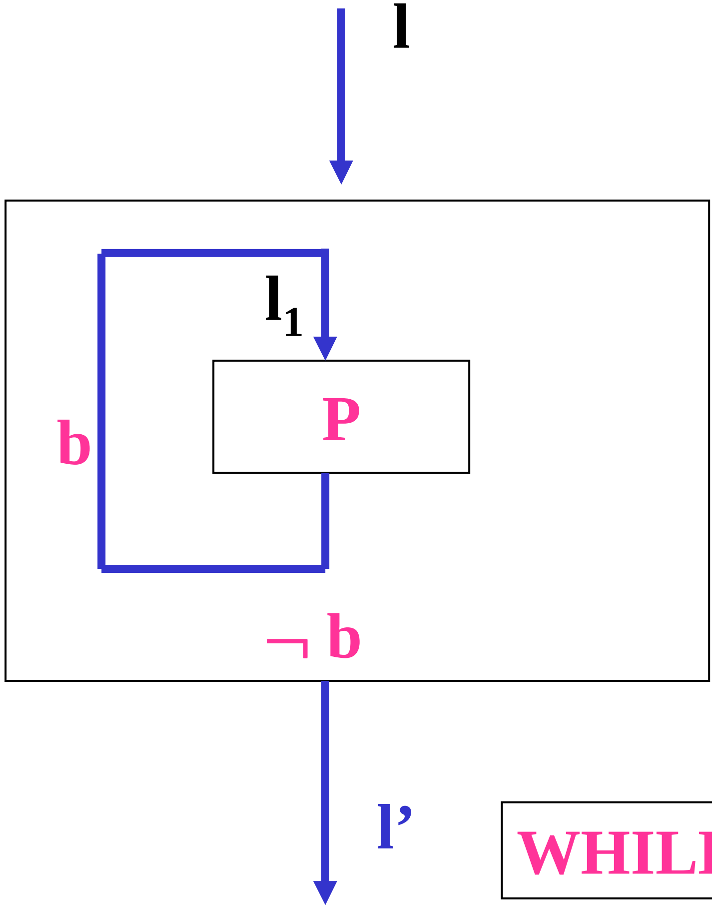
$(pc = l \wedge pc' = l_2 \wedge \neg b \wedge same(V) \wedge same(PC - \{pc\}) \vee$

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

$C(l_2, Q, l')$

IF b THEN P ELSE Q FI

While statement



$C(l, WHILE(b, l_1), l')$

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

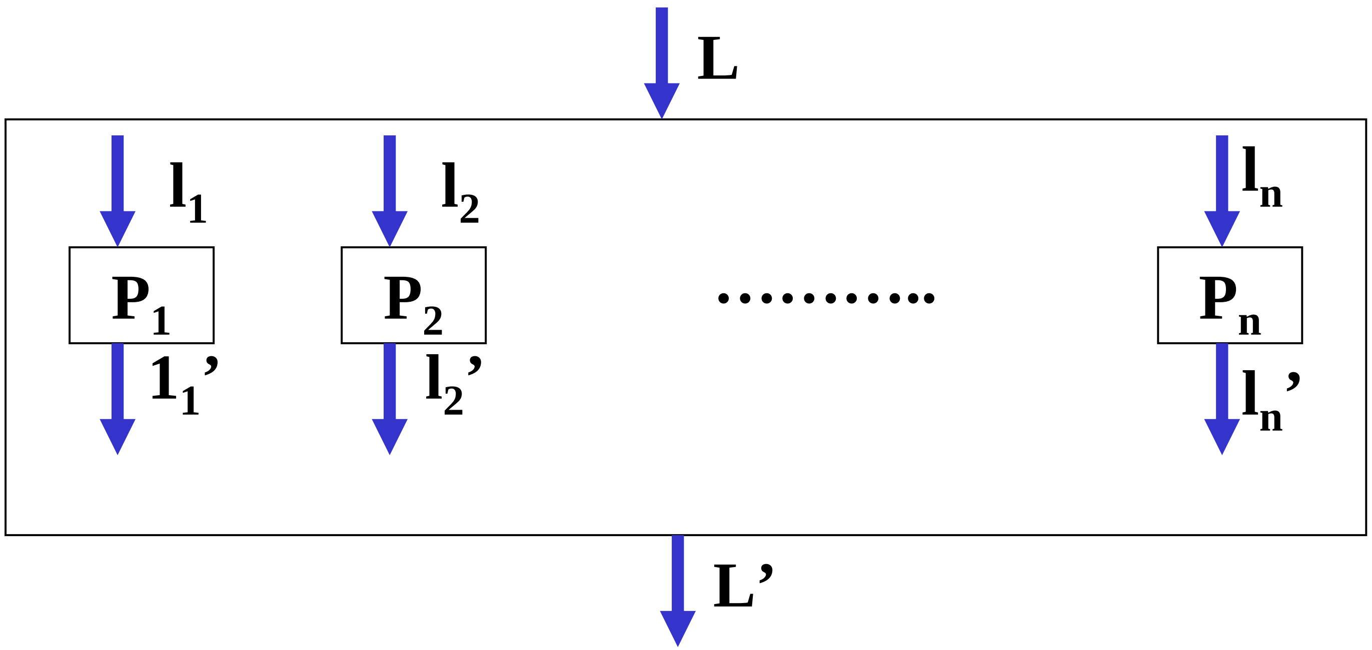
$(pc = l \wedge pc' = l' \wedge \neg b \wedge same(V) \wedge$
$same(PC - \{pc\}) \vee$

$C(l_1, P, l)$

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

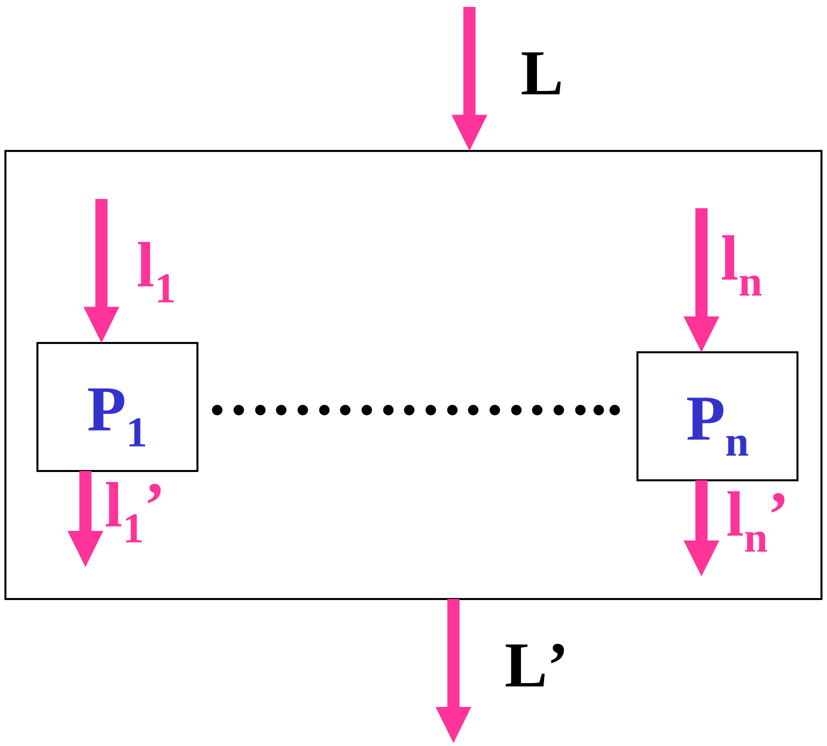
- **P** = **cobegin** **(P_1 || P_2 || …|| P_n)** **coend**
- **P_1, P_2,..P_n** --- ***Sequential Programs***.
- **$C(l_1, P_1, l_1')$** --- The transitions of program **P_1** (defined ***inductively*** on the structure of **P_1**!).
- **V_i** ---- The set of variables of program **P_i**.
- Programs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

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

$$\vee$$

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

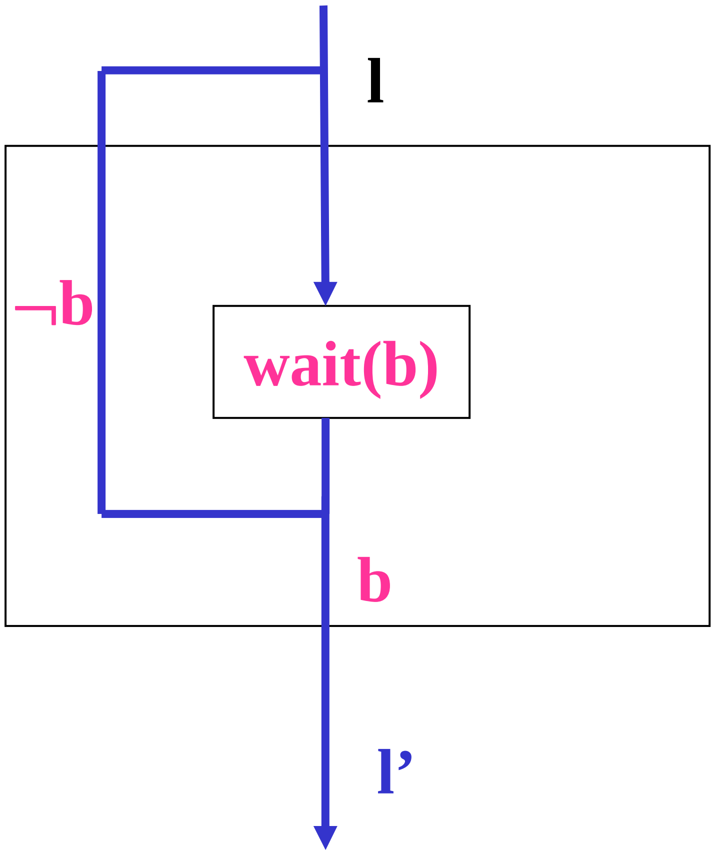
$$\vee \ldots \vee$$

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

$$\vee$$

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

The Transition Predicate

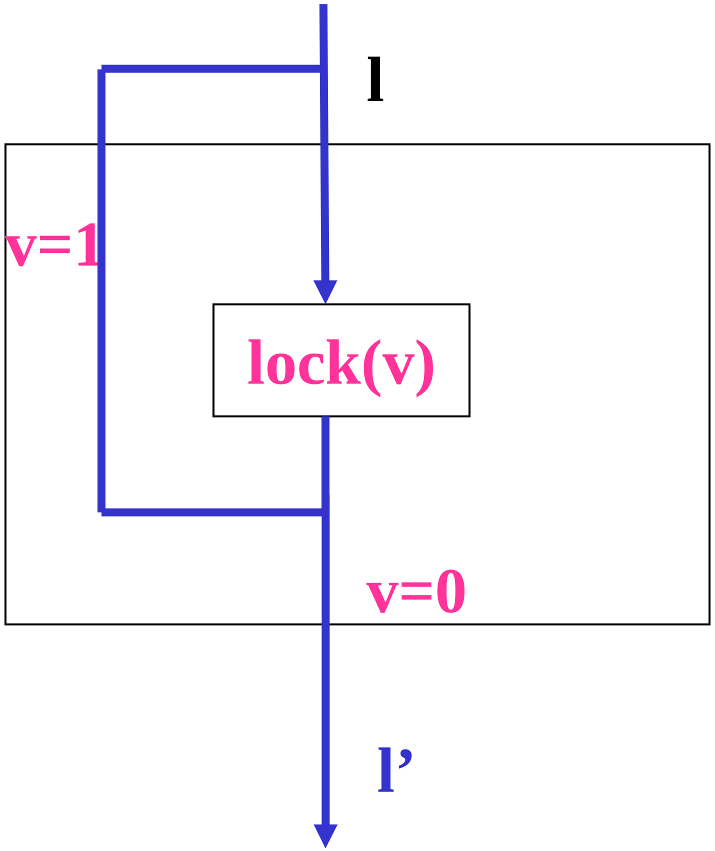


C(l, wait(b) , l')

$$(pc_i = l \land pc_i' = l \land \neg b \land same(V_i))$$
$$\lor$$
$$(pc_i = l \land pc_i' = l' \land b \land same(V_i))$$

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')

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

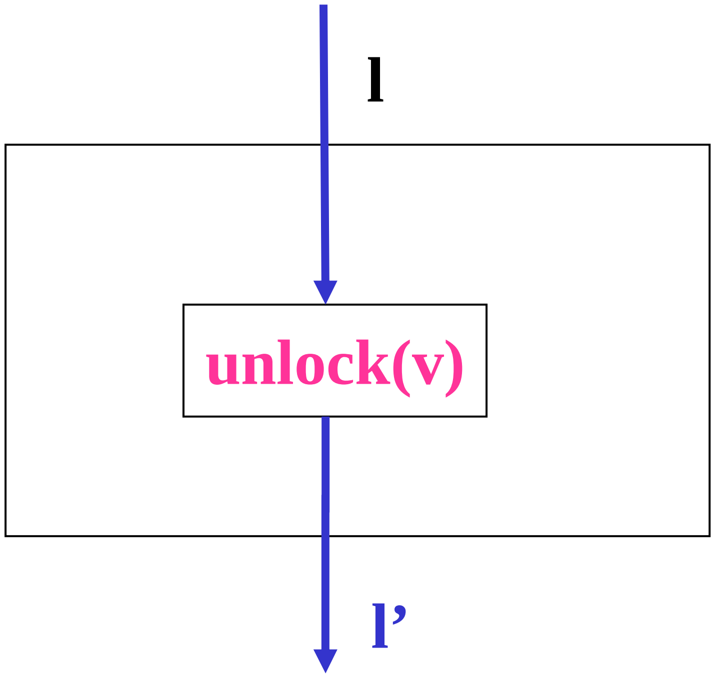
$$\vee$$

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

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