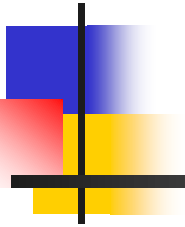


# Model Checking

## Slides for MarktOberdorf Summer School 2010



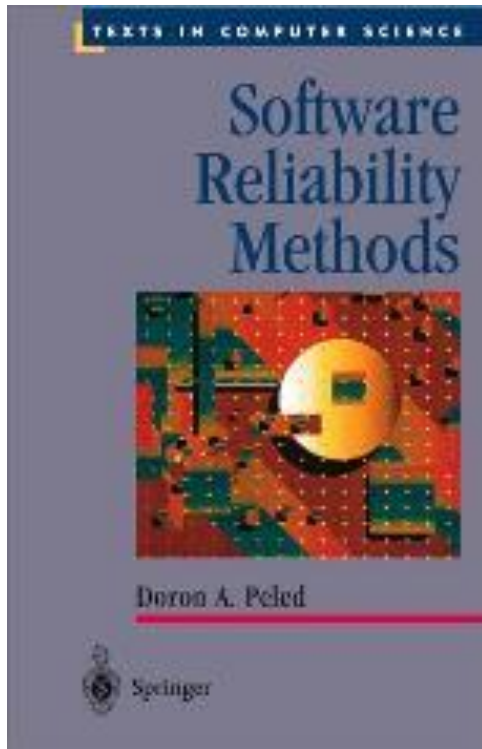
Doron A. Peled  
Bar Ilan  
University,  
Israel



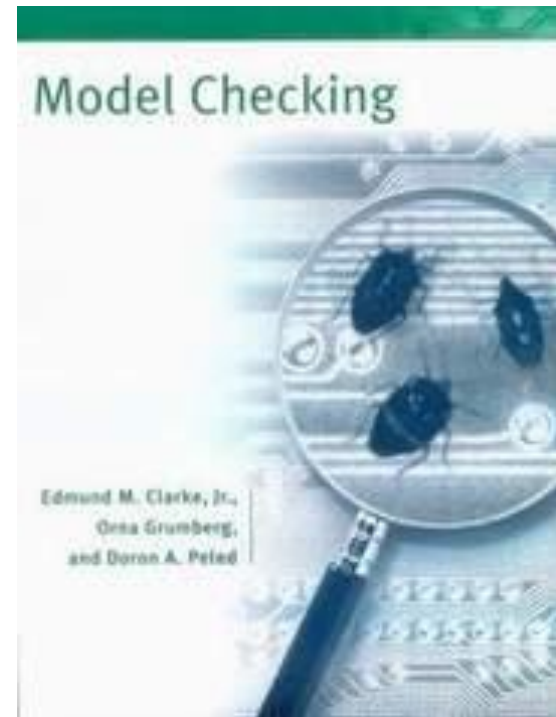
Version 2010

# Some related books:

Mainly:



Also:



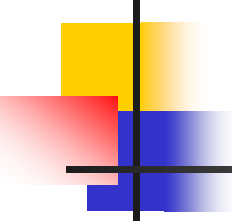
<http://www.dcs.warwick.ac.uk/~doron>



# Model Checking

---

- Modeling Systems
- Obtaining Specification (or formalizing them).
- Comparing model with specification algorithmically.
- Works for finite state systems + extensions.



# Verification Starts with Modeling!

## Sequential systems.

---

- Perform some computational task.
- Have some *initial condition*, e.g.,  
 $\forall 0 \leq i \leq n \ A[i] \text{ integer.}$
- Have some *final assertion*, e.g.,  
 $\forall 0 \leq i \leq n-1 \ A[i] \leq A[i+1].$   
(What is the problem with this spec?)
- Are supposed to terminate.



# Concurrent Systems

---

Involve several computation agents.

Termination may indicate an abnormal event (interrupt, strike).

May exploit diverse computational power.

May involve remote components.

May interact with users (Reactive).

May involve hardware components (Embedded).



# Problems in modeling systems

---

- Representing concurrency:
  - Allow one transition at a time, or
  - Allow coinciding transitions.
- Granularity of transitions.
  - Assignments and checks?
  - Application of methods?
- Global (all the system) or local (one thread at a time) states.

# Modeling.

## The states-based model.

- $V = \{v_0, v_1, v_2, \dots\}$  - a set of variables, over some domain.
- $p(v_0, v_1, \dots, v_n)$  - a parameterized assertion, e.g.,  
 $v_0 = v_1 + v_2 \wedge v_3 > v_4$ .
- A **state** is an assignment of values to the program variables. For example:  
 $s = \langle v_0 = 1, v_1 = 3, v_3 = 7, \dots, v_{18} = 2 \rangle$
- For predicate (first order assertion)  $p$ :  
 $p(s)$  is  $p$  under the assignment  $s$ .  
Example:  $p$  is  $x > y \wedge y > z$ .  $s = \langle x = 4, y = 3, z = 5 \rangle$ .  
Then we have  $4 > 3 \wedge 3 > 5$ , which is *false*.



# State space

---

- The **state space** of a program is the set of *all possible states* for it.
- For example, if  $V = \{a, b, c\}$  and the variables are over the naturals, then the state space includes:  
 $\langle a=0, b=0, c=0 \rangle, \langle a=1, b=0, c=0 \rangle,$   
 $\langle a=1, b=1, c=0 \rangle, \langle a=932, b=5609, c=6658 \rangle \dots$





# Atomic Transitions

---

- Each atomic transition represents a small piece of code such that no smaller piece of code is observable.
- Is  $\mathbf{a} := \mathbf{a} + 1$  atomic?
- In some systems, e.g., when  $\mathbf{a}$  is a register and the transition is executed using an *inc* command.

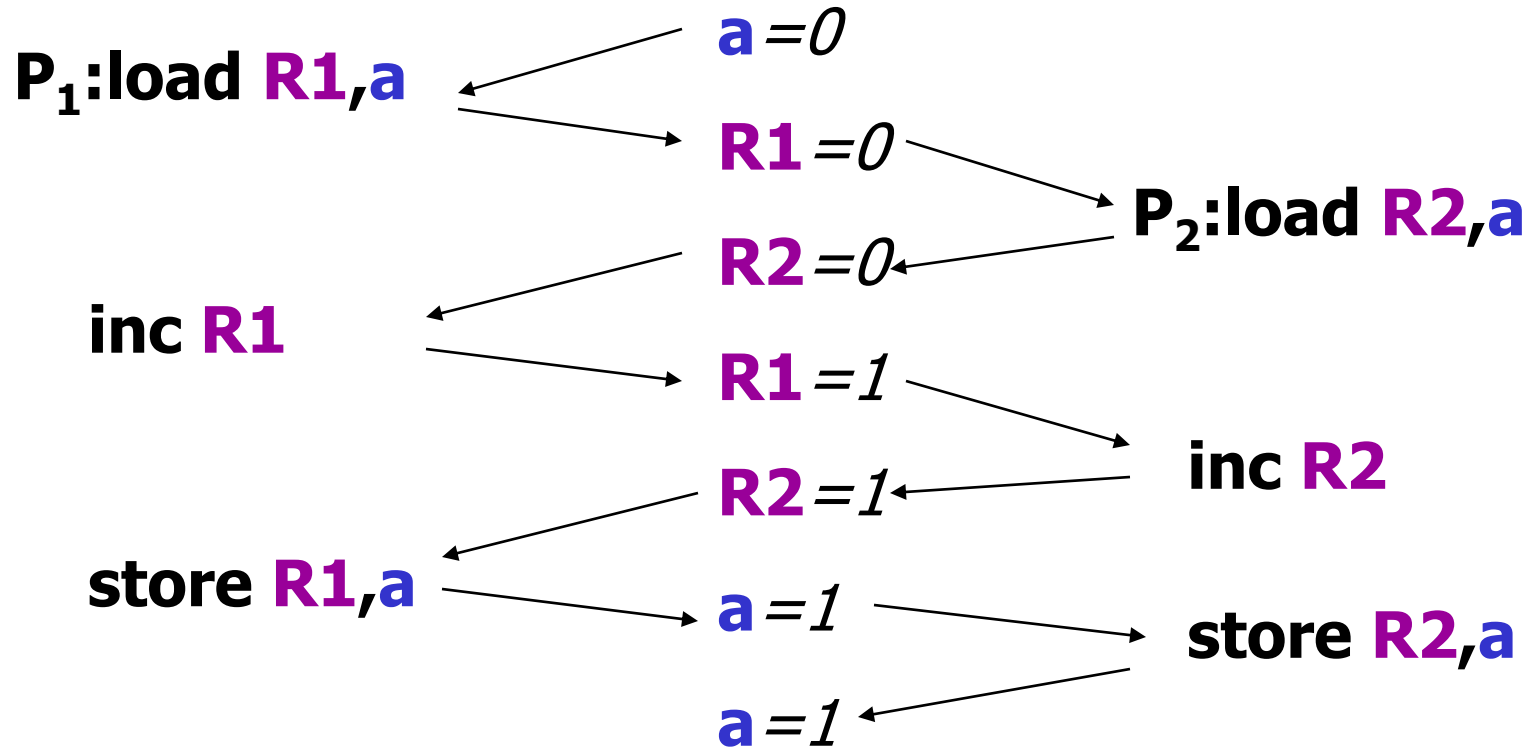
# Non atomicity

- Execute the following when  $a=0$  in two concurrent processes:
  - $P_1:a=a+1$
  - $P_2:a=a+1$
  - Result:  $a=2$ .
  - Is this always the case?
- Consider the actual translation:
  - $P_1$ :load  $R1,a$
  - inc  $R1$
  - store  $R1,a$
  - $P_2$ :load  $R2,a$
  - inc  $R2$
  - store  $R2,a$
  - $a$  may be also 1.



# Scenario

---





# Representing transitions

---

- Each transition has two parts:
  - The enabling condition: a predicate.
  - The transformation: a multiple assignment.

- For example:

$$a > b \rightarrow (c, d) := (d, c)$$

This transition can be executed in states where  $a > b$ . The result of executing it is switching the value of  $c$  with  $d$ .



# Initial condition

---

- A predicate  $I$ .
- The program can start from states  $s$  such that  $I(s)$  holds.
- For example:  
 $I(s) = a > b \wedge b > c$ .



# A transition system

---

- A (finite) set of variables  $V$  over some domain.
- A set of states  $\Sigma$ .
- A (finite) set of transitions  $T$ , each transition  $e \rightarrow t$  has
  - an enabling condition  $e$ , and
  - a transformation  $t$ .
- An initial condition  $I$ .



# Example

---

- $V = \{a, b, c, d, e\}$ .
- $\Sigma$ : all assignments of natural numbers for variables in  $V$ .
- $T = \{c > 0 \rightarrow (c, e) := (c - 1, e + 1),$   
 $d > 0 \rightarrow (d, e) := (d - 1, e + 1)\}$
- $I: c = a \wedge d = b \wedge e = 0$
- What does this transition system do?



# The interleaving model

---

- An **execution** is a *maximal* finite or infinite sequence of states  $s_0, s_1, s_2, \dots$   
That is: finite if nothing is enabled from the last state.
- The first state  $s_0$  satisfies the initial condition, I.e.,  $I(s_0)$ .
- Moving from one state  $s_i$  to its successor  $s_{i+1}$  is by executing a transition  $e \rightarrow t$ :
  - $e(s_i)$ , i.e.,  $s_i$  satisfies  $e$ .
  - $s_{i+1}$  is obtained by applying  $t$  to  $s_i$ .



# Example:

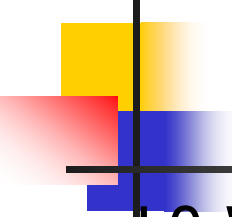
$T = \{c > 0 \rightarrow (c, e) := (c - 1, e + 1)$

$d > 0 \rightarrow (d, e) := (d - 1, e + 1)\}$

$I: c = a \wedge d = b \wedge e = 0$

- $s_0 = \langle a=2, b=1, c=2, d=1, e=0 \rangle$
- $s_1 = \langle a=2, b=1, c=1, d=1, e=1 \rangle$
- $s_2 = \langle a=2, b=1, c=1, d=0, e=2 \rangle$
- $s_3 = \langle a=2, b=1, c=0, d=0, e=3 \rangle$

# The transitions



---

```
L0:While True do
  NC0:wait(Turn=0);
  CR0:Turn=1
endwhile ||
L1:While True do
  NC1:wait(Turn=1);
  CR1:Turn=0
endwhile
```

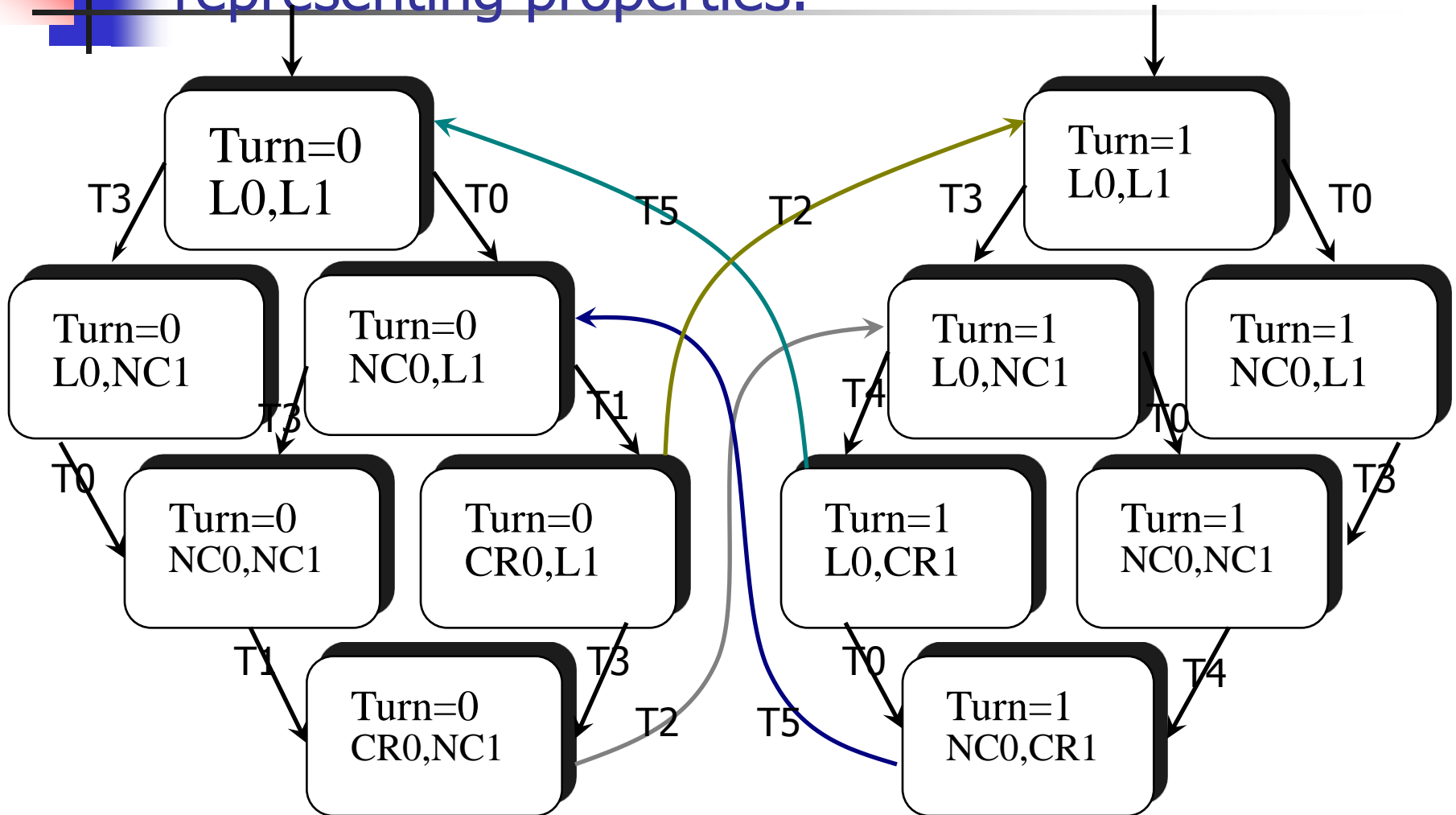
```
T0:PC0=L0 → PC0:=NC0
T1:PC0=NC0/\Turn=0 →
  PC0:=CR0
T2:PC0=CR0 →
  (PC0,Turn):=(L0,1)
T3:PC1=L1 → PC1=NC1
T4:PC1=NC1/\Turn=1 →
  PC1:=CR1
T5:PC1=CR1 →
  (PC1,Turn):=(L1,0)
```

Initially:  $PC0=L0 \wedge PC1=L1$

Is this the only reasonable way to model this program?

The state graph: Successor relation between *reachable* states.

Nodes are labeled with propositions representing properties.



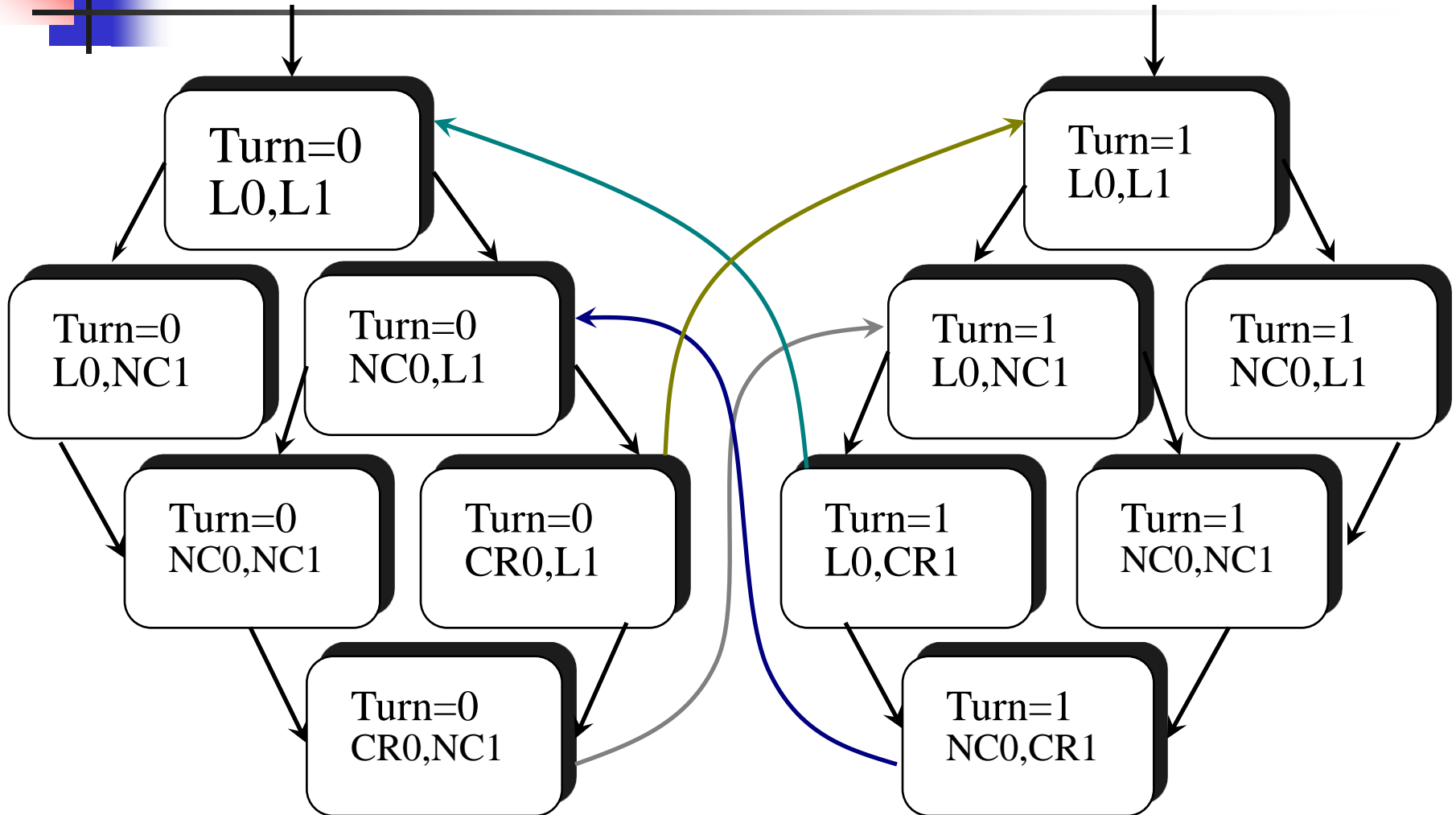


# Some important points

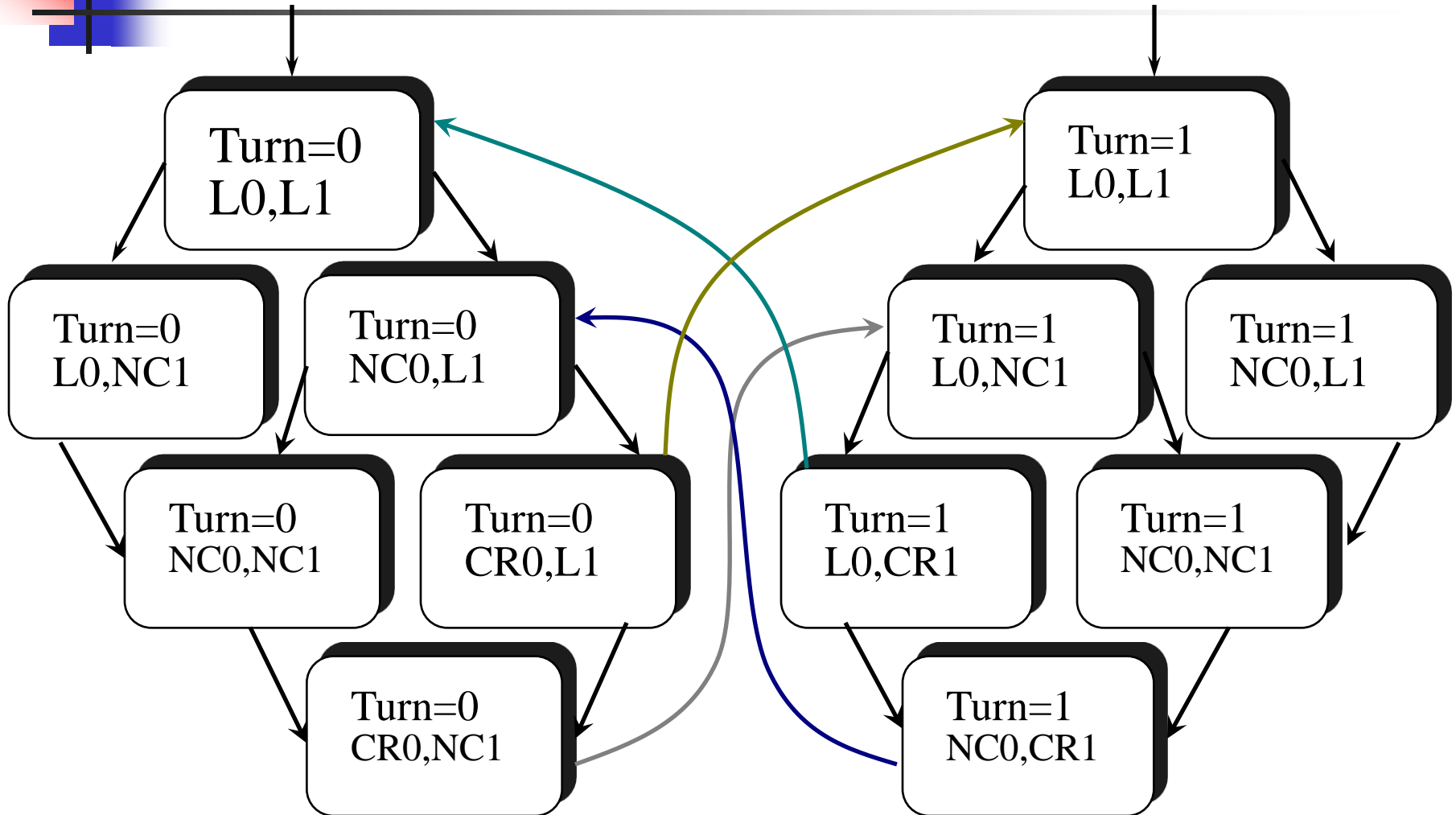
---

- *Reachable* states: obtained from an initial state through a sequence of enabled transitions.
- *Executions*: the set of maximal paths (finite or terminating in a node where nothing is enabled).
- *Nondeterministic choice*: when more than a single transition is enabled at a given state. We have a nondeterministic choice when at least one node at the state graph has more than one successor.
- Propositions correspond to properties that either hold or do not hold in a state.

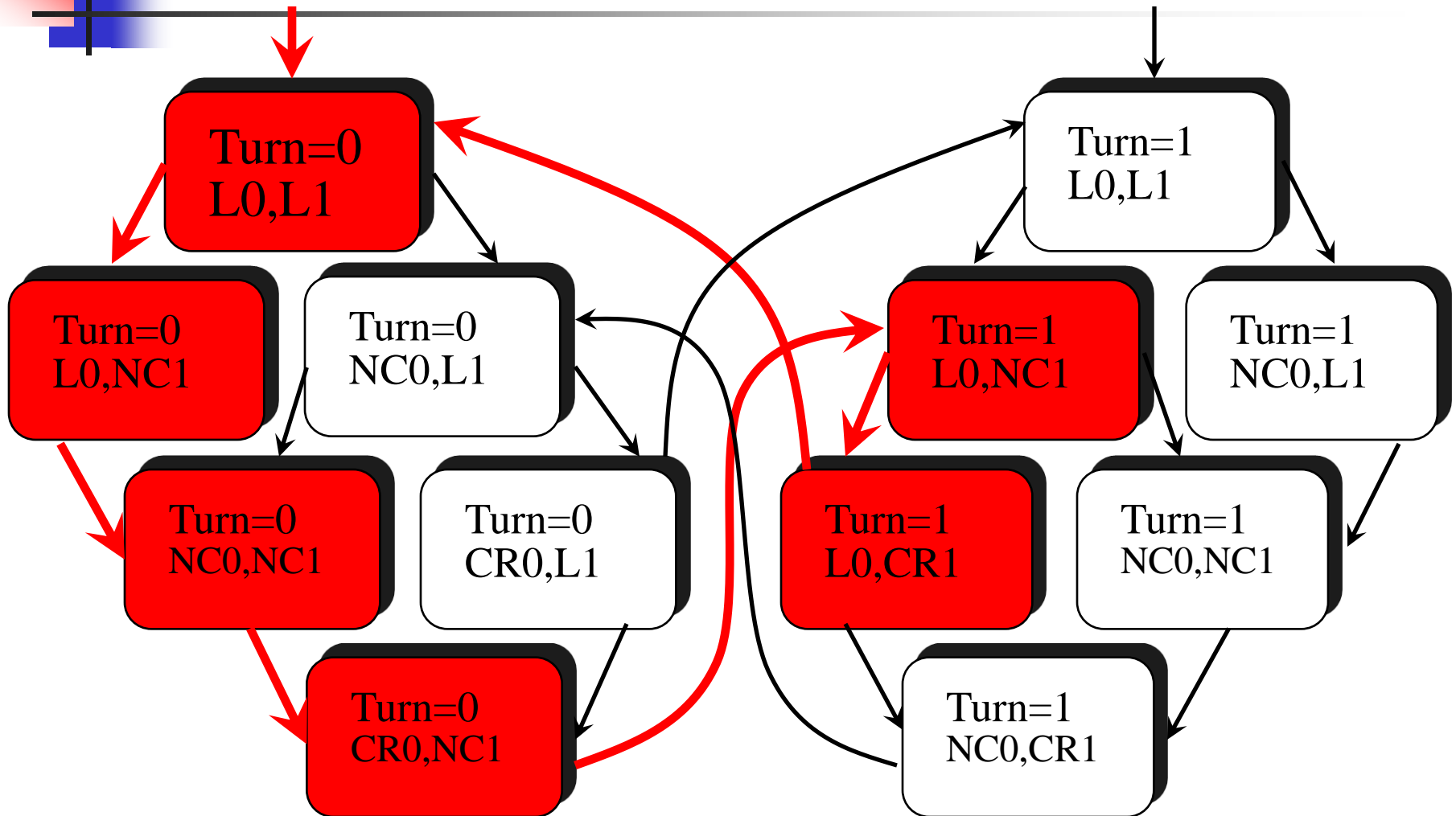
Always  $\neg(PC0=CR0 \wedge PC1=CR1)$   
*(Mutual exclusion)*



# Always if Turn=0 then at some point Turn=1

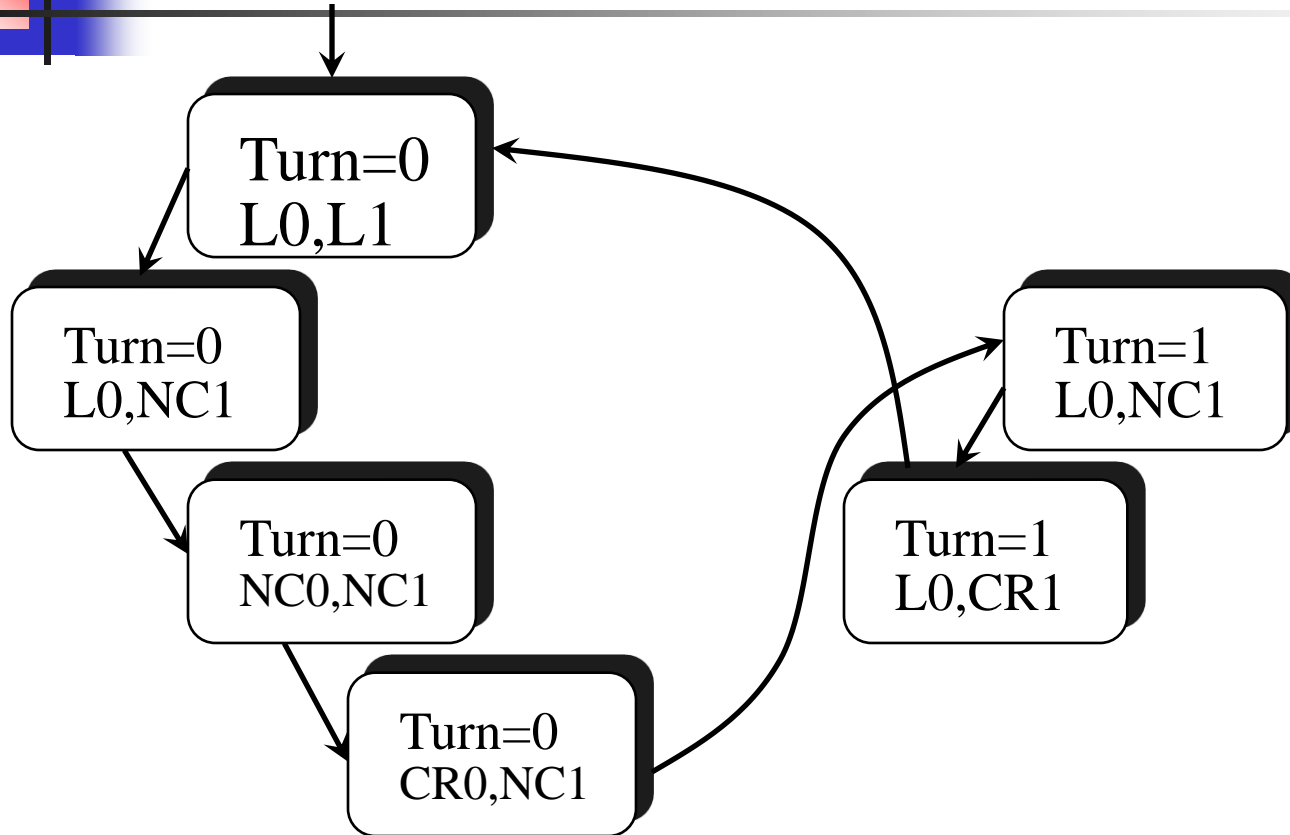


# Always if Turn=0 then at some point Turn=1



# *Interleaving semantics:*

Execute one transition at a time.

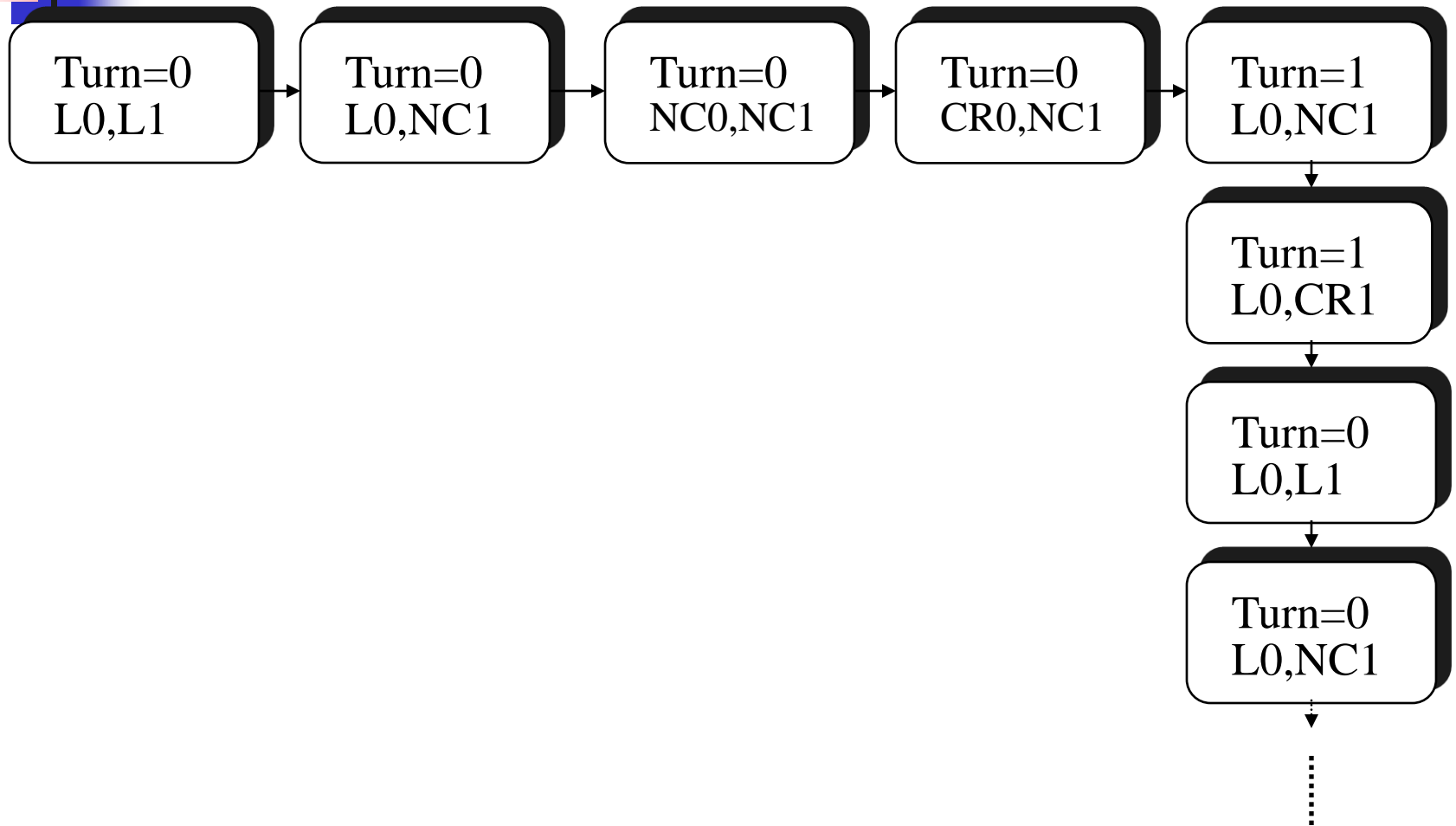


Need to check the property

**for every possible interleaving!**



# Interleaving semantics



# Busy waiting

```
L0:While True do
  NC0:wait(Turn=0);
  CR0:Turn=1
endwhile ||
L1:While True do
  NC1:wait(Turn=1);
  CR1:Turn=0
endwhile
```

T0:PC0=L0 → PC0:=NC0

T1:PC0=NC0/\Turn=0 → PC0:=CR0

T1':PC0=NC0/\Turn=1 → PC0:=NC0

T2:PC0=CR0 → (PC0,Turn):=(L0,1)

T3:PC1=L1 → PC1=NC1

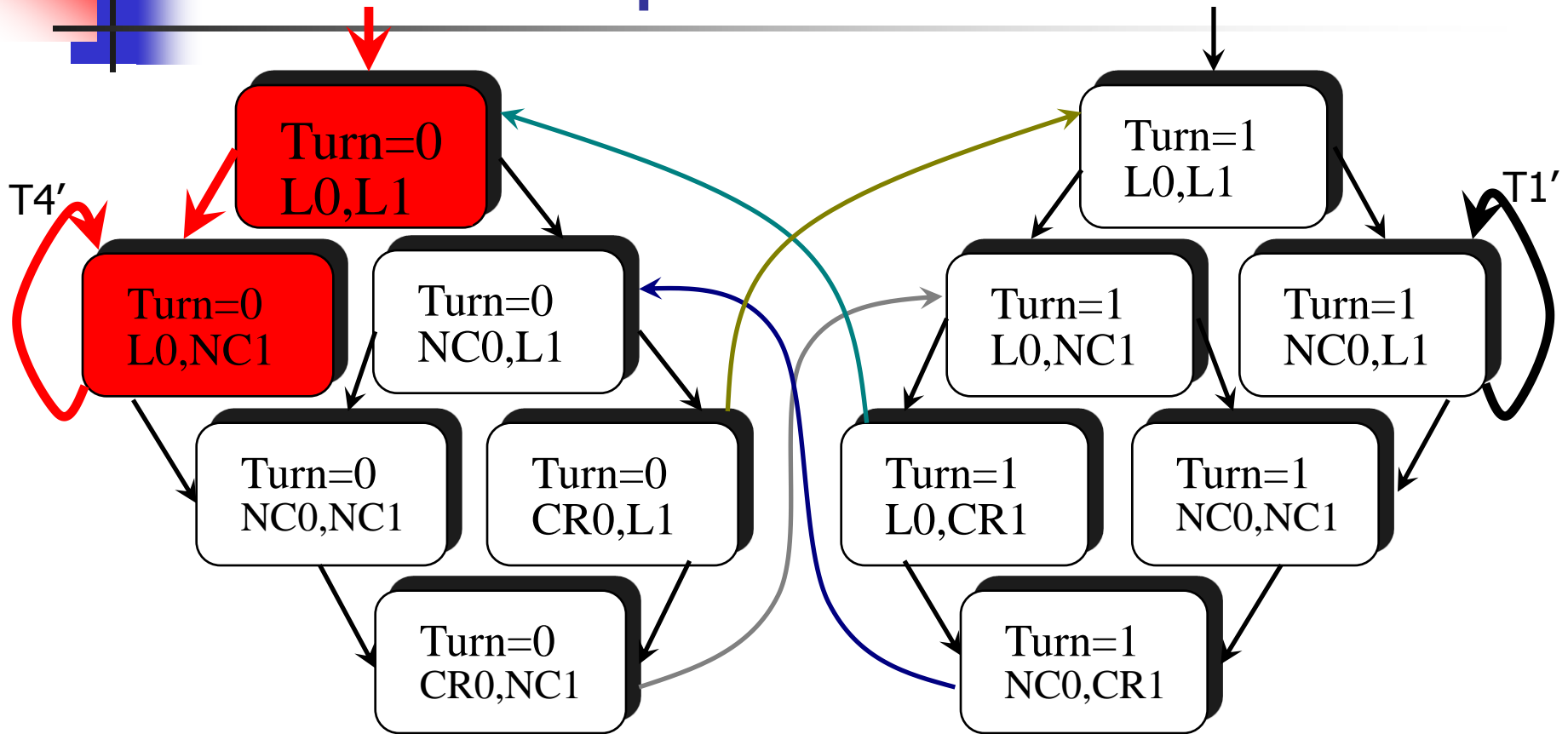
T4:PC1=NC1/\Turn=1 → PC1:=CR1

T4':PC1=NC1/\Turn=0 → PC1:=NC1

T5:PC1=CR1 → (PC1,Turn):=(L1,0)

Initially: PC0=L0/\PC1=L1

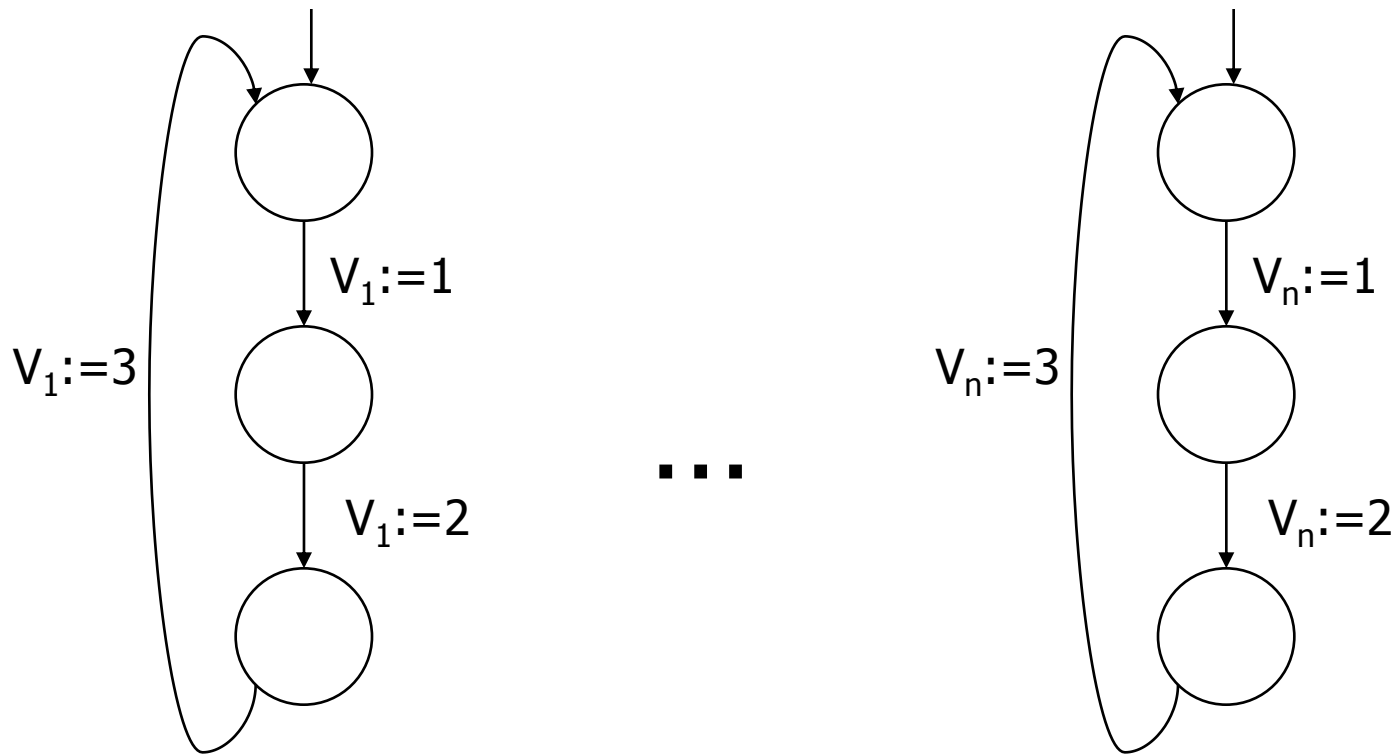
# Always when Turn=0 then at some point Turn=1



Now it does not hold!

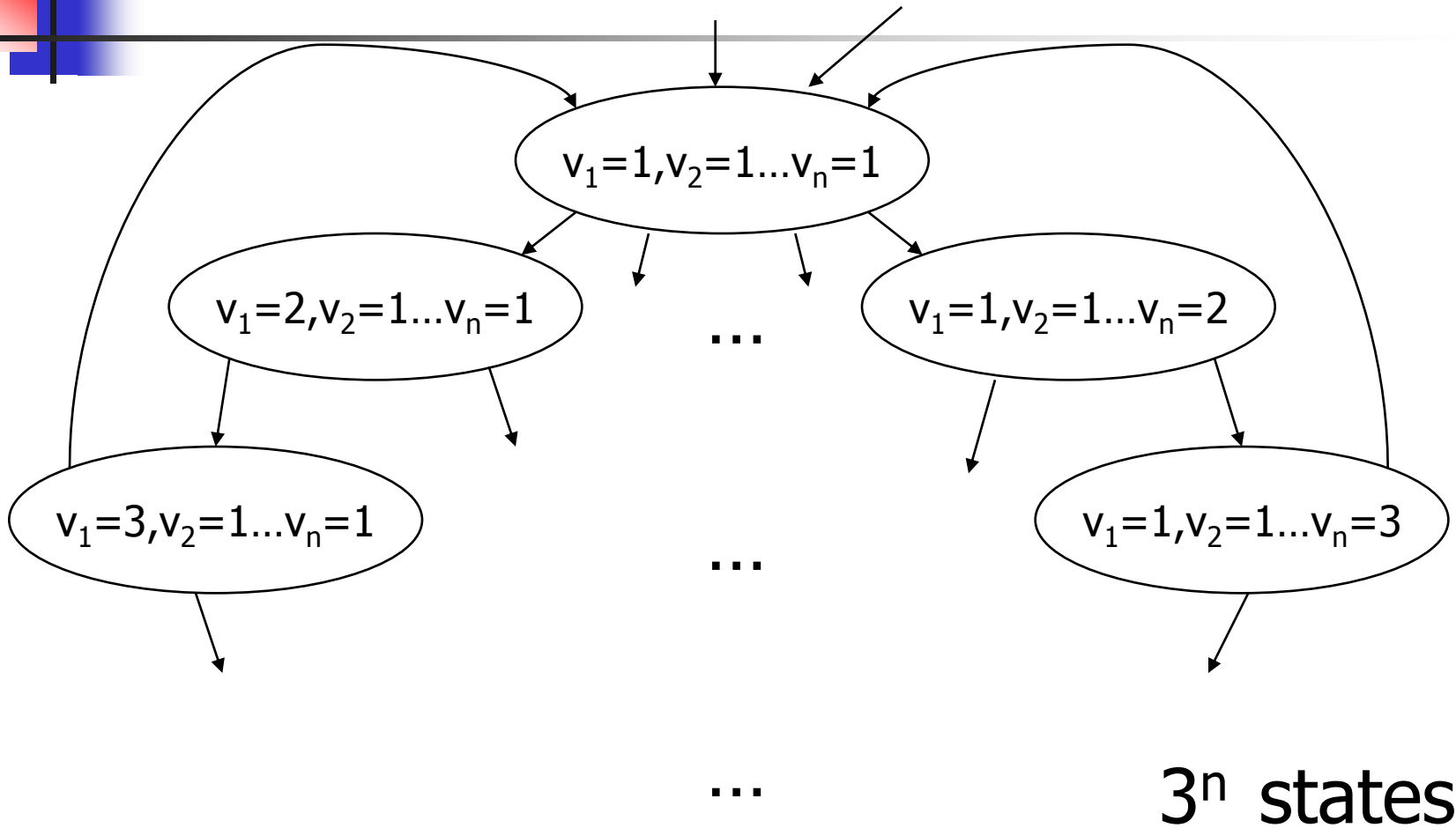
(Red subgraph generates a counterexample execution.)

# Combinatorial explosion



How many states?

# Global states





# Specification Formalisms

---

(Book: Chapter 5)



# Properties of formalisms

---

- *Formal.* Unique interpretation.
- *Intuitive.* Simple to understand (visual).
- *Succinct.* Spec. of reasonable size.
- *Effective.*
  - Check that there are no contradictions.
  - Check that the spec. is implementable.
  - Check that the implementation satisfies spec.
- *Expressive.*
  - May be used to generate initial code.

Specifying the *implementation* or its *properties*?



# A transition system

---

- A (finite) set of variables  $V$ .
- A set of states  $\Sigma$ .
- A (finite) set of transitions  $T$ , each transition  $e \rightarrow t$  has
  - an enabling condition  $e$  and a transformation  $t$ .
- An initial condition  $I$ .
- Denote by  $R(s, s')$  the fact that  $s'$  is a successor of  $s$ .





# The interleaving model

---

- An **execution** is a finite or infinite sequence of states  $s_0, s_1, s_2, \dots$
- The initial state satisfies the initial condition, I.e.,  $I(s_0)$ .
- Moving from one state  $s_i$  to  $s_{i+1}$  is by executing a transition  $e \rightarrow t$ :
  - $e(s_i)$ , I.e.,  $s_i$  satisfies  $e$ .
  - $s_{i+1}$  is obtained by applying  $t$  to  $s_i$ .
- Lets assume all sequences are **infinite** by extending finite ones by “**stuttering**” the last state.



# Temporal logic

---

- Dynamic, speaks about several “worlds” and the relation between them.
- Our “**worlds**” are the **states** in an execution.
- There is a linear relation between them, each two sequences in our execution are ordered.
- Interpretation: over an **execution**, later over **all executions**.



# LTL: Syntax

---

$\varphi ::= (\varphi) \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \mathcal{U} \varphi \mid$   
 $\quad \quad \quad \square \varphi \mid \langle \rangle \varphi \mid \mathcal{O} \varphi \mid p$

$\square \varphi$  — “box”, “always”, “forever”

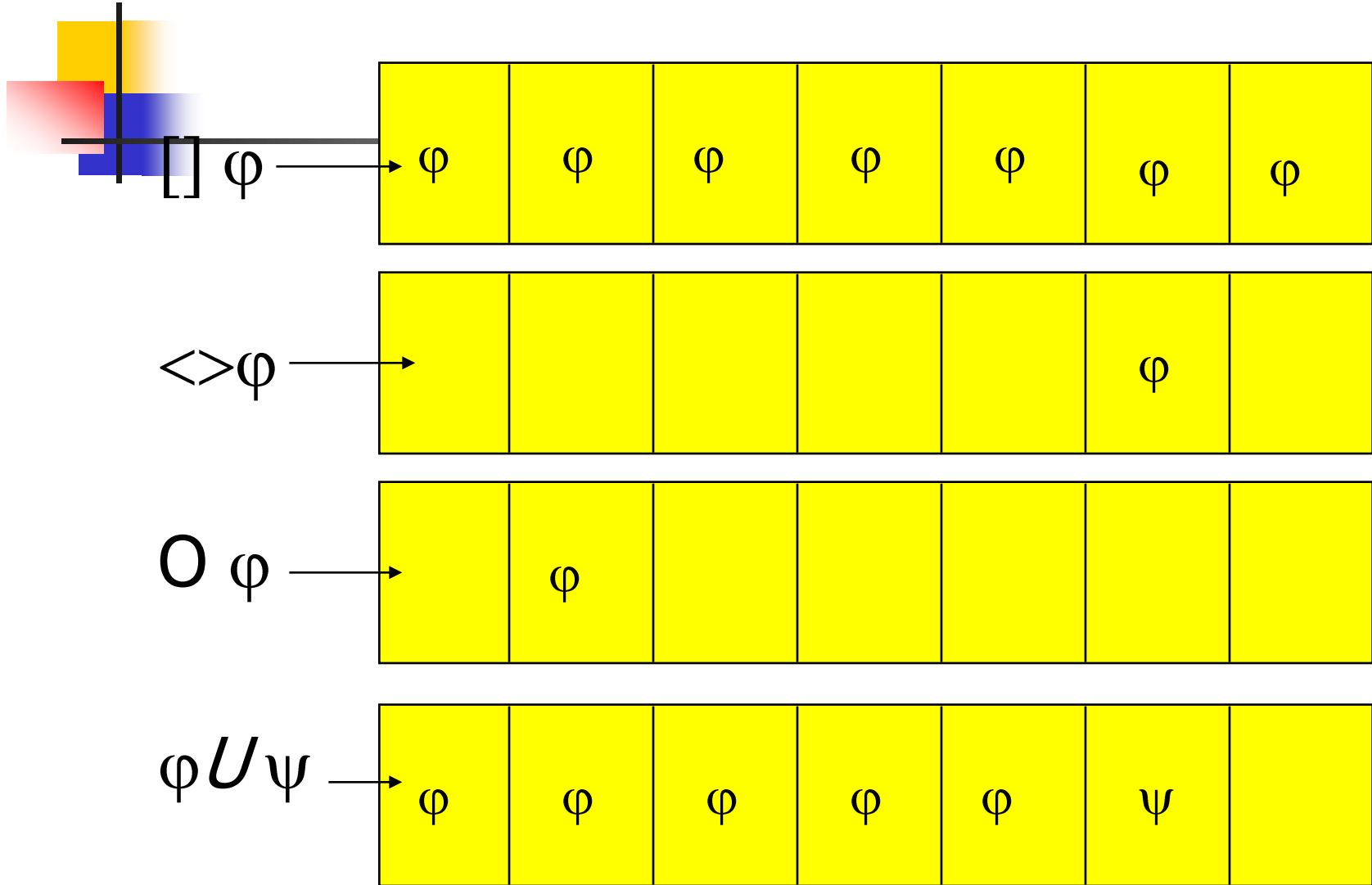
$\langle \rangle \varphi$  — “diamond”, “eventually”, “sometimes”

$\mathcal{O} \varphi$  — “nexttime”

$\varphi \mathcal{U} \psi$  — “until”

Propositions  $p, q, r, \dots$  Each represents some state property ( $x > y + 1, z = t, \text{at\_CR}, \text{etc.}$ )

# Semantics *over suffixes of execution*





# Can discard some operators

---

- Instead of  $\langle \rangle p$ , write  $true \ U \ p$ .
- Instead of  $[]p$ , we can write  $\neg(\langle \rangle \neg p)$ , or  $\neg(true \ U \ \neg p)$ .

Because  $[]p = \neg \neg []p$ .

$\neg []p$  means it is not true that  $p$  holds forever, or at some point  $\neg p$  holds or  $\langle \rangle \neg p$ .



# Combinations

---

- $\Box \langle \rangle p$  “ $p$  will happen infinitely often”
- $\langle \rangle \Box p$  “ $p$  will happen from some point forever”.
- $(\Box \langle \rangle p) \rightarrow (\Box \langle \rangle q)$  “If  $p$  happens infinitely often, then  $q$  also happens infinitely often”.



# Some relations:

---

- $\Box(\varphi \wedge \psi) = (\Box\varphi) \wedge (\Box\psi)$
- But  $\langle \Box \rangle(\varphi \wedge \psi) \neq (\langle \Box \rangle\varphi) \wedge (\langle \Box \rangle\psi)$

		$\psi$		$\varphi$		
--	--	--------	--	-----------	--	--

- $\langle \Box \rangle(\varphi \vee \psi) = (\langle \Box \rangle\varphi) \vee (\langle \Box \rangle\psi)$
- But  $\Box(\varphi \vee \psi) \neq (\Box\varphi) \vee (\Box\psi)$

$\psi$	$\psi$ $\varphi$	$\psi$	$\psi$ $\varphi$	$\varphi$	$\psi$	$\varphi$
--------	------------------	--------	------------------	-----------	--------	-----------



# Formal semantic definition

---

- Let  $\sigma$  be a sequence  $s_0 s_1 s_2 \dots$
- Let  $\sigma^i$  be a suffix of  $\sigma$ :  $s_i s_{i+1} s_{i+2} \dots$  ( $\sigma^0 = \sigma$ )
- $\sigma^i \models p$ , where  $p$  a proposition, if  $s_i \models p$ .
- $\sigma^i \models \varphi \wedge \psi$  if  $\sigma^i \models \varphi$  and  $\sigma^i \models \psi$ .
- $\sigma^i \models \varphi \vee \psi$  if  $\sigma^i \models \varphi$  or  $\sigma^i \models \psi$ .
- $\sigma^i \models \neg \varphi$  if it is not the case that  $\sigma^i \models \varphi$ .
- $\sigma^i \models \langle \rangle \varphi$  if for some  $j \geq i$ ,  $\sigma^j \models \varphi$ .
- $\sigma^i \models [] \varphi$  if for each  $j \geq i$ ,  $\sigma^j \models \varphi$ .
- $\sigma^i \models \varphi \mathcal{U} \psi$  if for some  $j \geq i$ ,  $\sigma^j \models \psi$ .  
and for each  $i \leq k < j$ ,  $\sigma^k \models \varphi$ .





# Then we interpret:

---

- *For a state:*  
 $s \models p$  as in propositional logic.
- *For an execution:*  
 $\sigma \models \varphi$  is interpreted over a sequence, as in previous slide.
- *For a system/program:*  
 $P \models \varphi$  holds if  $\sigma \models \varphi$  for every sequence  $\sigma$  of  $P$ .



# Specifications

---

- $[\ ] (PC0=NC0 \rightarrow \langle \rangle PC0=CR0)$
  - $[\ ] (PC0=NC0 \rightarrow PC0=NC0 \cup Turn=0)$
- Ex. The processes alternate in entering their critical sections.
- Ex. Each process enters its critical section infinitely often.

# Traffic light example



Green → Yellow → Red

Always has exactly one light:

$$\square (\neg(\text{gr} \wedge \text{ye}) \wedge \neg(\text{ye} \wedge \text{re}) \wedge \neg(\text{re} \wedge \text{gr}) \wedge (\text{gr} \vee \text{ye} \vee \text{re}))$$

Correct change of color:

$$\square ((\text{gr} \cup \text{ye}) \vee (\text{ye} \cup \text{re}) \vee (\text{re} \cup \text{gr}))$$

# Another kind of traffic light

Green → Yellow → Red → Yellow

First attempt:

~~$[ ] ( ( ( gr \vee re ) U ye ) \vee ( ye U ( gr \vee re ) ) ) )$~~

Correct specification:

$[ ] ( ( gr \rightarrow ( gr U ( ye \wedge ( ye U re ) ) ) ) )$

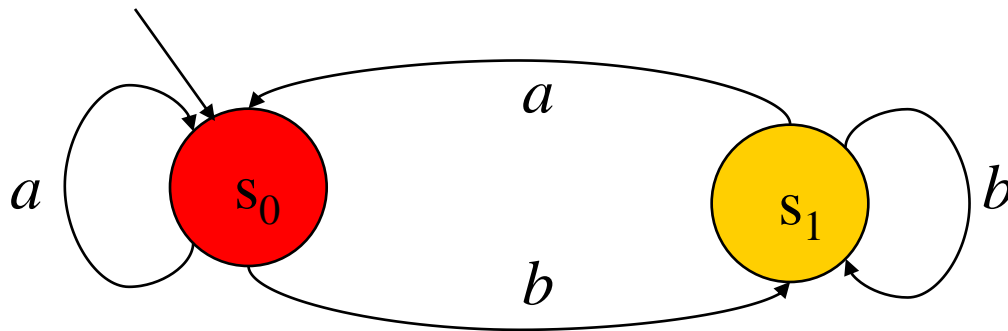
$\wedge ( re \rightarrow ( re U ( ye \wedge ( ye U gr ) ) ) ) )$

$\wedge ( ye \rightarrow ( ye U ( gr \vee re ) ) ) )$

← Needed only when we can start with yellow

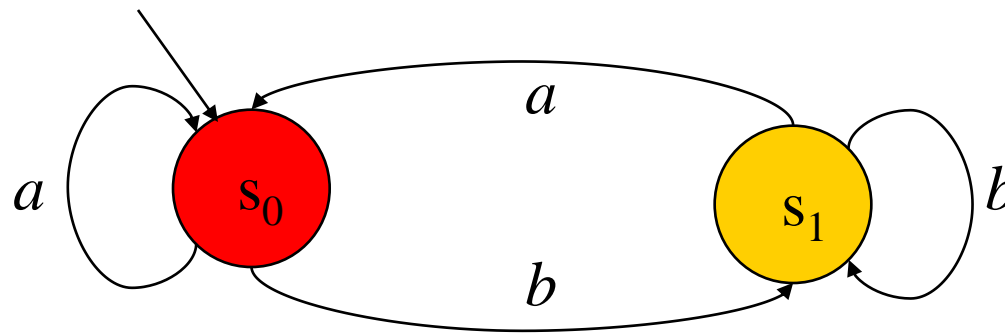
# Automata over finite words

- $A = \langle \Sigma, S, \Delta, I, F \rangle$
- $\Sigma$  (finite) - the alphabet.
- $S$  (finite) - the states.
- $\Delta \subseteq S \times \Sigma \times S$  - the transition relation.
- $I \subseteq S$  - the starting states.
- $F \subseteq S$  - the accepting states.



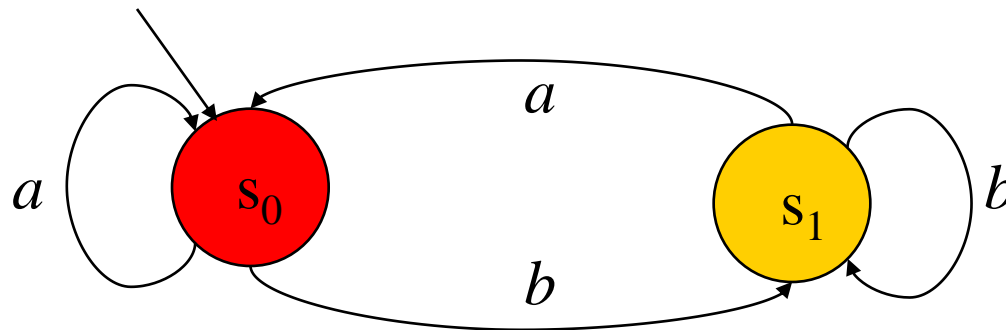
# The transition relation

- $(s_0, a, s_0)$
- $(s_0, b, s_1)$
- $(s_1, a, s_0)$
- $(s_1, b, s_1)$



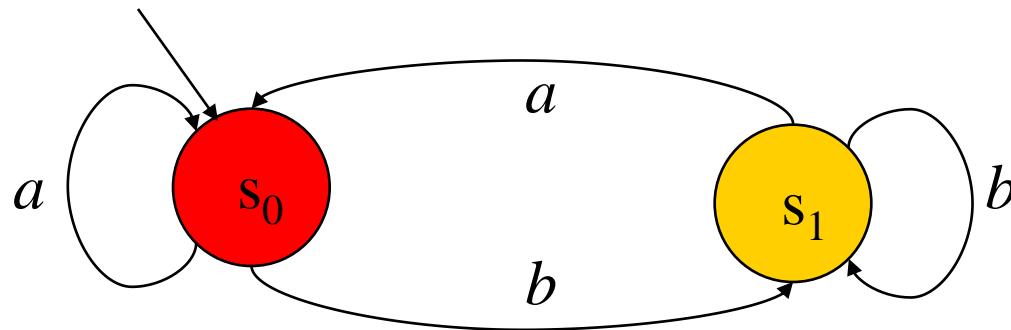
# A *run* over a word

- A word over  $\Sigma$ , e.g., *abaab*.
- A sequence of states, e.g.  $s_0 s_0 s_1 s_0 s_0 s_1$ .
- Starts with an initial state.
- Follows the transition relation  $(s_i, c_j, s_{i+1})$ .
- Accepting if ends at accepting state.



# The *language* of an automaton

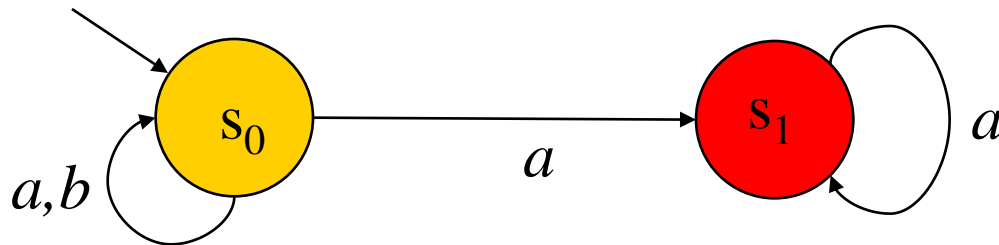
- The words that are accepted by the automaton.
- Includes *aabbba*, *abbbba*.
- Does not include *abab*, *abbb*.
- What is the language?



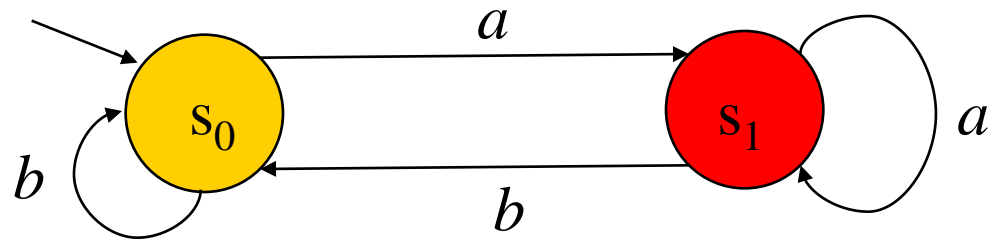
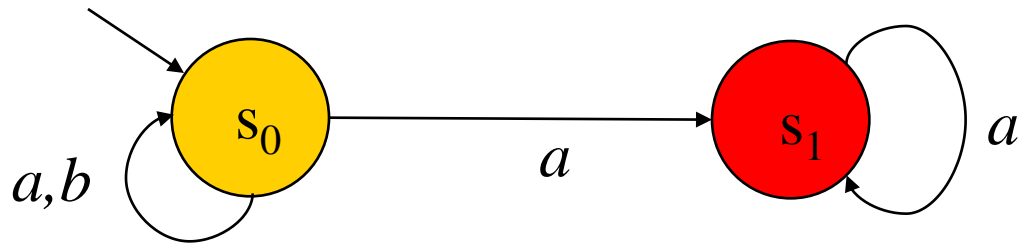


# Nondeterministic automaton

- Transitions:  $(s_0, a, s_0)$ ,  $(s_0, b, s_0)$ ,  $(s_0, a, s_1)$ ,  $(s_1, a, s_1)$ .
- What is the language of this automaton?

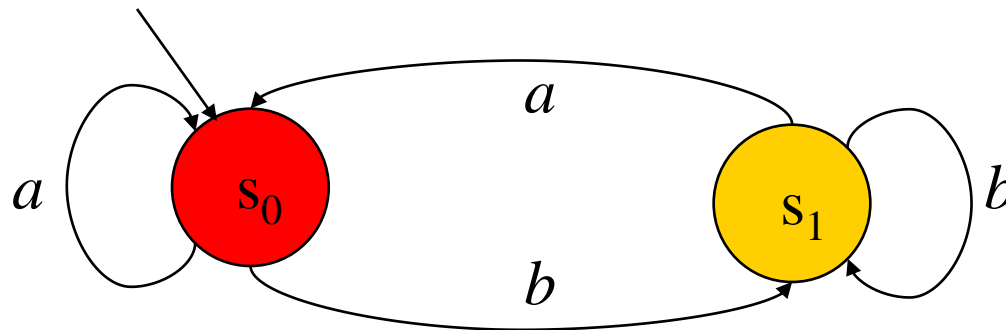


# Equivalent deterministic automaton



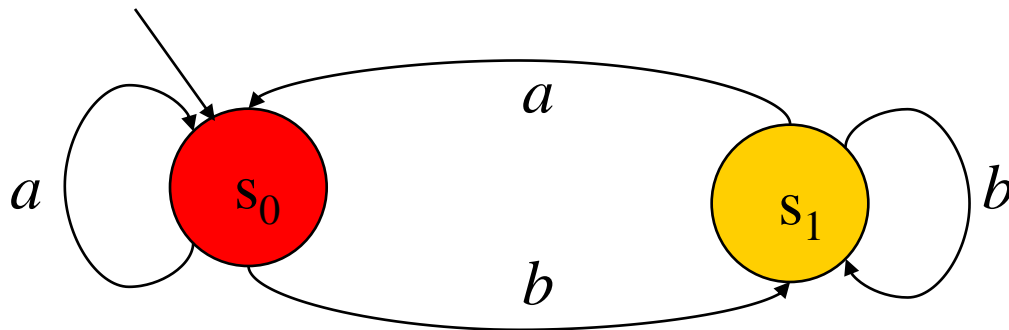
# Automata over infinite words

- Similar definition.
- Runs on infinite words over  $\Sigma$ .
- Accepts when an accepting state occurs infinitely often in a run.



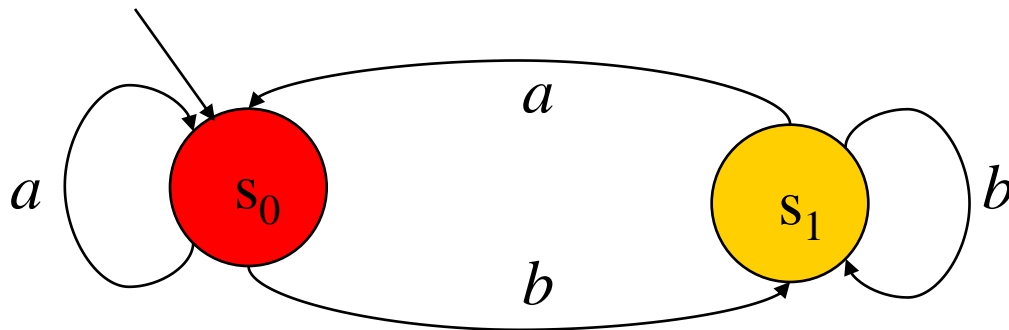
# Automata over infinite words

- Consider the word  $abababab\dots$
- There is a run  $s_0s_0s_1s_0s_1s_0s_1\dots$
- This run is accepting, since  $s_0$  appears infinitely many times.



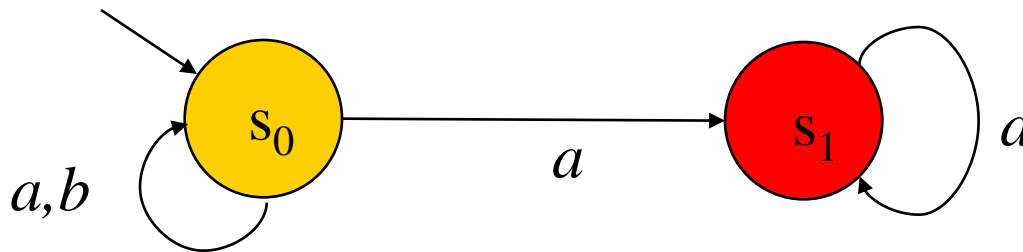
# Other runs

- For the word  $bbbb\dots$  the run is  $s_0 s_1 s_1 s_1 s_1\dots$  and is not accepting.
- For the word  $aaabbbb\dots$ , the run is  $s_0 s_0 s_0 s_0 s_1 s_1 s_1 s_1\dots$
- What is the run for  $ababbabb\dots$  ...?



# Nondeterministic automaton

- What is the language of this automaton?
- What is the LTL specification if  $b \leftrightarrow PC0=CR0, a = \neg b$ ?



- Can you find a deterministic automaton with same language?
- Can you prove there is no such deterministic automaton?

# No deterministic automaton

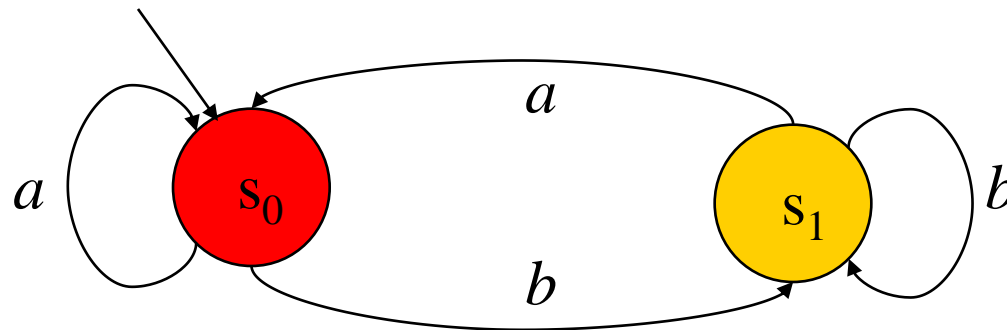
for  $(a+b)^* a^\omega$

---

- In a deterministic automaton there is one run for each word.
- After some sequence of  $a$ 's, i.e.,  $aaa...a$  must reach some accepting state.
- Now add  $b$ , obtaining  $aaa...ab$ .
- After some more  $a$ 's, i.e.,  $aaa...abaaa...a$  must reach some accepting state.
- Now add  $b$ , obtaining  $aaa...abaaa...ab$ .
- Continuing this way, one obtains a run that has infinitely many  $b$ 's but reaches an accepting state (in a finite automaton, at least one would repeat) infinitely often.

# Specification using Automata

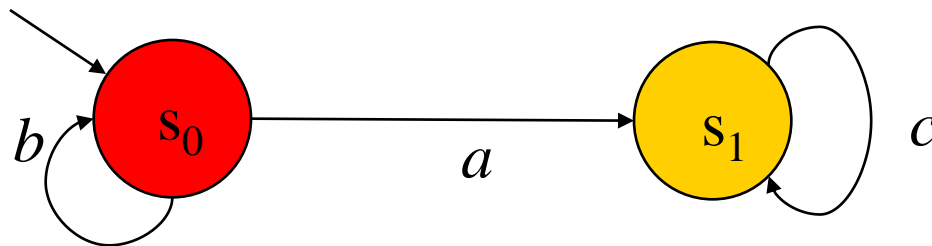
- Let each letter correspond to some propositional property.
- Example:  $a$  -- P0 enters critical section,  
 $b$  -- P0 does not enter section.
- $[\ ] \leftrightarrow PC0 = CR0$



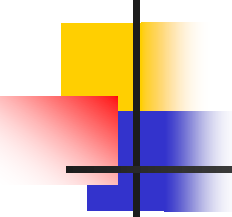


# Mutual Exclusion

- $a$  --  $PC0=CR0/\backslash PC1=CR1$
- $b$  --  $\neg(PC0=CR0/\backslash PC1=CR1)$
- $c$  -- true
- $[]\neg(PC0=CR0/\backslash PC1=CR1)$



Apply now to our  
program:



---

```
L0:While True do
  NC0:wait(Turn=0);
  CR0:Turn=1
endwhile ||
L1:While True do
  NC1:wait(Turn=1);
  CR1:Turn=0
endwhile
```

T0:PC0=L0 → PC0=NC0

T1:PC0=NC0/\Turn=0 →

PC0:=CR0

T2:PC0=CR0 →

(PC0,Turn):=(L0,1)

T3:PC1=L1 → PC1=NC1

T4:PC1=NC1/\Turn=1 →

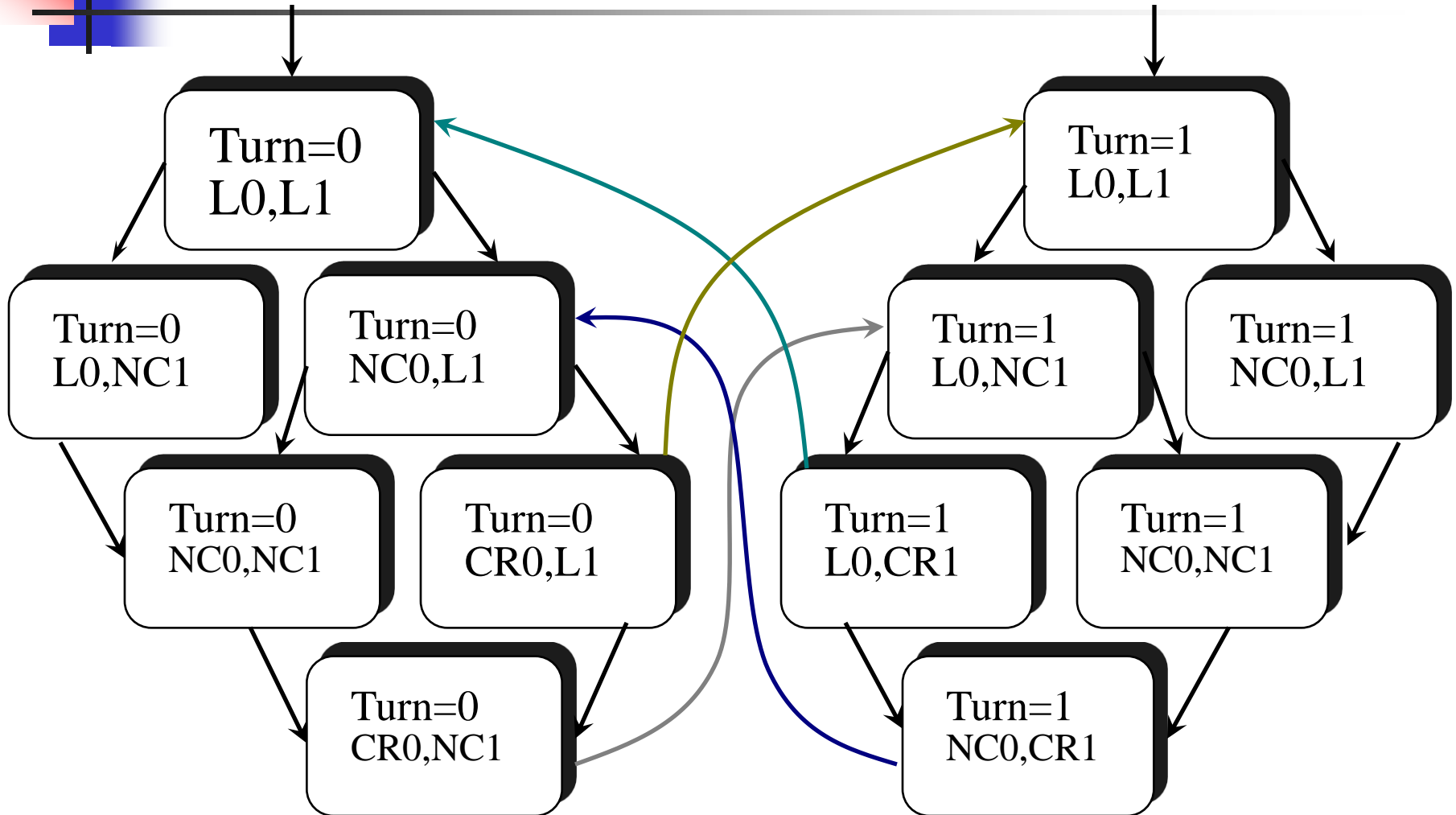
PC1:=CR1

T5:PC1=CR1 →

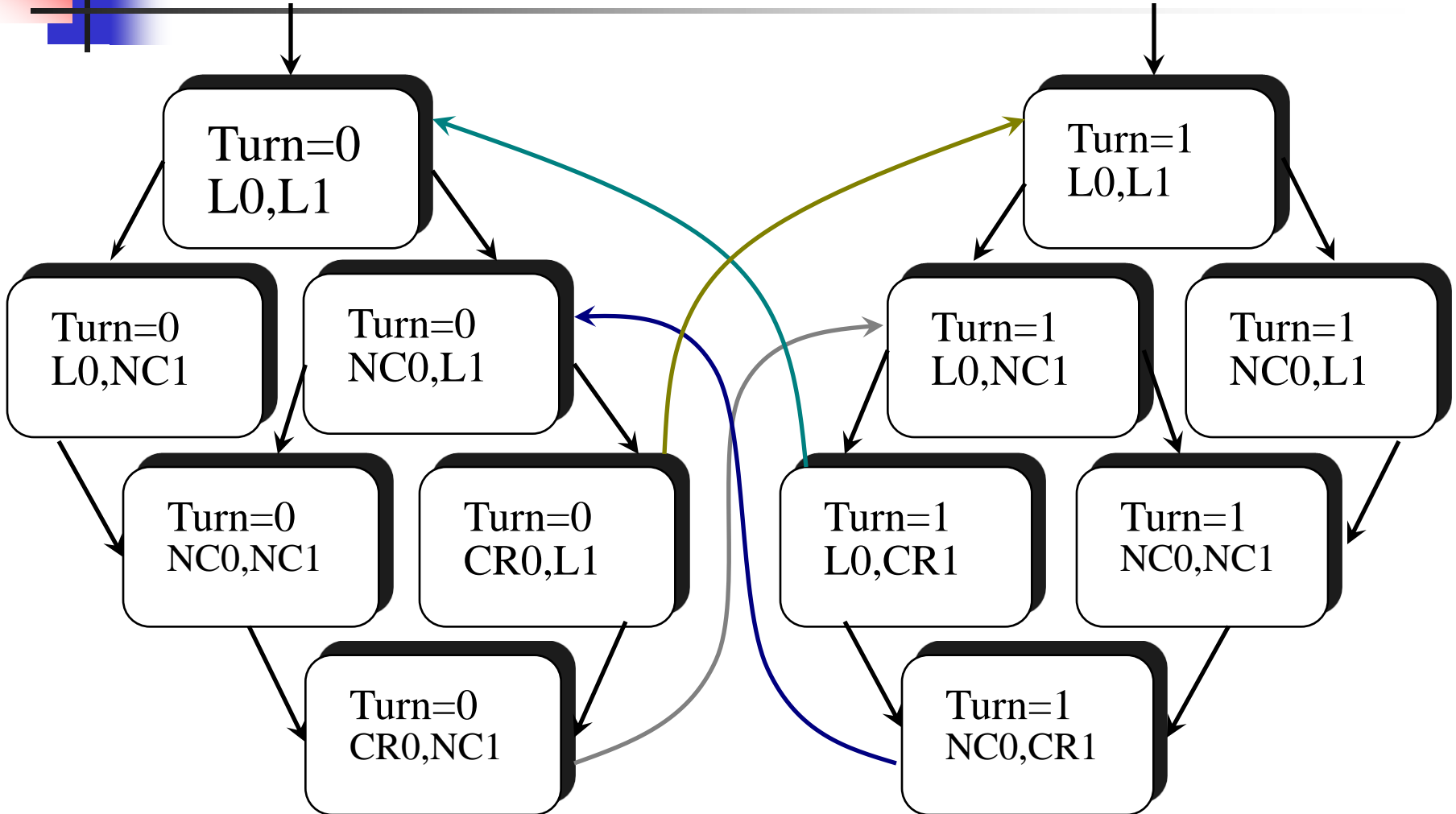
(PC1,Turn):=(L1,0)

Initially: PC0=L0/\PC1=L1

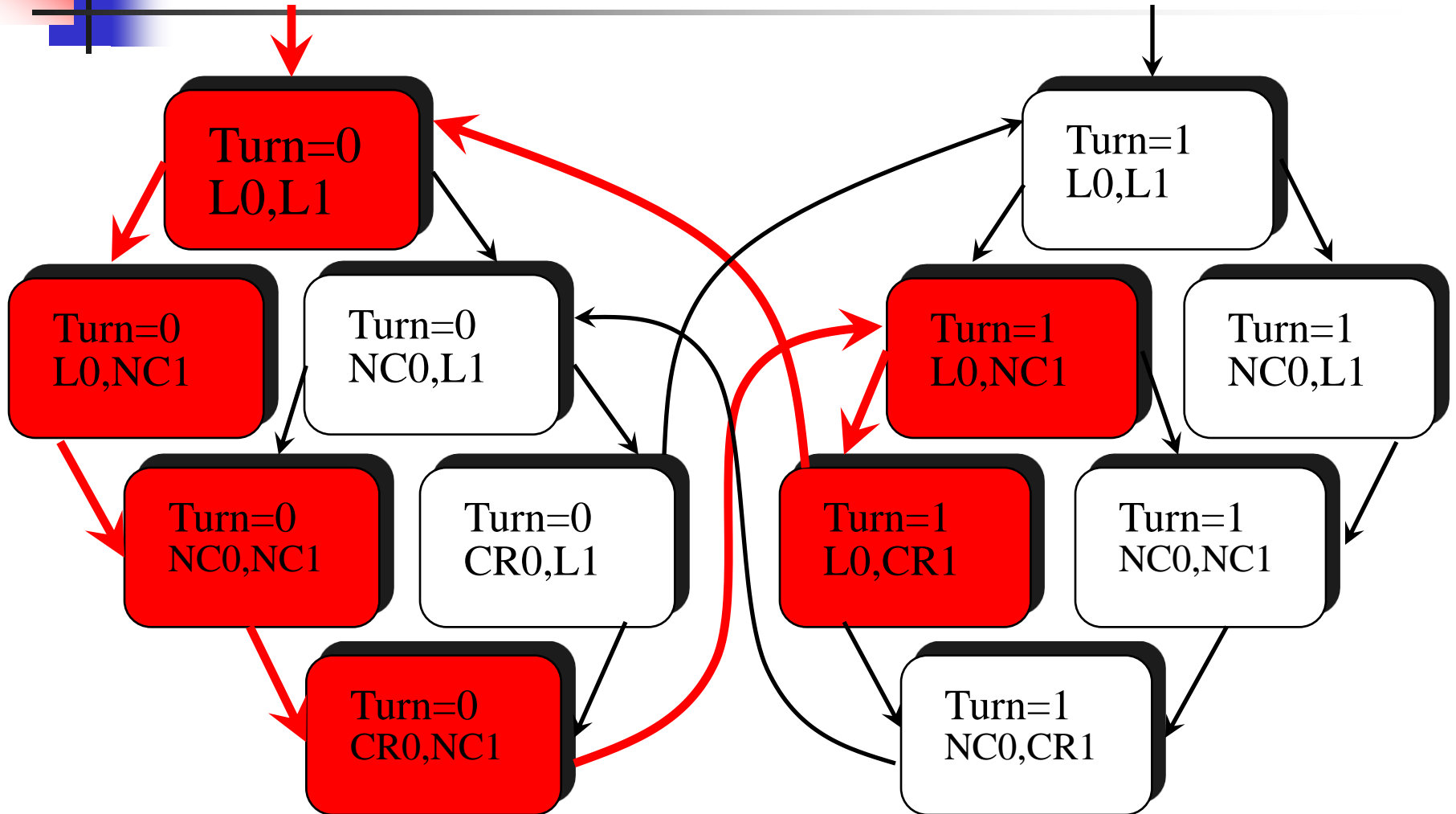
# The state space



$[\ ] \neg (PC0 = CR0 \wedge PC1 = CR1)$   
*(Mutual exclusion)*

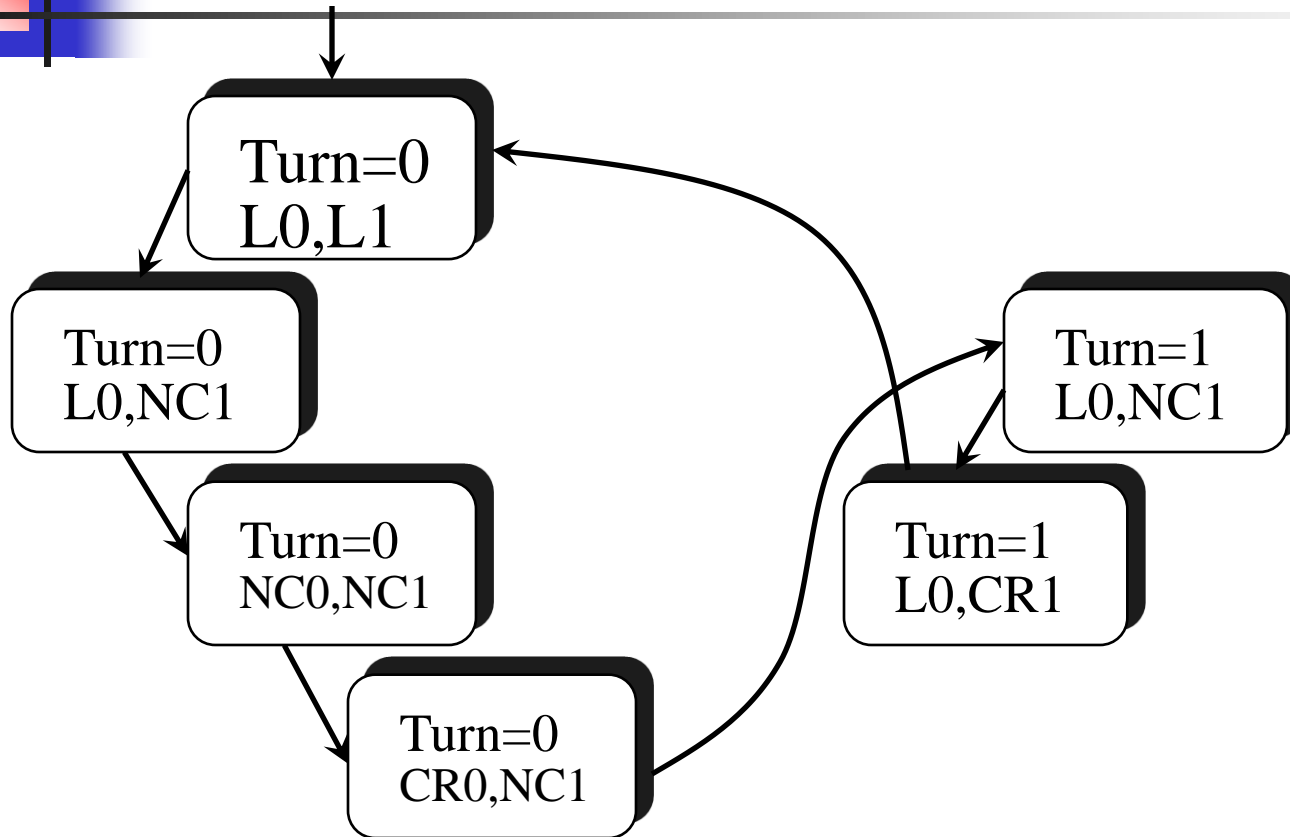


[(Turn=0 → <> Turn=1)]



# *Interleaving semantics:*

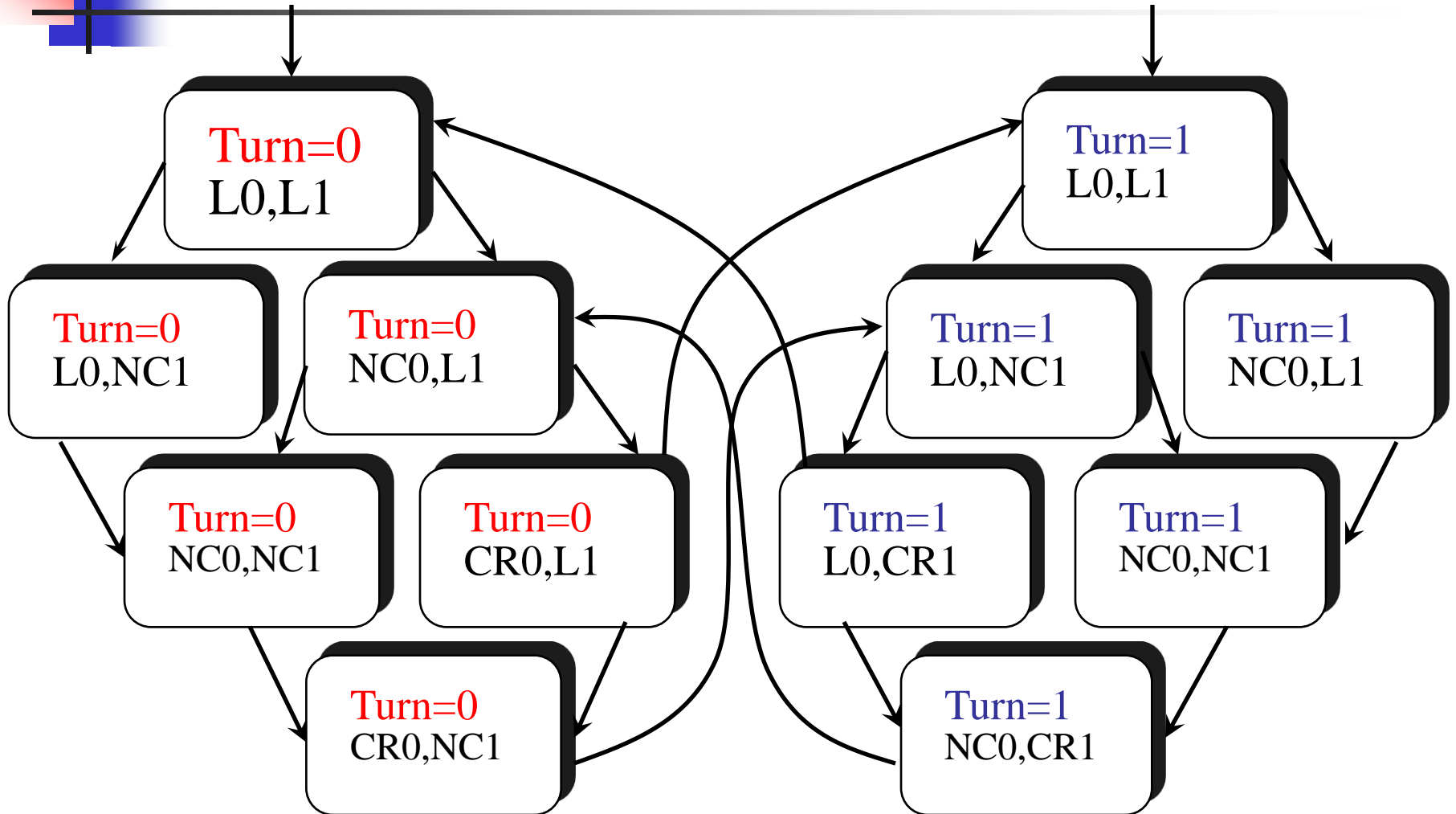
Execute one transition at a time.



Need to check the property

**for every possible interleaving!**

[(Turn=0 → <> Turn=1)]





# Correctness condition

---

- We want to find a correctness condition for a model to satisfy a specification.
- Language of a model:  $L(\text{Model})$
- Language of a specification:  $L(\text{Spec})$ .
  
- We need:  $L(\text{Model}) \subseteq L(\text{Spec})$ .





# Correctness

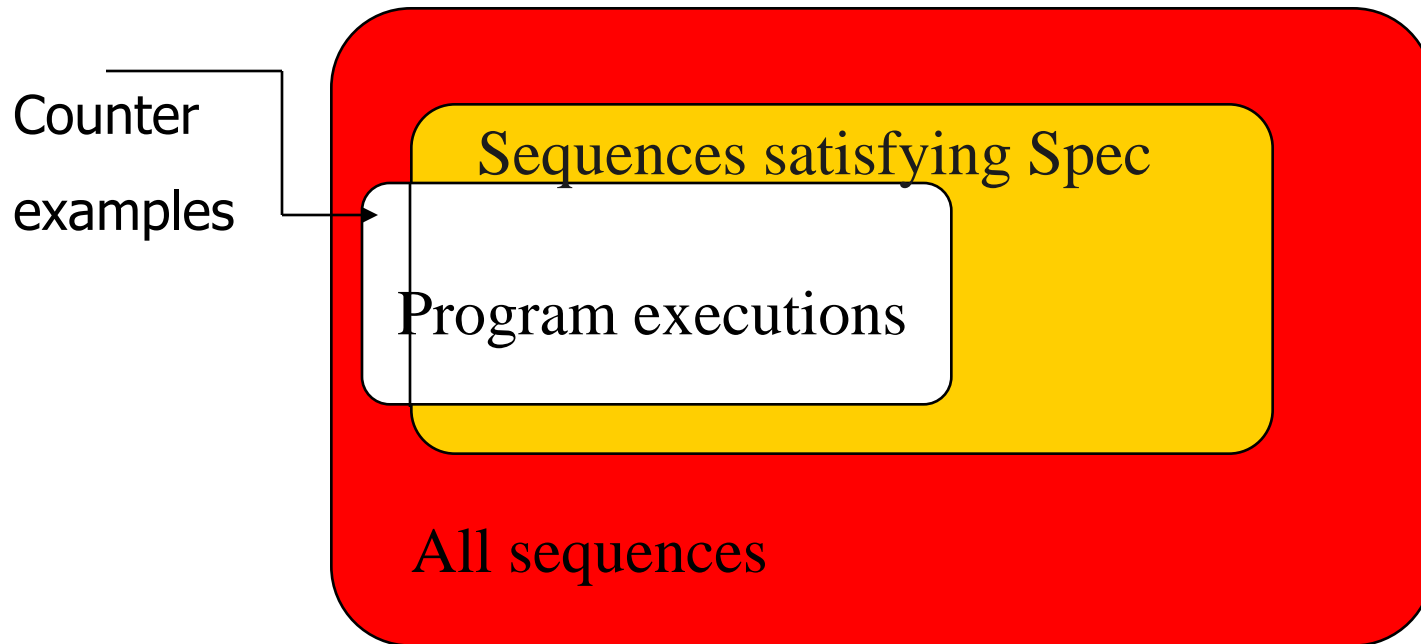
---

Sequences satisfying Spec

Program executions

All sequences

# Incorrectness





# Automatic Verification

---

(Book: Chapter 6)



# How can we check the model?

---

- The model is a graph.
- The specification should refer the graph representation.
- Apply graph theory algorithms.



# What properties can we check?

---

- Invariant: a property that needs to hold in each reachable state.
- Deadlock detection: can we reach a state where the program is blocked?
- Dead code: does the program have parts that are never executed.



# How to perform the checking?

---

- Apply a search strategy (Depth first search, Breadth first search).
- Check states/transitions during the search.
- If property does not hold, **report counter example!**

# If it is so good, why learn deductive verification methods?



---

- Model checking works only for finite state systems. Would not work with
  - Unconstrained integers.
  - Unbounded message queues.
  - General data structures:
    - queues
    - trees
    - stacks
  - parametric algorithms and systems.



# The state space explosion

---

- Need to represent the state space of a program in the computer memory.
- Each state can be as big as the entire memory!
- Many states:  
Each integer variable has  $2^{32}$  possibilities.  
Two such variables have  $2^{64}$  possibilities.
- In concurrent protocols, the number of states can exponentially with the number of processes.





## If it is so constrained, is it of any use?

---

- Many protocols are finite state.
- Many programs or procedure are finite state in nature. Can use **abstraction** techniques.
- Sometimes it is possible to decompose a program, and prove part of it by model checking and part by theorem proving.
- Many techniques to reduce the state space explosion.

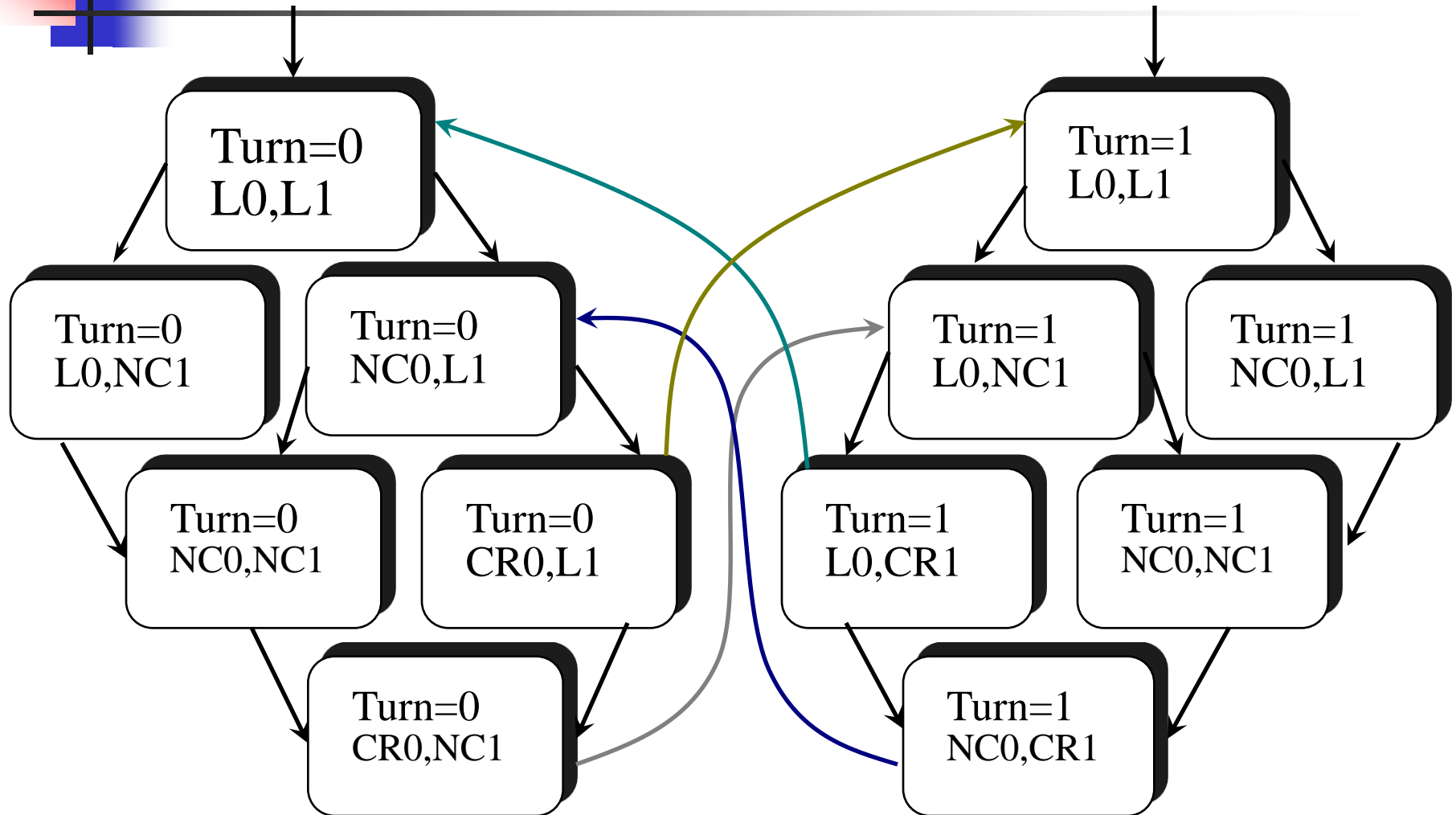


## How can we check properties with DFS?

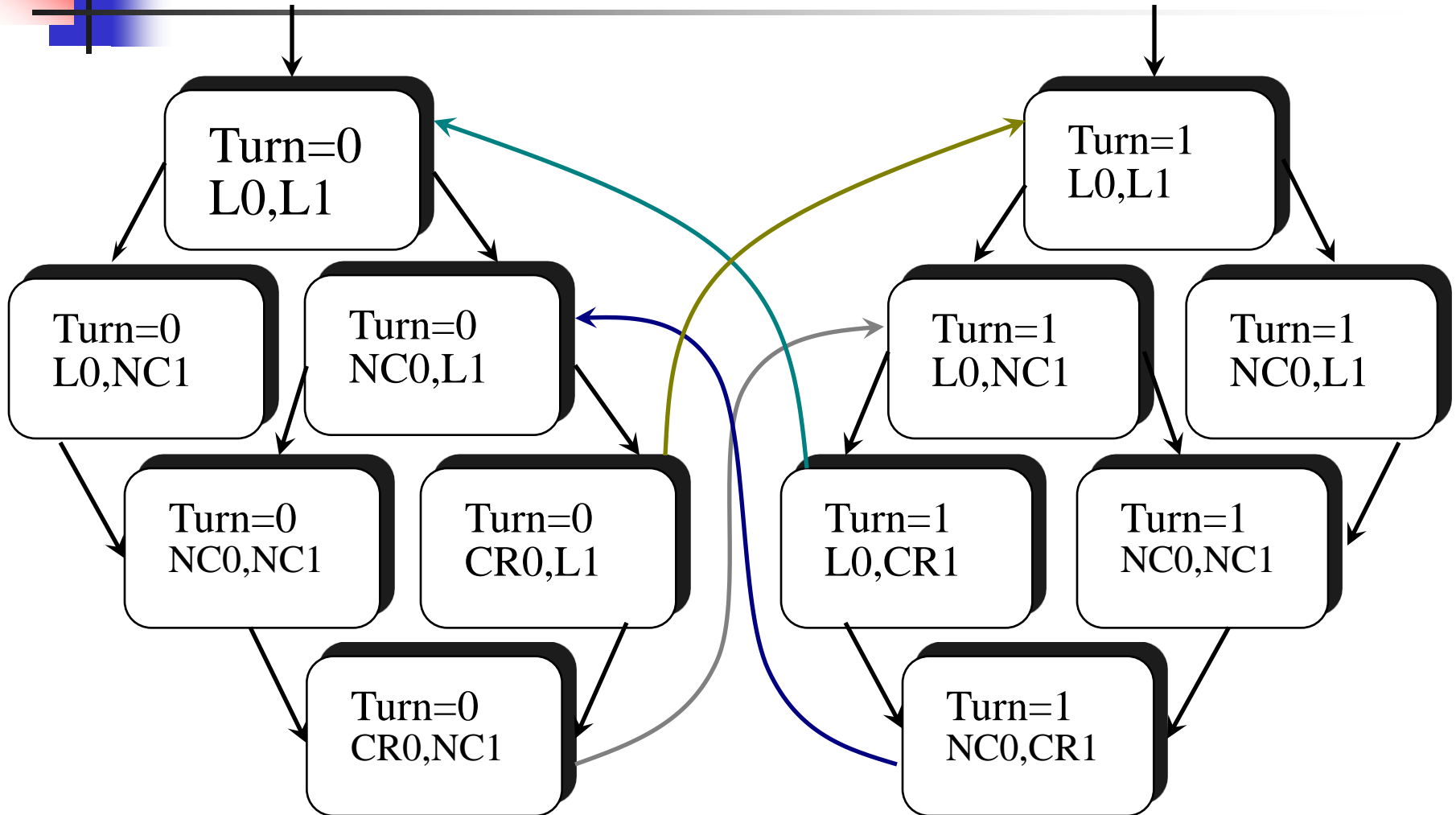
---

- Invariants: check that all reachable states satisfy the invariant property. If not, show a path from an initial state to a bad state.
- Deadlocks: check whether a state where no process can continue is reached.
- Dead code: as you progress with the DFS, mark all the transitions that are executed at least once.

# The state relation between states.



$\neg(PC0=CR0 \wedge PC1=CR1)$  is  
an invariant!



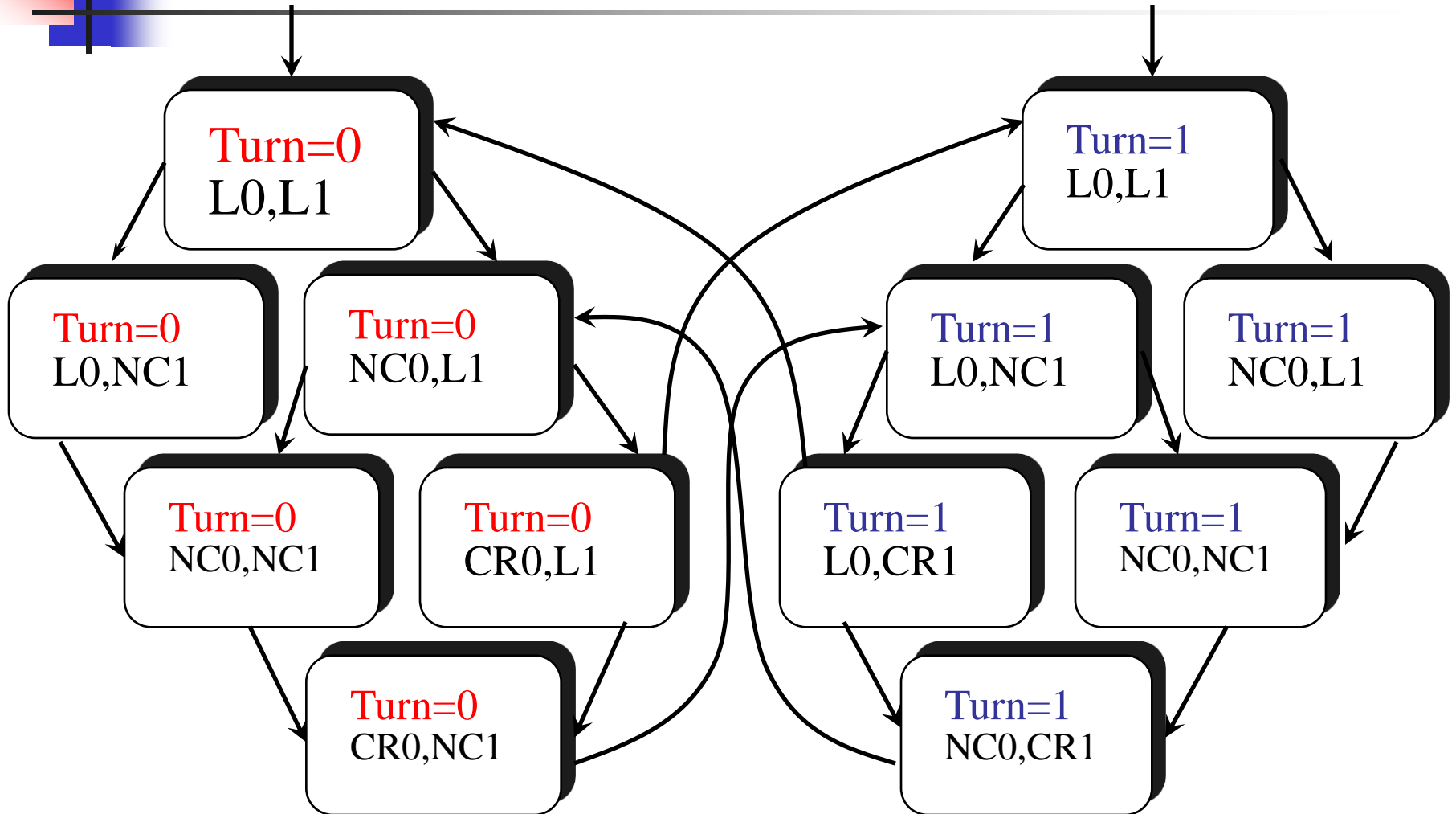


# Want to do more!

---

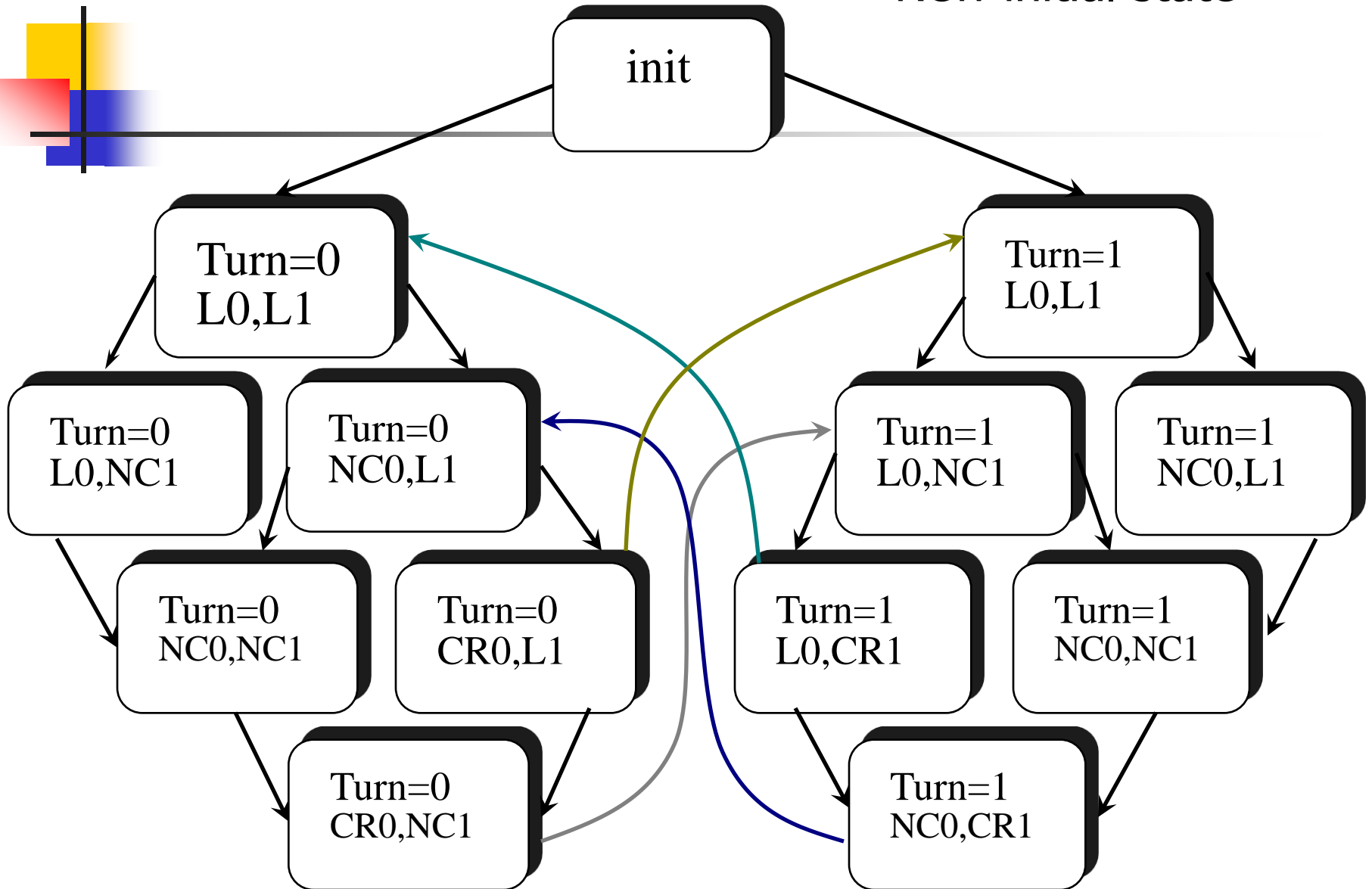
- Want to check more properties.
- Want to have a unique algorithm to deal with all kinds of properties.
- This is done by writing specification in more complicated formalisms.
- We will see that in the next lecture.

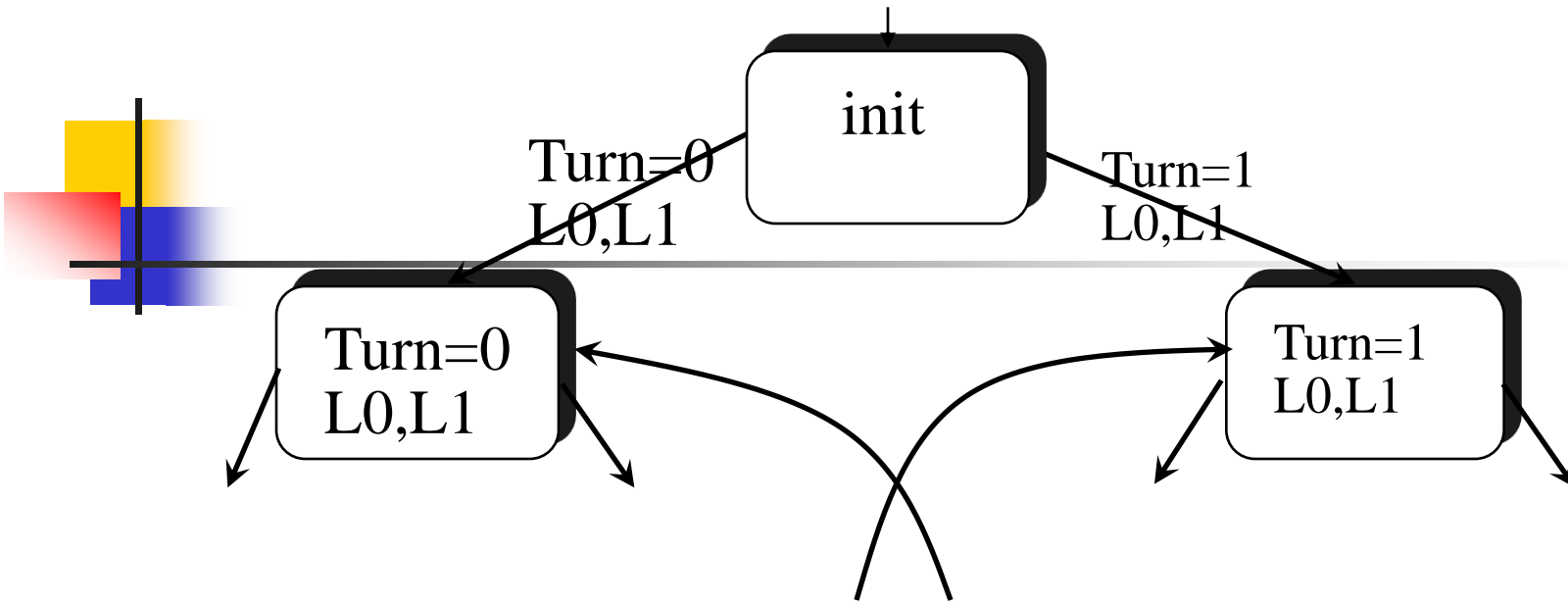
[(Turn=0 → <> Turn=1)]



Convert graph into Buchi automaton

New initial state





- Propositions are attached to incoming nodes.
- All nodes are accepting.





# Correctness condition

---

- We want to find a correctness condition for a model to satisfy a specification.
- Language of a model:  $L(\text{Model})$
- Language of a specification:  $L(\text{Spec})$ .
  
- We need:  $L(\text{Model}) \subseteq L(\text{Spec})$ .



# Correctness

---

Sequences satisfying Spec

Program executions

All sequences



# How to prove correctness?

---

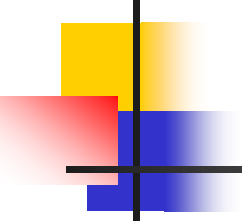
- Show that  $L(\text{Model}) \subseteq L(\text{Spec})$ .
- Equivalently:  
Show that  $L(\text{Model}) \cap \overline{L(\text{Spec})} = \emptyset$ .
- Also: can obtain Spec by translating from LTL!



# What do we need to know?

---

- How to intersect two automata?
- How to complement an automaton?
- How to translate from LTL to an automaton?

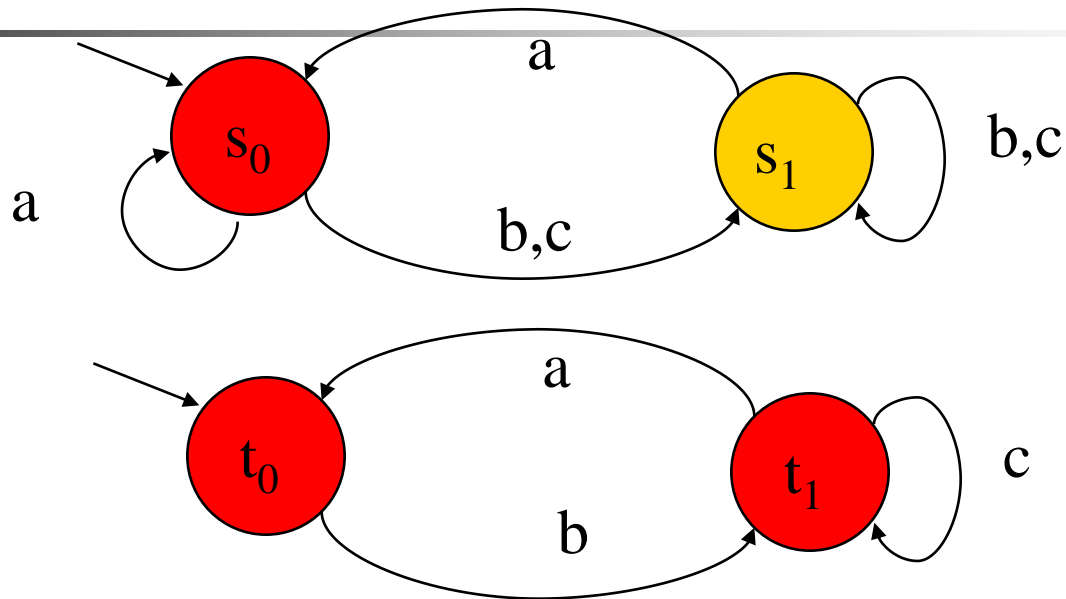


# Intersecting $M_1 = (S_1, \Sigma, T_1, I_1, A_1)$ and $M_2 = (S_2, \Sigma, T_2, I_2, S_2)$

---

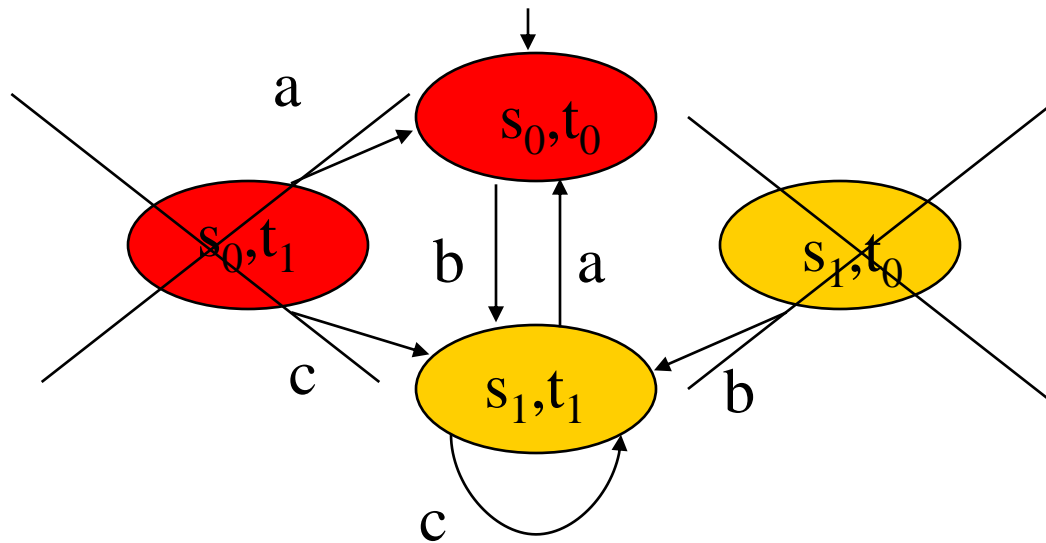
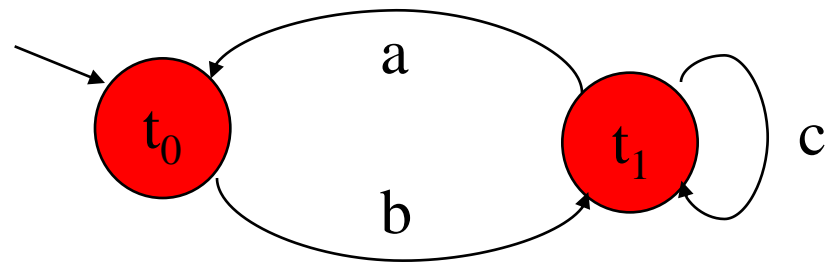
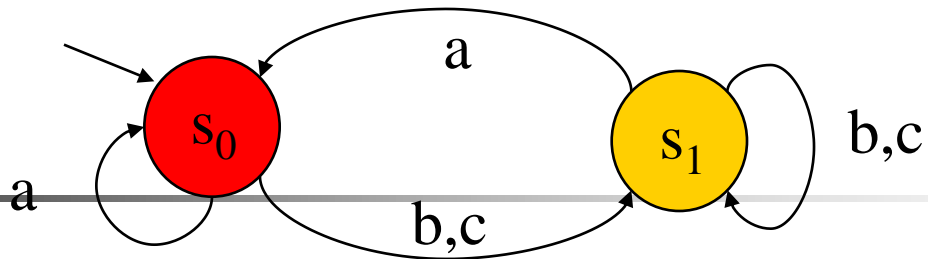
- Run the two automata in parallel.
- Each state is a pair of states:  $S_1 \times S_2$
- Initial states are pairs of initials:  $I_1 \times I_2$
- Acceptance depends on first component:  $A_1 \times S_2$
- Conforms with transition relation:  
 $(x_1, y_1) - a \rightarrow (x_2, y_2)$  when  
 $x_1 - a \rightarrow x_2$  and  $y_1 - a \rightarrow y_2$ .

# Example (all states of second automaton accepting!)

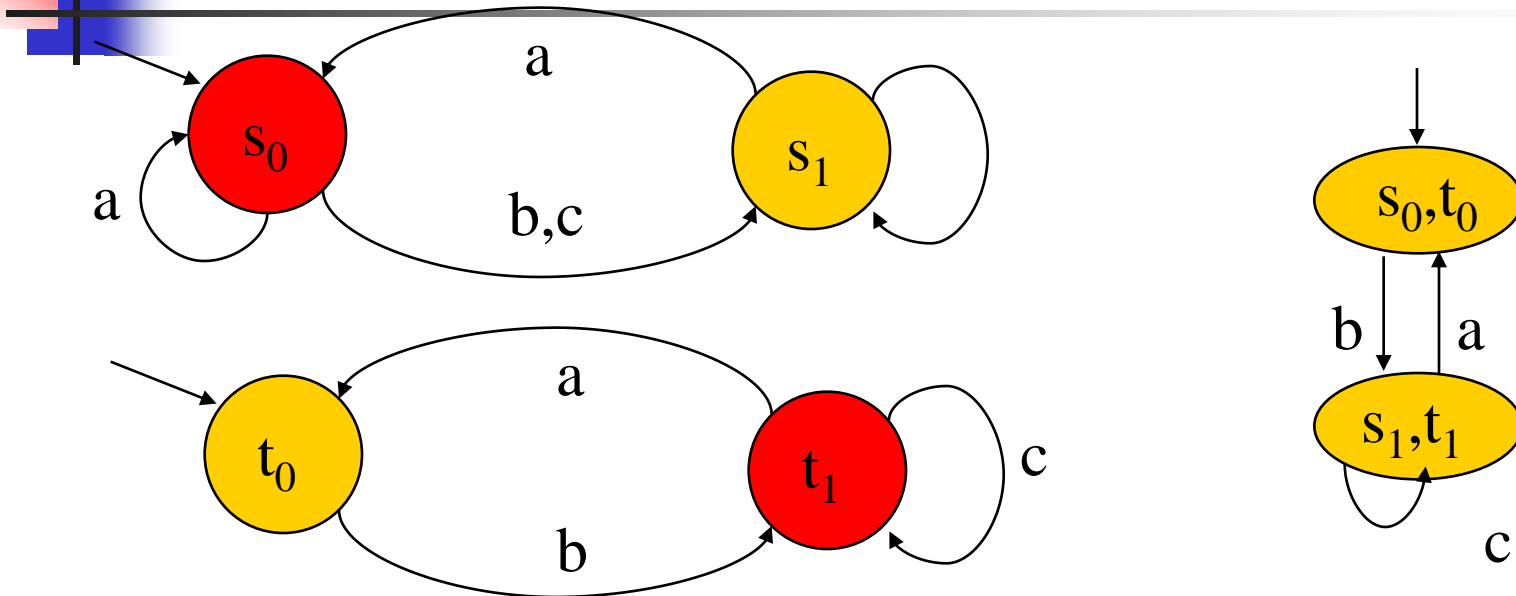


States:  $(s_0, t_0), (s_0, t_1), (s_1, t_0), (s_1, t_1)$ .

Accepting:  $(s_0, t_0), (s_0, t_1)$ . Initial:  $(s_0, t_0)$ .



# More complicated when $A_2 \neq S_2$



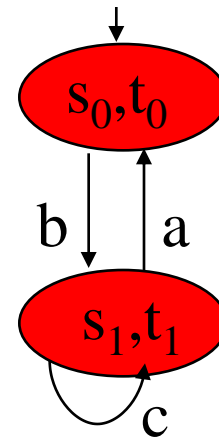
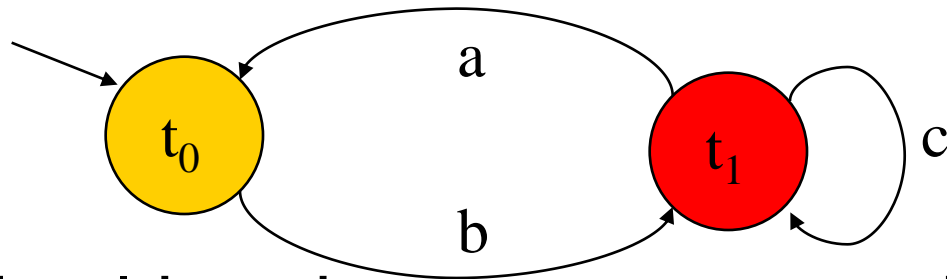
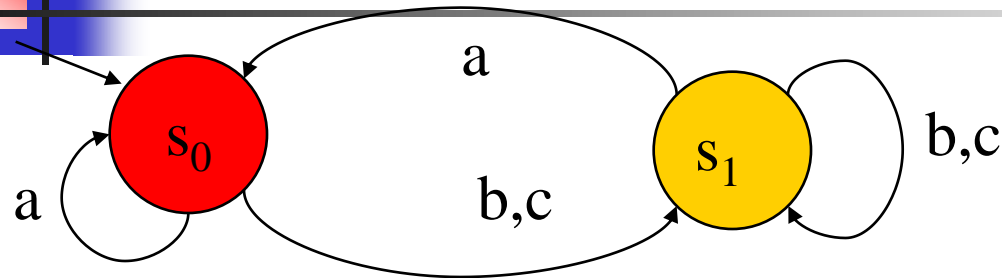
Should we have acceptance when both components accepting? I.e.,  $\{(s_0, t_1)\}$ ?

No, consider  $(ba)^\omega$

It should be accepted, but never passes that state.



# More complicated when $A_2 \neq S_2$

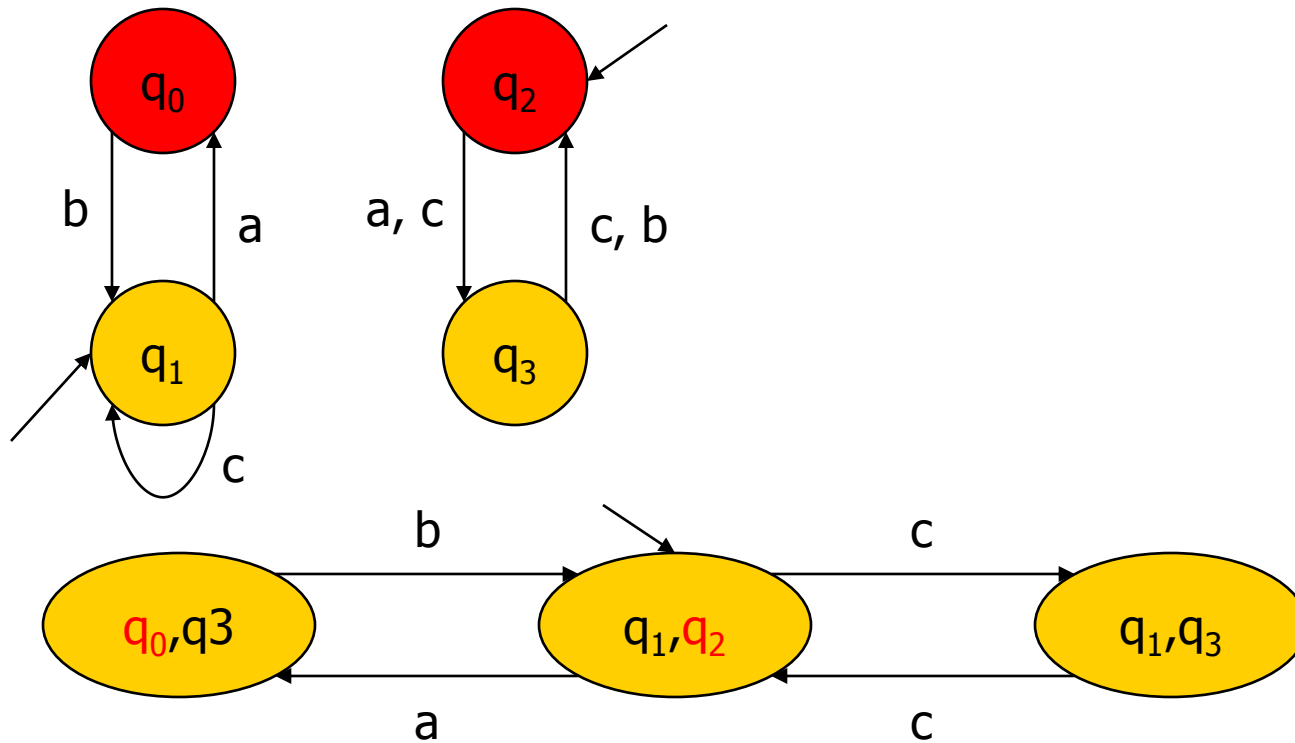


Should we have acceptance when at least one component is accepting? I.e.,  $\{(s_0, t_0), (s_0, t_1), (s_1, t_1)\}$ ?

No, consider  $b c^\omega$

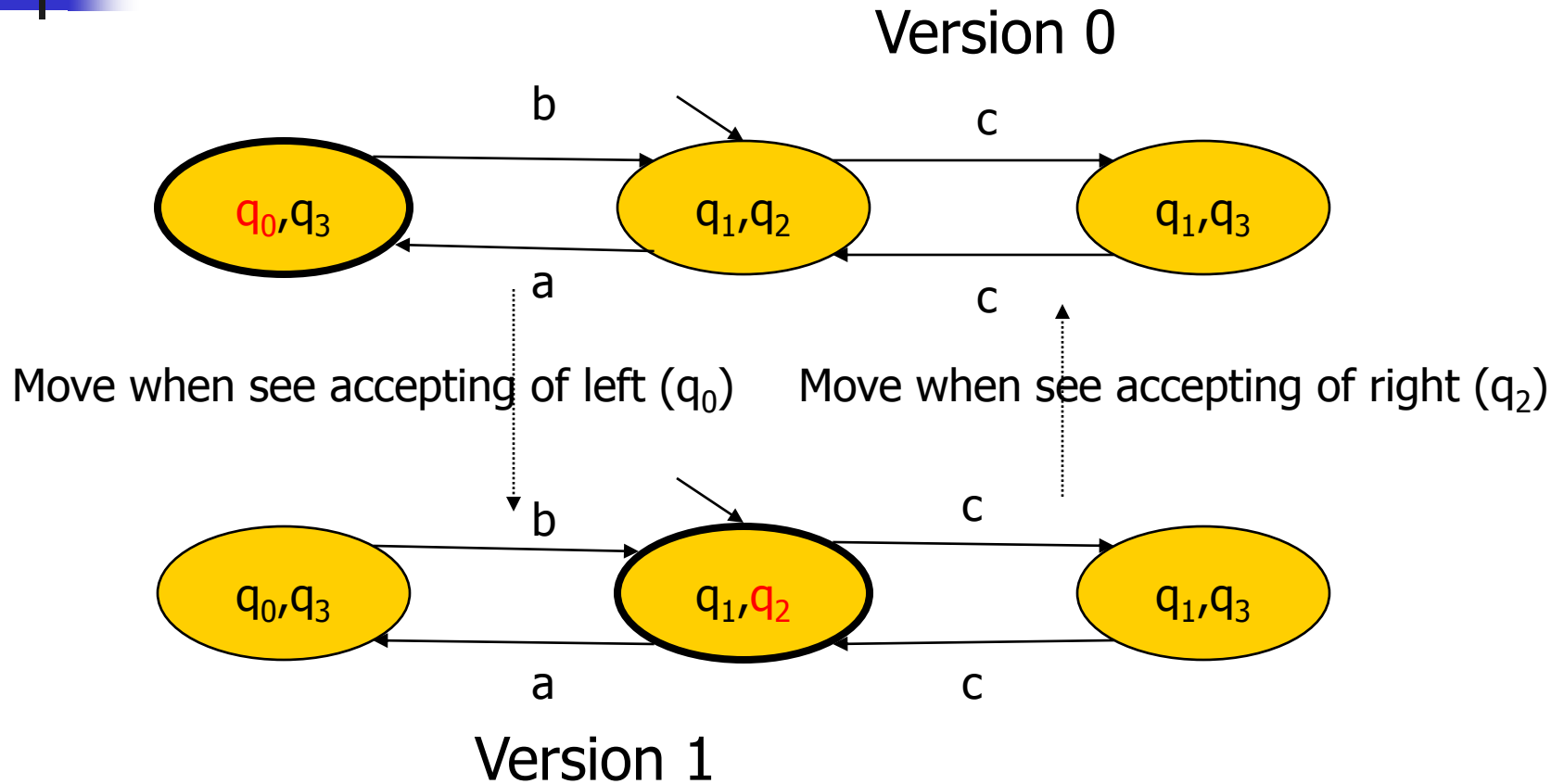
It should not be accepted, but here will loop through  $(s_1, t_1)$

# Intersection - general case



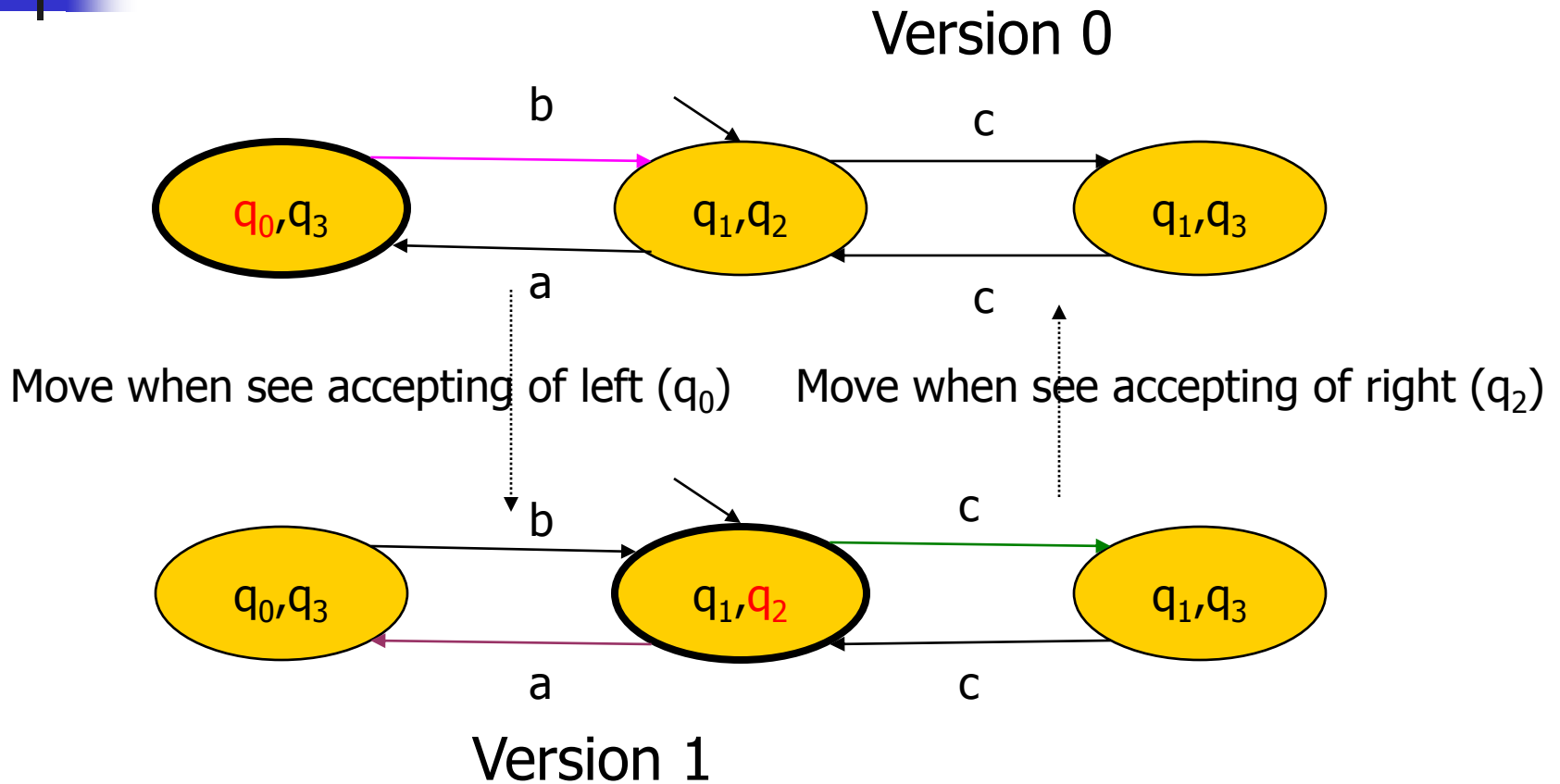
# Version 0: to catch $q_0$

## Version 1: to catch $q_2$

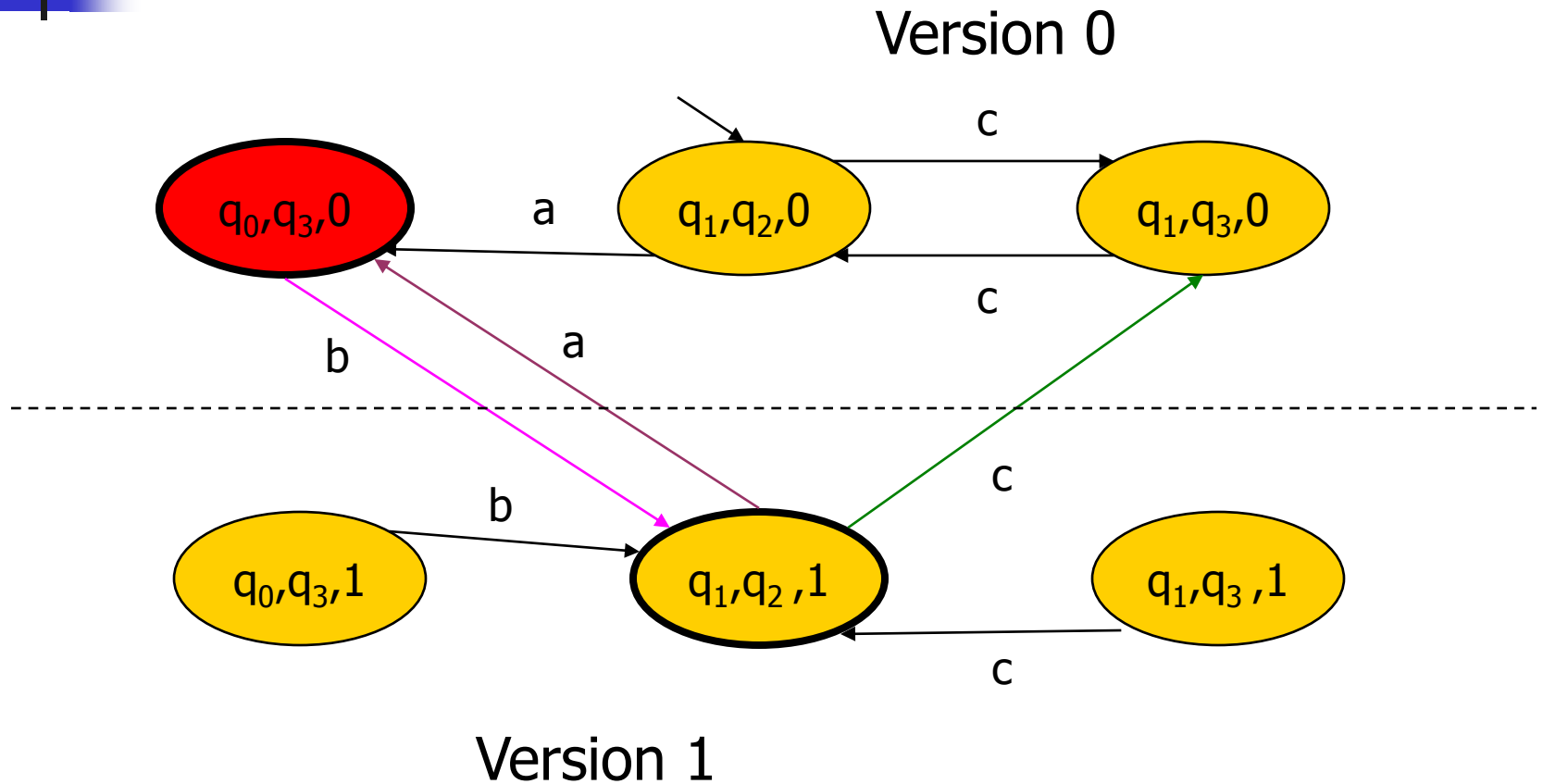


Version 0: to catch  $q_0$

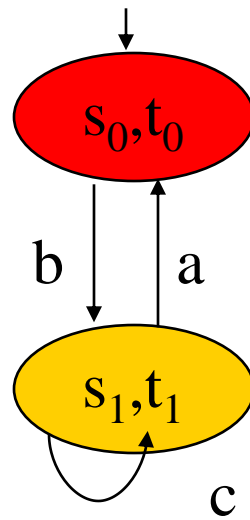
Version 1: to catch  $q_2$



Make an accepting state in one of the version according to a component accepting state

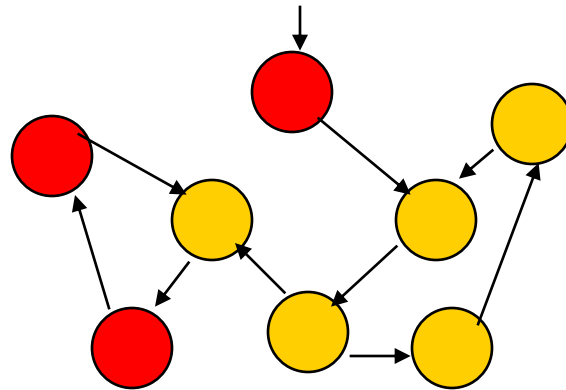


# How to check for emptiness?



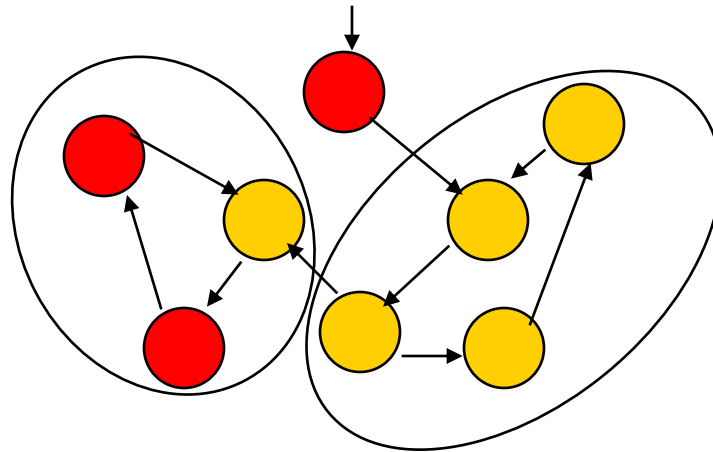
# Emptiness...

Need to check if there exists an accepting run (passes through an accepting state infinitely often).



# Strongly Connected Component (SCC)

A set of states with a path between each pair of them.



Can use Tarjan's DFS algorithm for finding maximal SCC's.



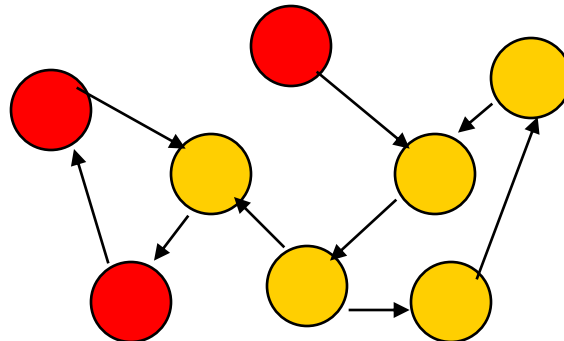
# Finding accepting runs

If there is an accepting run, then at least one accepting state appears on it infinitely often.

Look at a suffix of this run where *all the states appear infinitely often*.

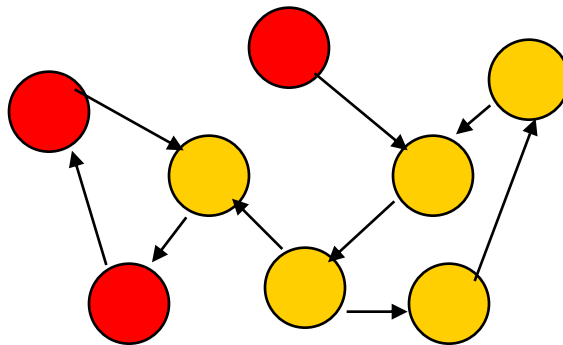
These states form a strongly connected component on the automaton graph, including an accepting state.

Find a component like that and form an accepting cycle including the accepting state.



# That is...

- A strongly connected component: a set of nodes where each node is reachable by a path from each other node. Find if there is a (maximal) reachable strongly connected component with an accepting node.
- If there is such a reachable component we can form a cycle through an accepting state (and vice versa!)

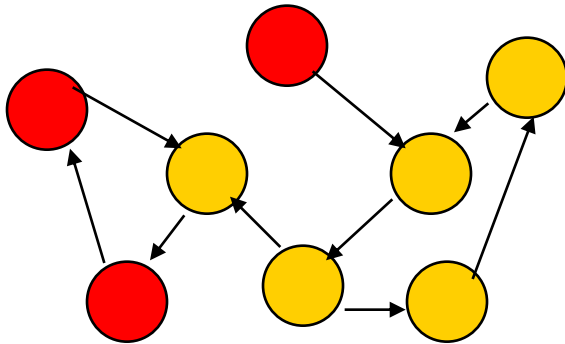


The system does not satisfy its specification if and only if there is a “lasso” shape through an accepting state.

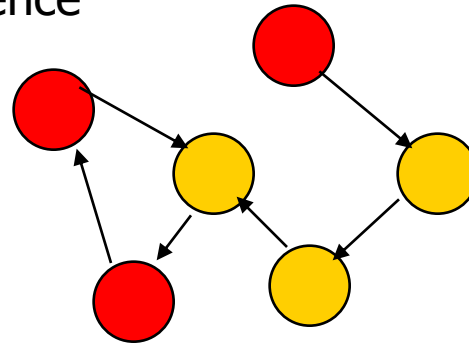
# Catching bugs with a lasso...

- Use a double DFS algorithm to find reachable cycles with accepting state “on-the-fly”.
- “Ultimately periodic” (lasso) sequences allows reporting errors using finite representation.

State space



Ultimately periodic accepted sequence



# How to complement a Buchi automaton?



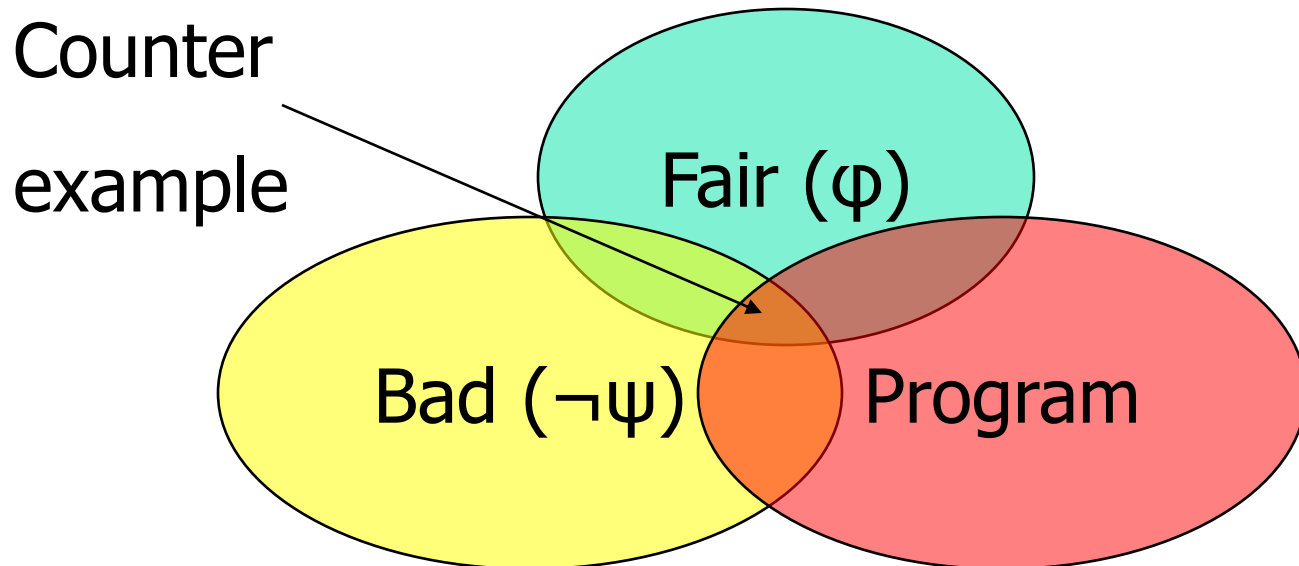
---

- “Its complicated” [Facebook, 2007]
- Can ask for the negated property (the sequences that should never occur).
- Can translate from LTL formula  $\varphi$  to automaton  $A$ , and complement  $A$ . But: can translate  $\neg\varphi$  into an automaton directly!

# Model Checking under Fairness

Express the fairness as a property  $\varphi$ .

To prove a property  $\psi$  under fairness,  
model check  $\varphi \rightarrow \psi$ .





# Model Checking under Fairness

---

Specialize model checking. For weak process fairness: search for a reachable strongly connected component, where for each process  $P$  either

- it contains an occurrence of a transition from  $P$ , or
- it contains a state where  $P$  is disabled.



# Translating from logic to automata

---

(Book: Chapter 6)



# Why translating?

---

- Want to write the specification in some logic.
- Want model-checking tools to be able to check the specification automatically.



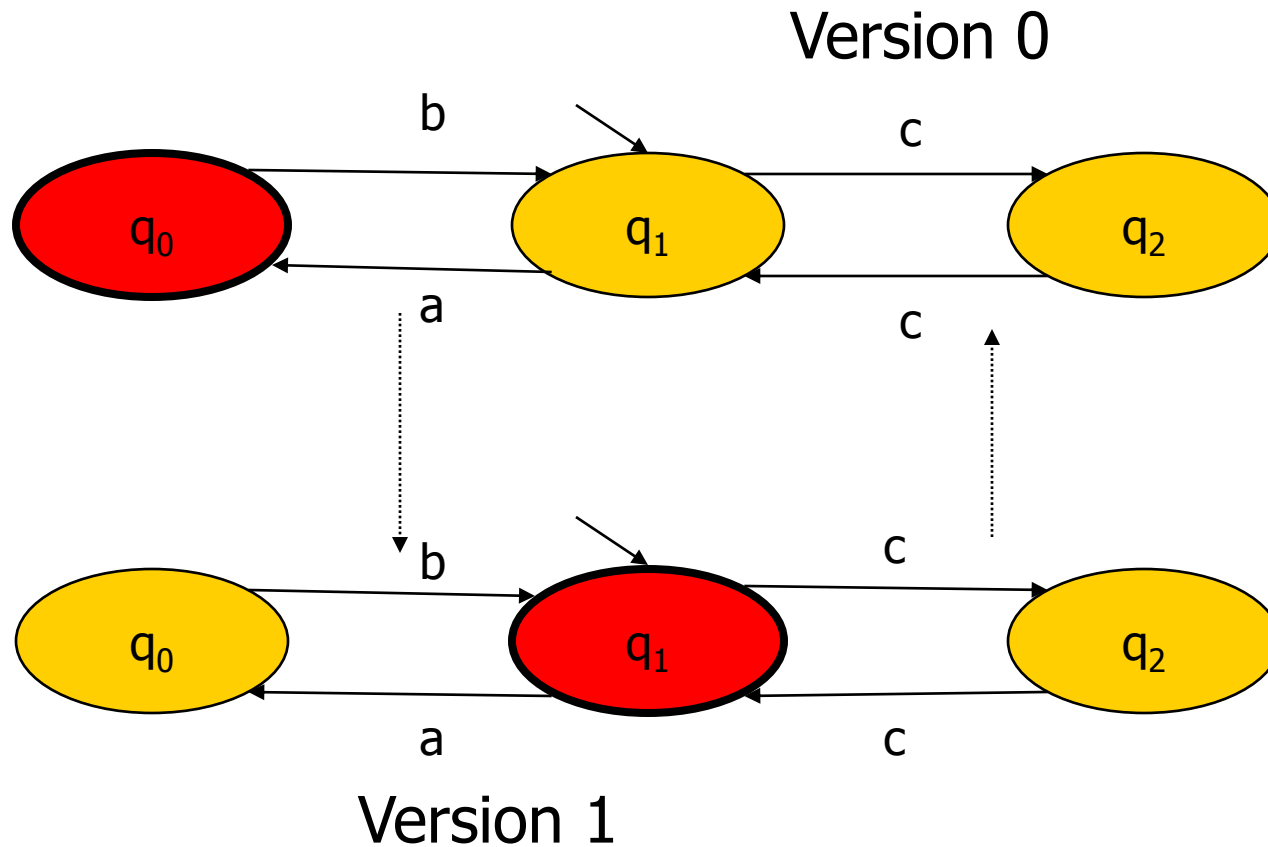


# Generalized Büchi automata

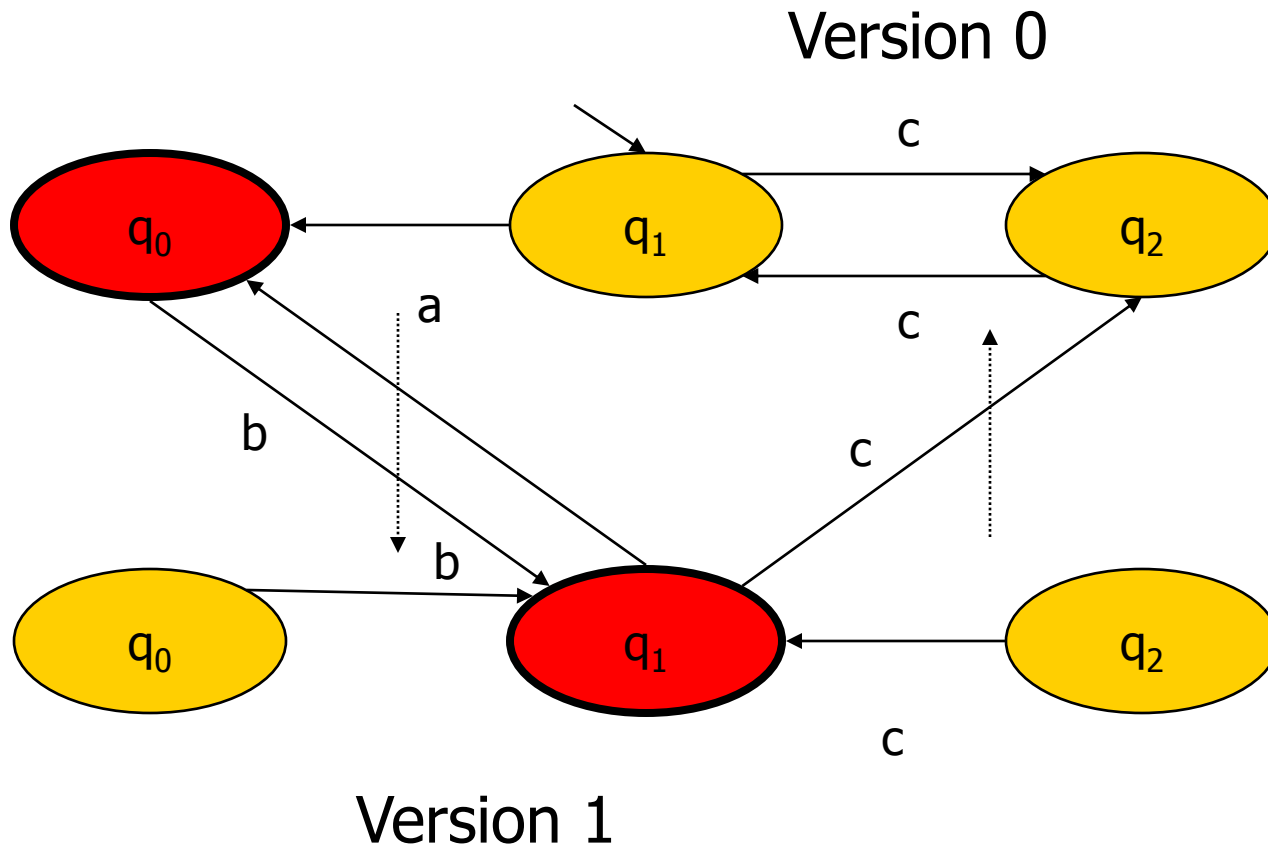
---

- Acceptance condition  $F$  is a set  $F = \{f_1, f_2, \dots, f_n\}$  where each  $f_i$  is a set of states.
- To accept, a run needs to pass infinitely often through a state from every set  $f_i$ .

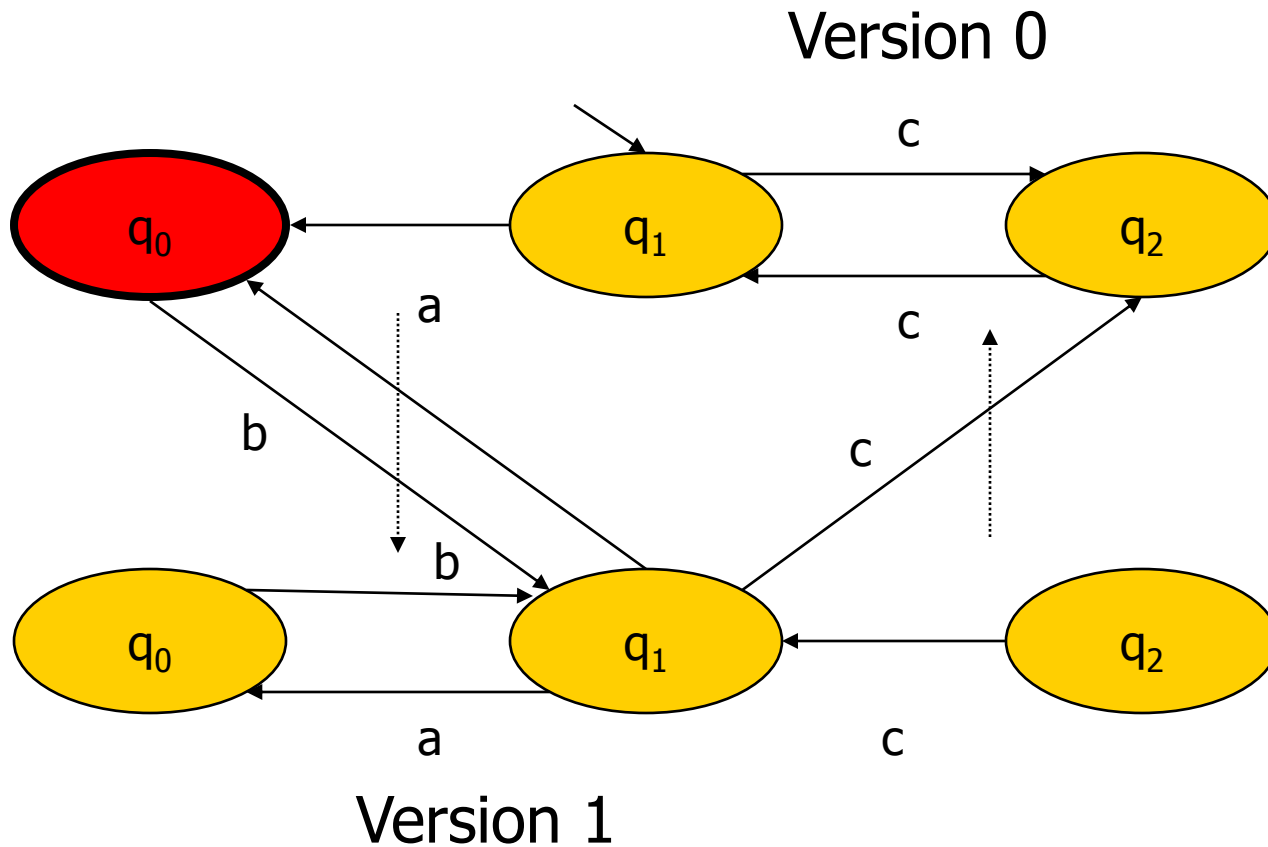
# Translating into simple Büchi automaton



# Translating into simple Büchi automaton



# Translating into simple Büchi automaton





# Preprocessing

---

- Convert into normal form, where negation only applies to propositional variables.
- $\neg[\ ]\varphi$  becomes  $\langle \rangle \neg\varphi$ .
- $\neg\langle \rangle\varphi$  becomes  $[\ ] \neg\varphi$ .
- What about  $\neg(\varphi \cup \psi)$ ?
- Define operator  $R$  such that
$$\neg(\varphi \cup \psi) = (\neg\varphi) R (\neg\psi),$$
$$\neg(\varphi R \psi) = (\neg\varphi) \cup (\neg\psi).$$

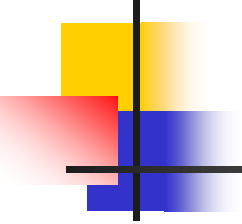


# Semantics of $pRq$

---

$\neg p$	$\neg p$	$\neg p$	$\neg p$	$\neg p$	$\neg p$	$\neg p$	$\neg p$	$\neg p$
$q$	$q$	$q$	$q$	$q$	$q$	$q$	$q$	$q$

$\neg p$	$\neg p$	$\neg p$	$\neg p$	$p$				
$q$	$q$	$q$	$q$	$q$				

- 
- 
- Replace  $\neg$ true by false, and  $\neg$ false by true.
  - Replace  $\neg (\varphi \vee \psi)$  by  $(\neg\varphi) \wedge (\neg\psi)$  and  $\neg (\varphi \wedge \psi)$  by  $(\neg\varphi) \vee (\neg\psi)$



# Eliminate implications, $\leftrightarrow$ , $\square$

---

- Replace  $\varphi \rightarrow \psi$  by  $(\neg \varphi) \vee \psi$ .
- Replace  $\leftrightarrow \varphi$  by  $(\text{true} \cup \varphi)$ .
- Replace  $\square \varphi$  by  $(\text{false} \cap \varphi)$ .



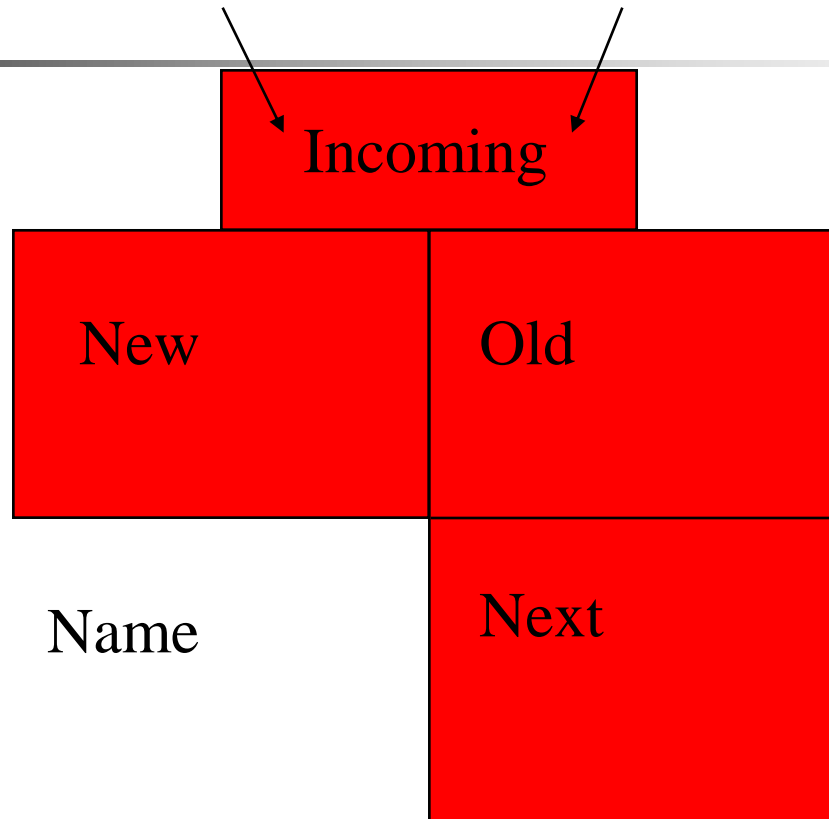


# Example

---

- Translate  $( \Box \leftrightarrow P ) \rightarrow ( \Box \leftrightarrow Q )$
- Eliminate implication  $\neg( \Box \leftrightarrow P ) \vee ( \Box \leftrightarrow Q )$
- Eliminate  $\Box, \leftrightarrow$ :  
 $\neg( \text{false } R( \text{true } \cup P ) ) \vee ( \text{false } R( \text{true } \cup Q ) )$
- Push negation inwards:  
 $( \text{true } \cup ( \text{false } R \neg P ) ) \vee ( \text{false } R( \text{true } \cup Q ) )$

# The data structure





# The main idea

---

- $\varphi \ U \ \psi = \psi \ \vee \ ( \varphi \ \wedge \ O \ ( \varphi \ U \ \psi ) )$
- $\varphi \ R \ \psi = \psi \ \wedge \ ( \varphi \ \vee \ O \ ( \varphi \ R \ \psi ) )$

This separates the formulas to two parts:  
one holds in the current state, and the other  
in the next state.



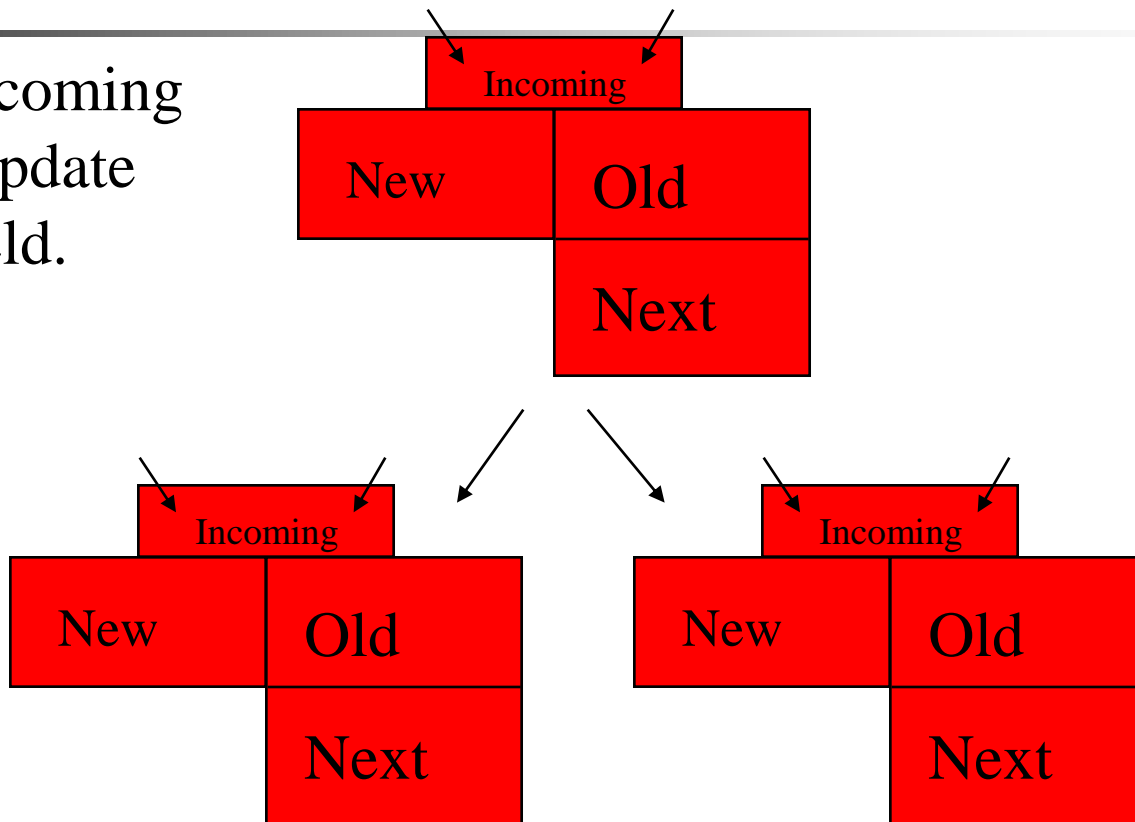
# How to translate?

---

- Take one formula from “New” and add it to “Old”.
- According to the formula, either
  - Split the current node into two, or
  - Evolve the node into a new version.

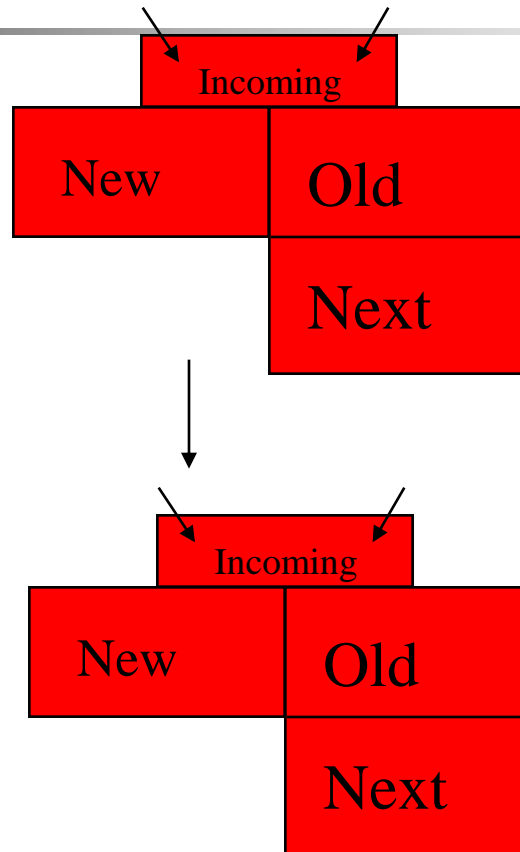
# Splitting

Copy incoming edges, update other field.



# Evolving

Copy incoming  
edges, update  
other field.





# Possible cases:

---

- $\varphi \cup \psi$ , split:

- Add  $\varphi$  to New, add  $\varphi \cup \psi$  to Next.
- Add  $\psi$  to New.

Because  $\varphi \cup \psi = \psi \vee (\varphi \wedge \neg (\varphi \cup \psi))$ .

- $\varphi \cap \psi$ , split:

- Add  $\varphi, \psi$  to New.
- Add  $\varphi \cap \psi$  to Next.

Because  $\varphi \cap \psi = \psi \wedge (\varphi \vee \neg (\varphi \cap \psi))$ .



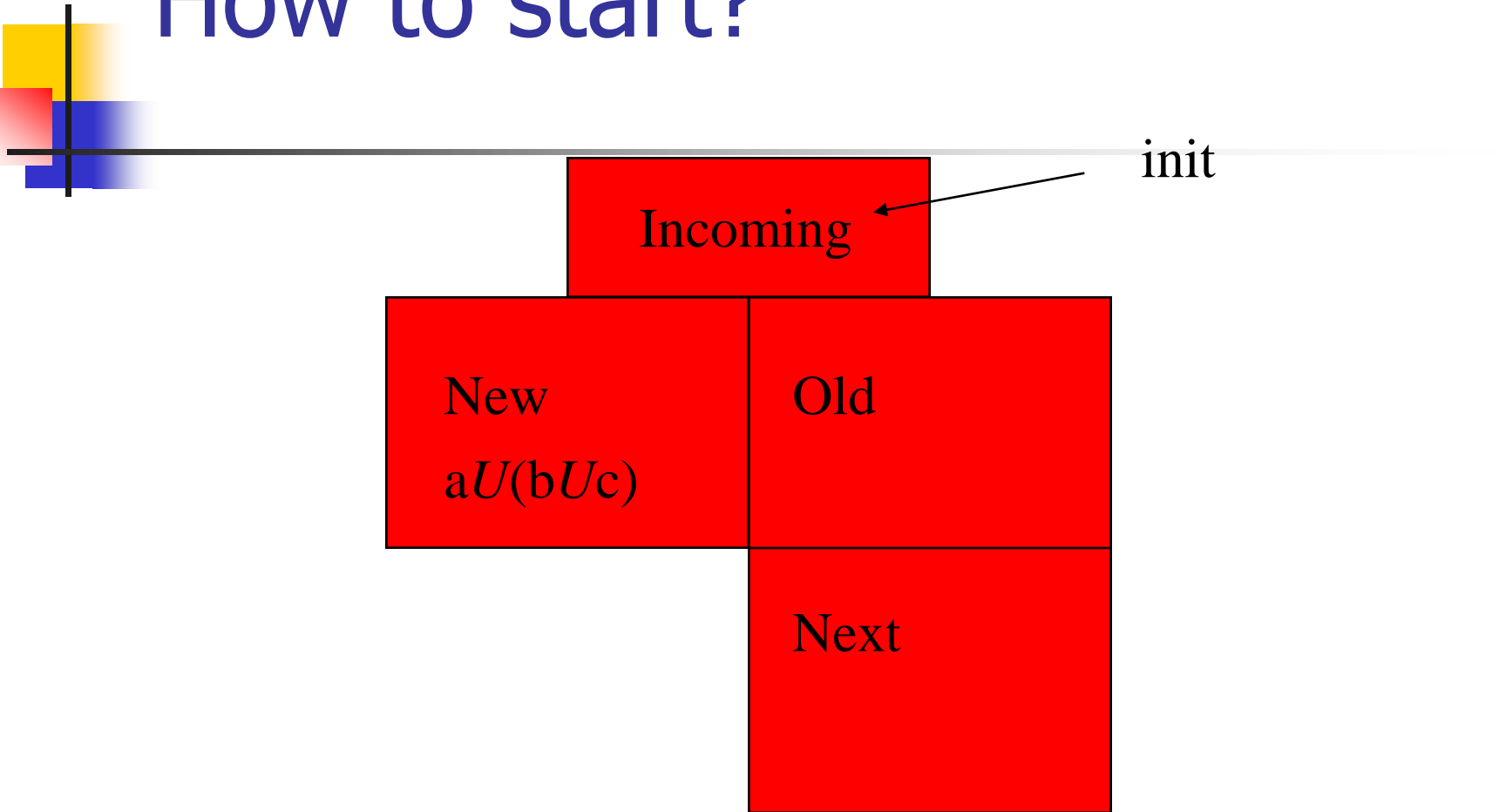
# More cases:

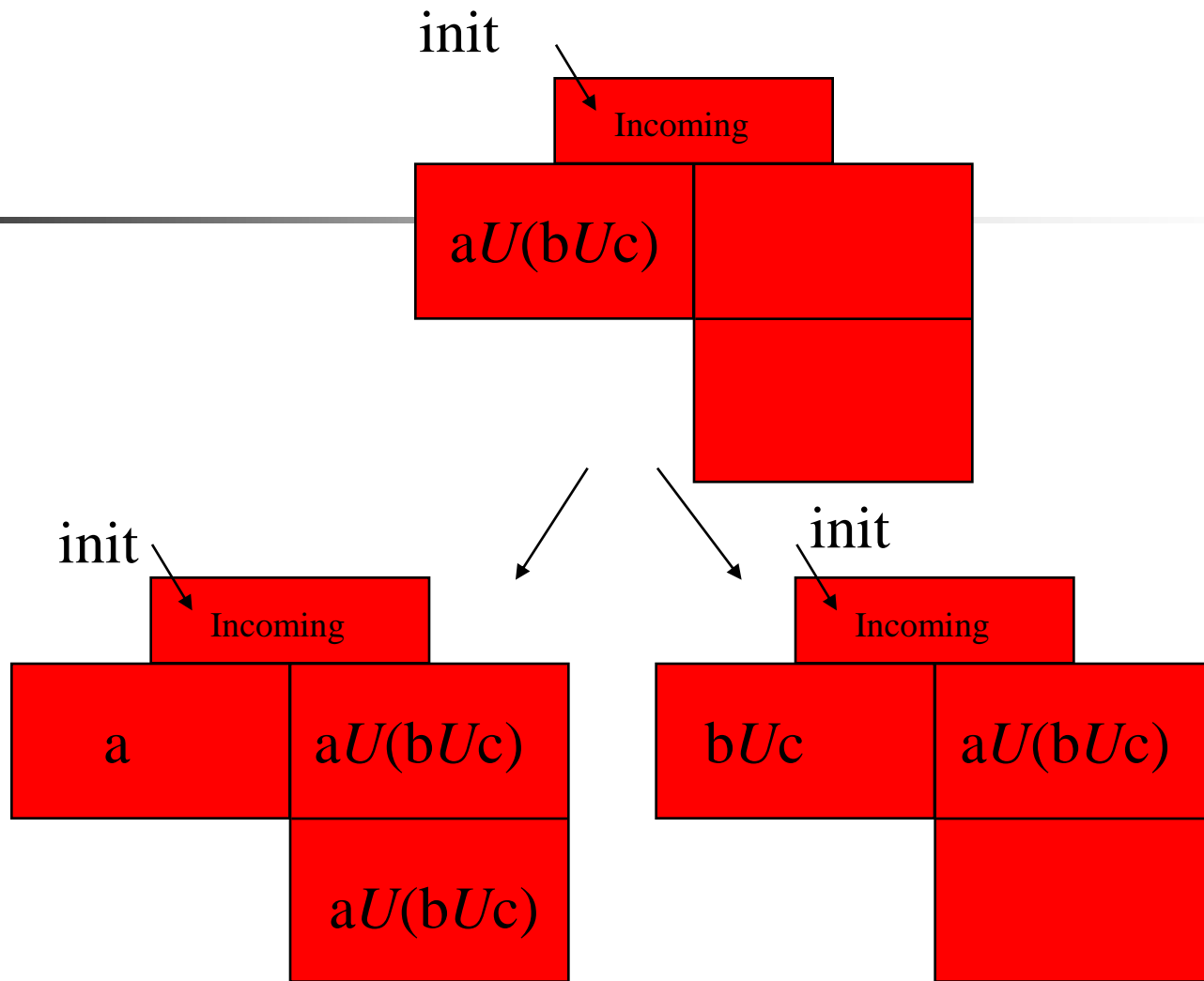
---

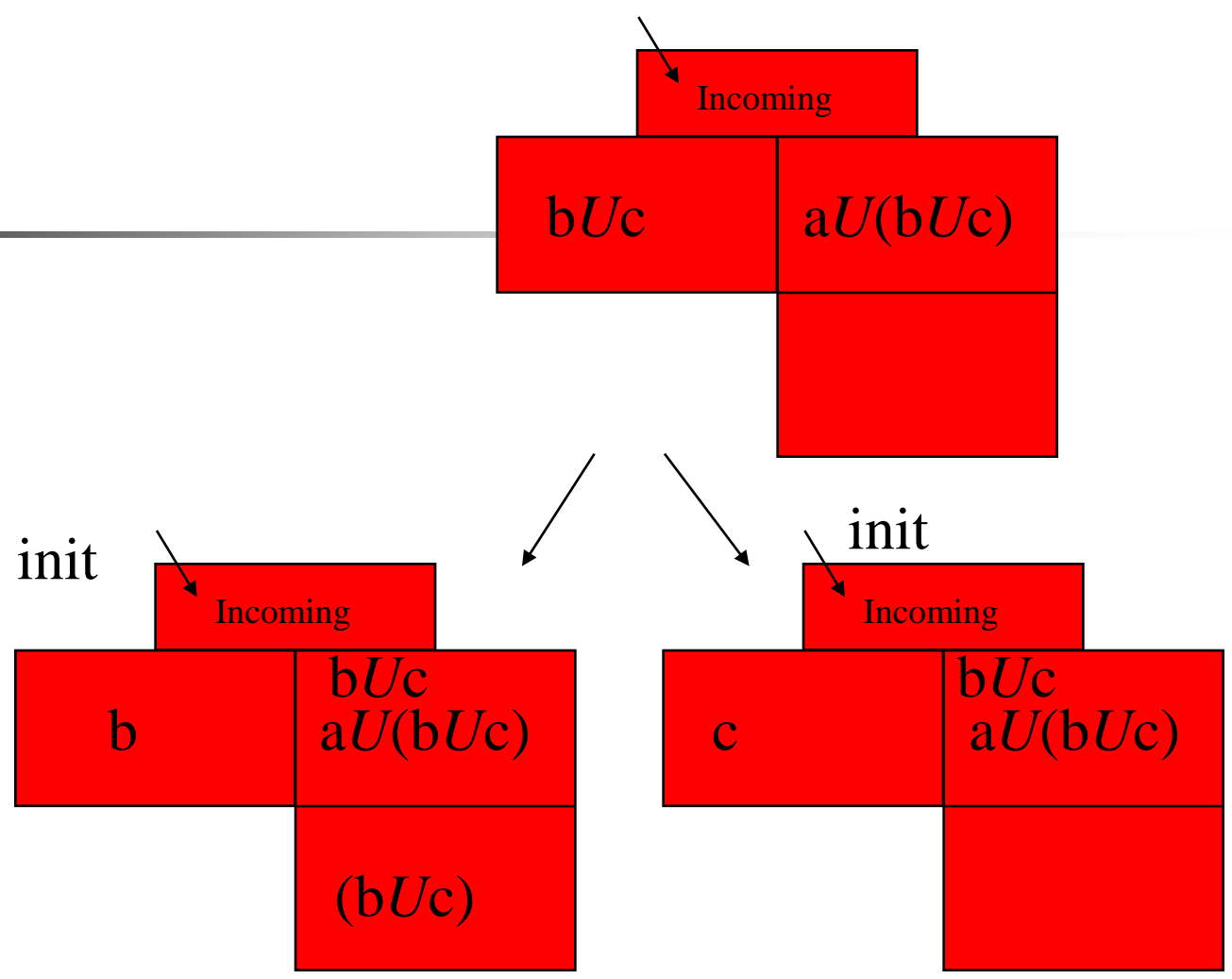
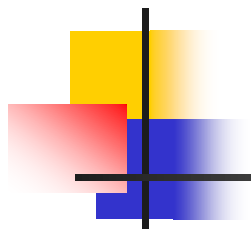
- $\varphi \vee \psi$ , split:
  - Add  $\varphi$  to New.
  - Add  $\psi$  to New.
- $\varphi \wedge \psi$ , evolve:
  - Add  $\varphi, \psi$  to New.
- $\text{O } \varphi$ , evolve:
  - Add  $\varphi$  to Next.



# How to start?









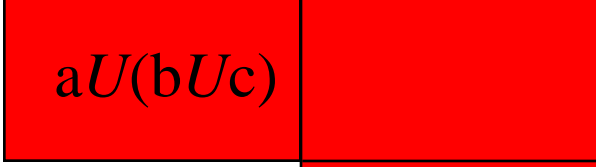
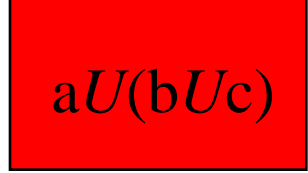
# When to stop splitting?

---

- When “New” is empty.
- Then compare against a list of existing nodes “Nodes”:
  - If such a with same “Old”, “Next” exists, just add the incoming edges of the new version to the old one.
  - Otherwise, add the node to “Nodes”. Generate a successor with “New” set to “Next” of father.



init



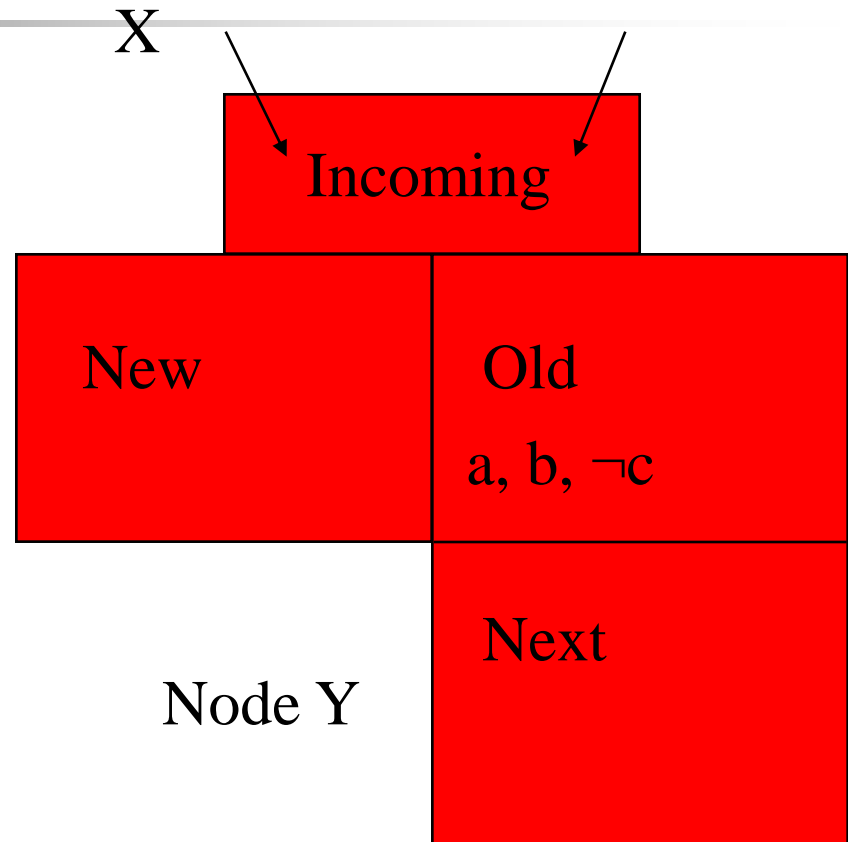
Creating a successor node.

When we enter to Nodes a new node (with different Old or Next than any other node), we start a new node by copying Next to New, and making an edge to the new successor.

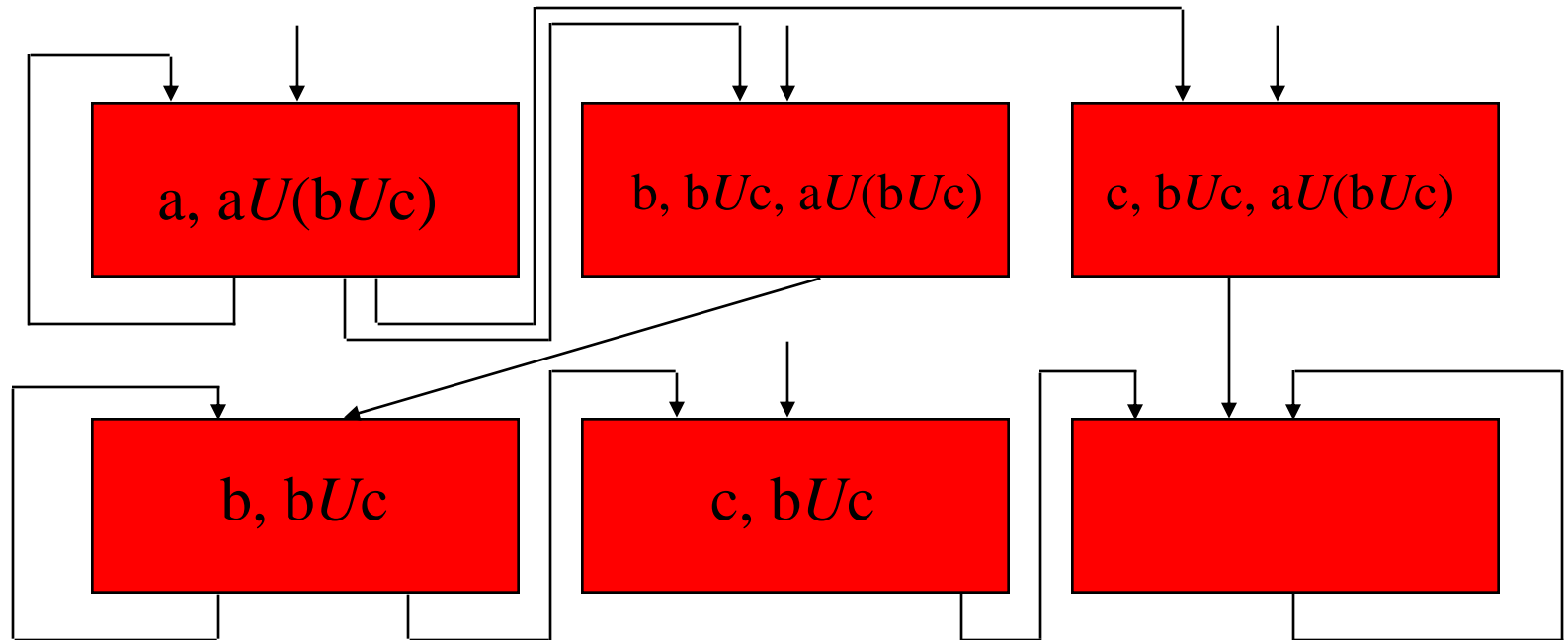
# How to obtain the automaton?

There is an edge from node X to Y labeled with propositions P (negated or non negated), if X is in the incoming list of Y, and Y has propositions P in field "Old".

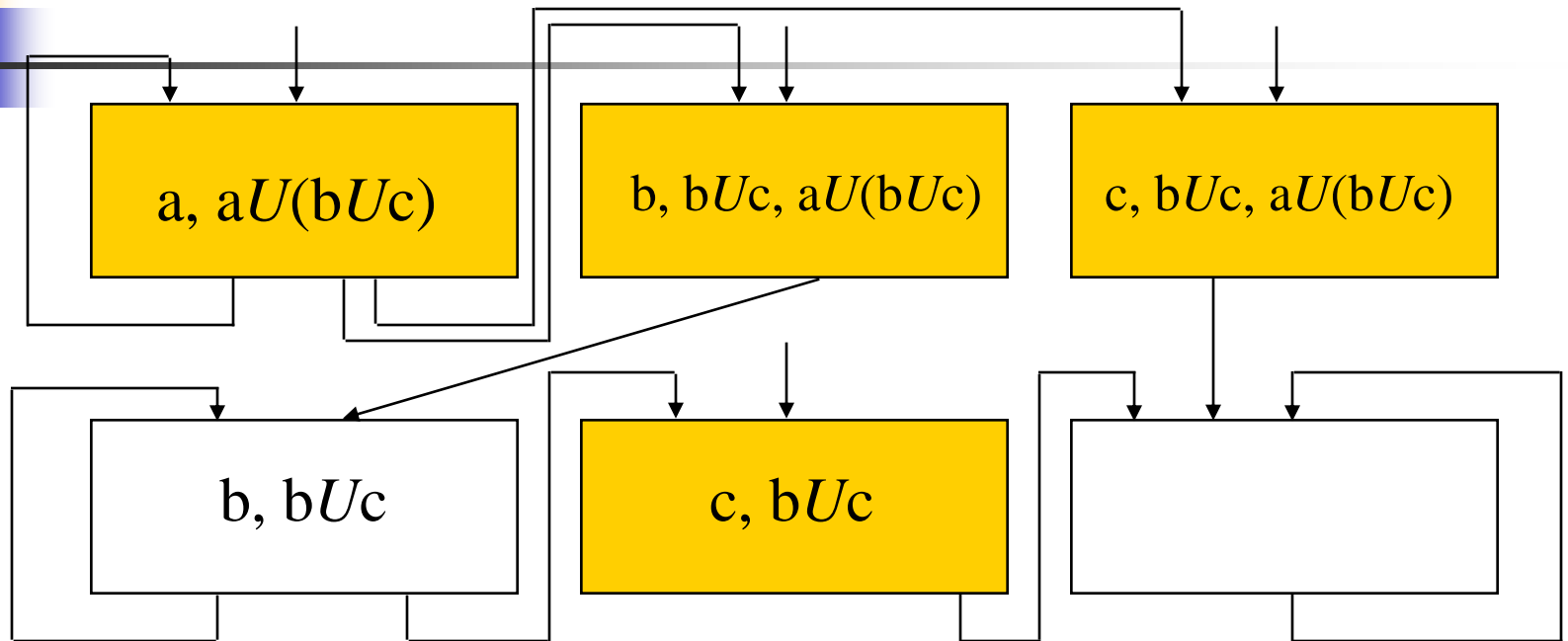
Initial node is init.



# The resulted nodes.



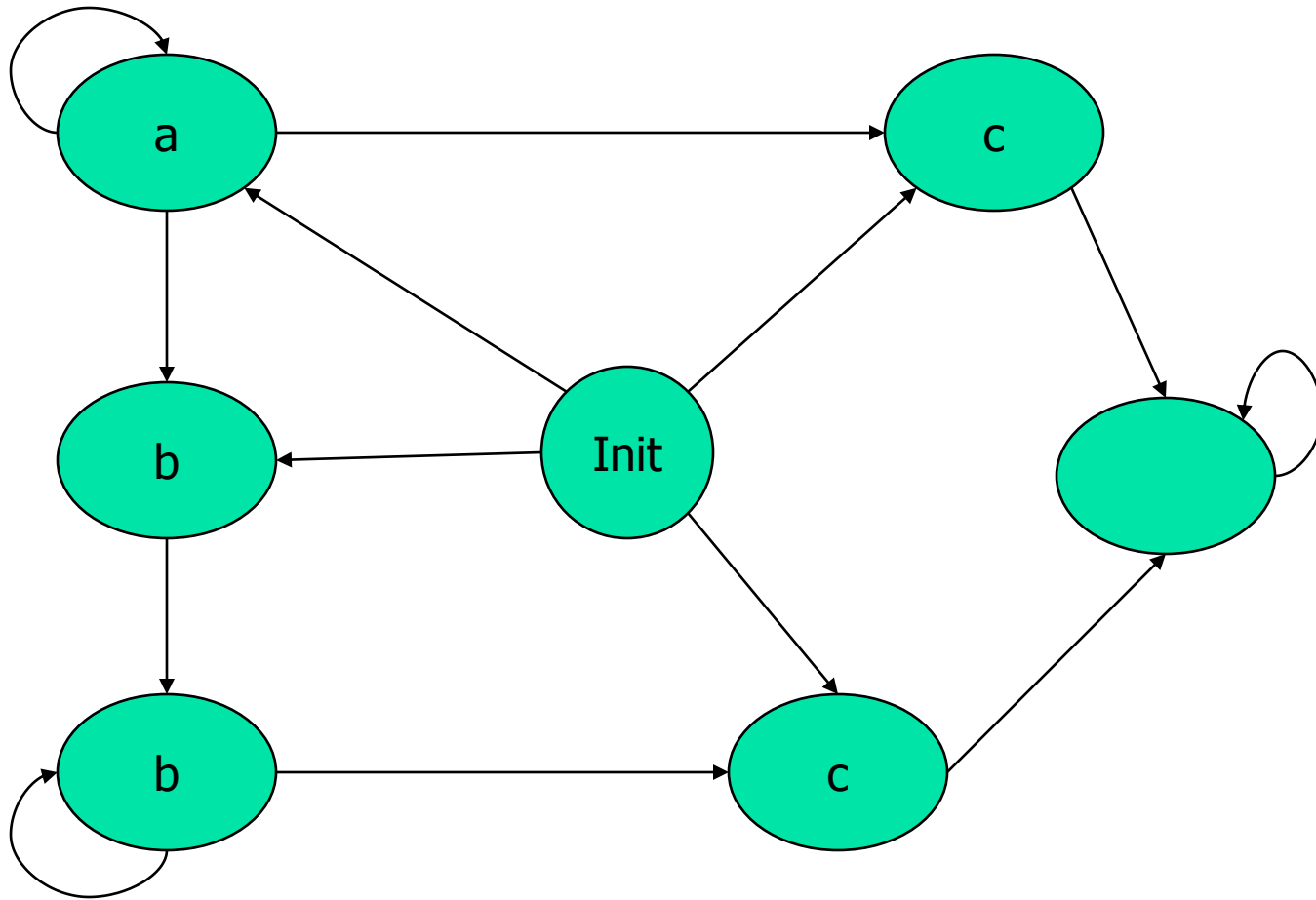
# Initial nodes



*All nodes with incoming edge from "init".*



# Include only atomic propositions



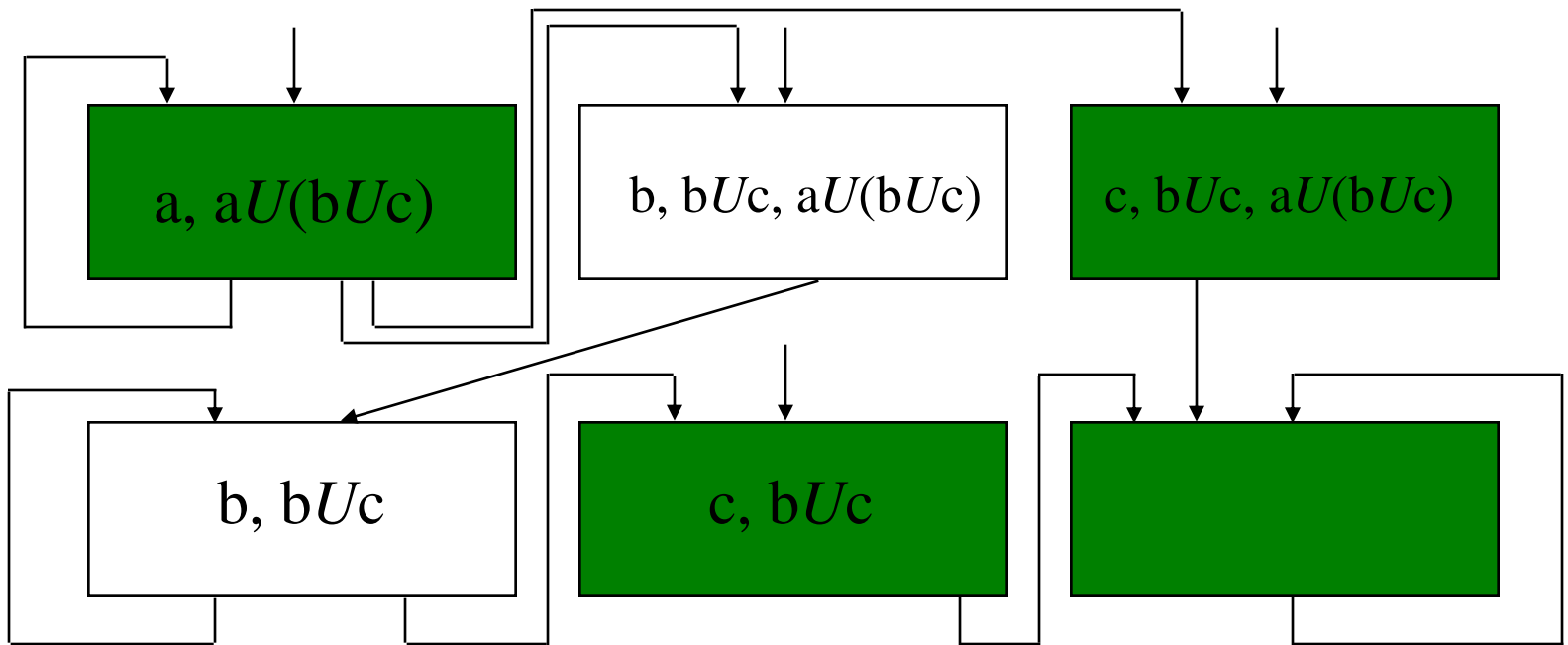


# Acceptance conditions

---

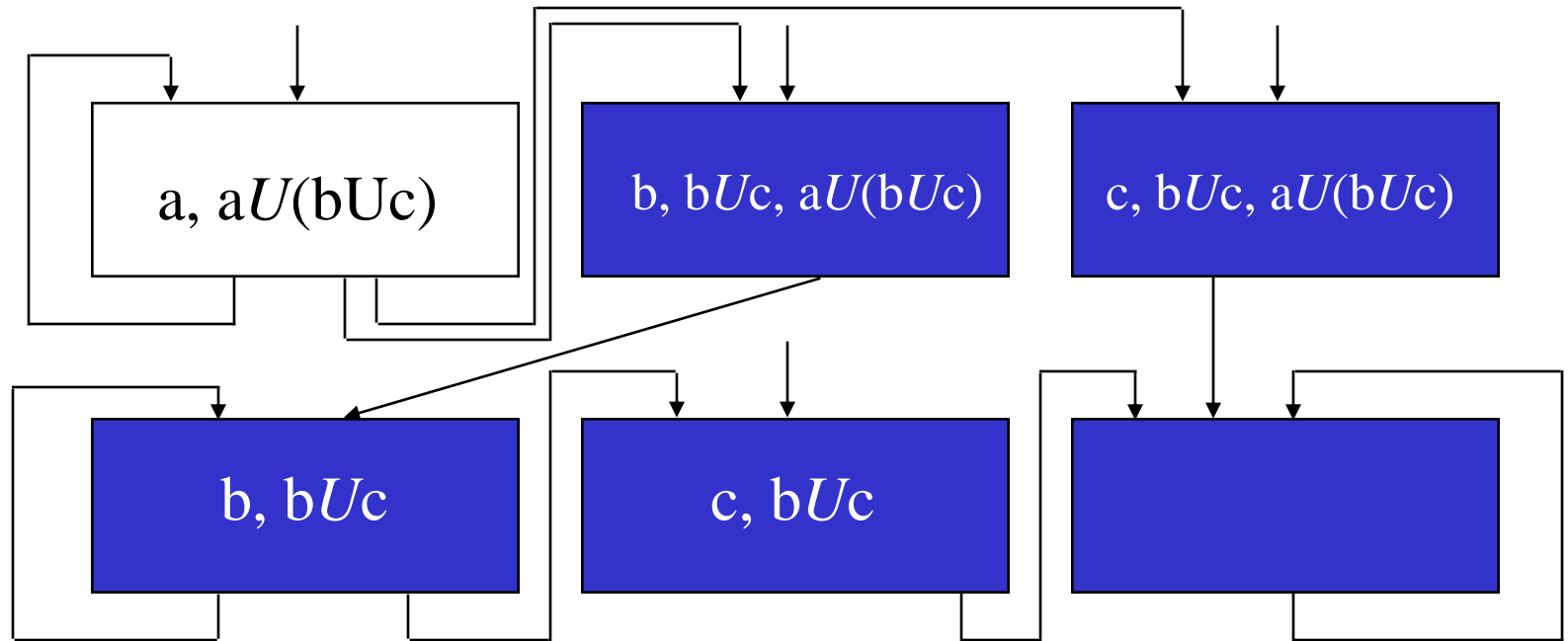
- Use “generalized Buchi automata”, where there are several acceptance sets  $f_1, f_2, \dots, f_n$ , and each accepted infinite sequence must include at least one state from each set infinitely often.
- Each set corresponds to a subformula of form  $\varphi \mathcal{U} \psi$ . Guarantees that it is never the case that  $\varphi \mathcal{U} \psi$  holds forever, without  $\varphi$ .

# Accepting w.r.t. $bUc$



*All nodes with  $c$ , or without  $bUc$ .*

# Acceptance w.r.t. $aU(bUc)$



*All nodes with  $bUc$  or without  $aU(bUc)$ .*



# Complexity!!!

---

- Model checking is complete for PSPACE both in the “size of the model” and the LTL property.
- “Size of the model” is the accumulated sizes of the processes.
- The upper bound is proved by NOT explicitly constructing the actual global state space graph or Buchi automaton for the LTL property, but rather performing a binary search over their product.
- All practical algorithms actually use exponential time and space.



# What is SPIN?

---

- Model-checker.
- Based on automata theory.
- Allows LTL or automata specification.
- Efficient (on-the-fly model checking, partial order reduction).
- Developed in Bell Laboratories.



# Dekker's algorithm

boolean c1 initially 1;  
boolean c2 initially 1;  
integer (1..2) turn initially 1;

```
P1::while true do
  begin
    non-critical section 1
    c1:=0;
    while c2=0 do
      begin
        if turn=2 then
          begin
            c1:=1;
            wait until turn=1;
            c1:=0;
          end
        end
      critical section 1
      c1:=1;
      turn:=2
    end.
```

```
P2::while true do
  begin
    non-critical section 2
    c2:=0;
    while c1=0 do
      begin
        if turn=1 then
          begin
            c2:=1;
            wait until turn=2;
            c2:=0;
          end
        end
      critical section 2
      c2:=1;
      turn:=1
    end.
```

# Dekker's algorithm

```
boolean c1 initially 1;  
boolean c2 initially 1;  
integer (1..2) turn initially 1;
```

P1::while true do

begin

non-critical section 1

c1:=0;

while c2=0 do

begin

if turn=2 then

begin

c1:=1;

wait until turn=1;

c1:=0;

end

end

critical section 1

c1:=1;

turn:=2

end.

c1=c2=0,  
turn=1

P2::while true do

begin

non-critical section 2

c2:=0;

while c1=0 do

begin

if turn=1 then

begin

c2:=1;

wait until turn=2;

c2:=0;

end

end

critical section 2

c2:=1;

turn:=1

end.



# Dekker's algorithm

```
boolean c1 initially 1;  
boolean c2 initially 1;  
integer (1..2) turn initially 1;
```

```
P1::while true do
```

```
begin
```

```
non-critical section 1
```

```
c1:=0;
```

```
while c2=0 do
```

```
begin
```

```
if turn=2 then
```

```
begin
```

```
c1:=1;
```

```
wait until turn=1;
```

```
c1:=0;
```

```
end
```

```
end
```

```
critical section 1
```

```
c1:=1;
```

```
turn:=2
```

```
end.
```

```
c1=c2=0,  
turn=1
```

```
P2::while true do
```

```
begin
```

```
non-critical section 2
```

```
c2:=0;
```

```
while c1=0 do
```

```
begin
```

```
if turn=1 then
```

```
begin
```

```
c2:=1;
```

```
wait until turn=2;
```

```
c2:=0;
```

```
end
```

```
end
```

```
critical section 2
```

```
c2:=1;
```

```
turn:=1
```

```
end.
```

# Dekker's algorithm

P1 waits for P2 to set c2 to 1 again. Since turn=1 (priority for P1), P2 is ready to do that. But never gets the chance, since P1 is constantly active checking c2 in its while loop.

P1::while true do

begin

non-critical section 1

c1:=0;

while c2=0 do  
begin

if turn=2 then

begin

c1:=1;

wait until turn=1;

c1:=0;

end

end

critical section 1

c1:=1;

turn:=2

end.

c1=c2=0,  
turn=1

P2::while true do

begin

non-critical section 2

c2:=0;

while c1=0 do

begin

if turn=1 then

begin

c2:=1;

wait until turn=2;

c2:=0;

end

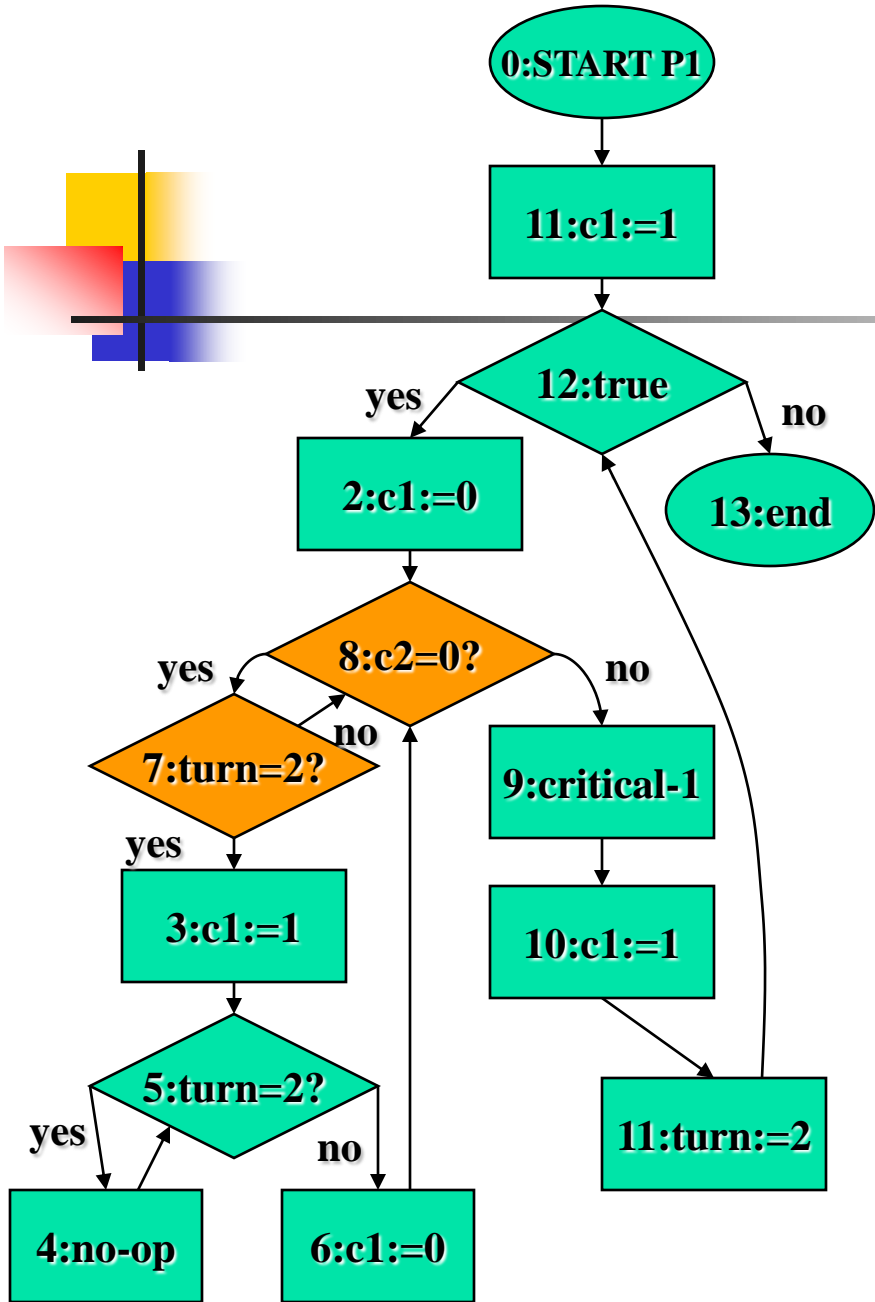
end

critical section 2

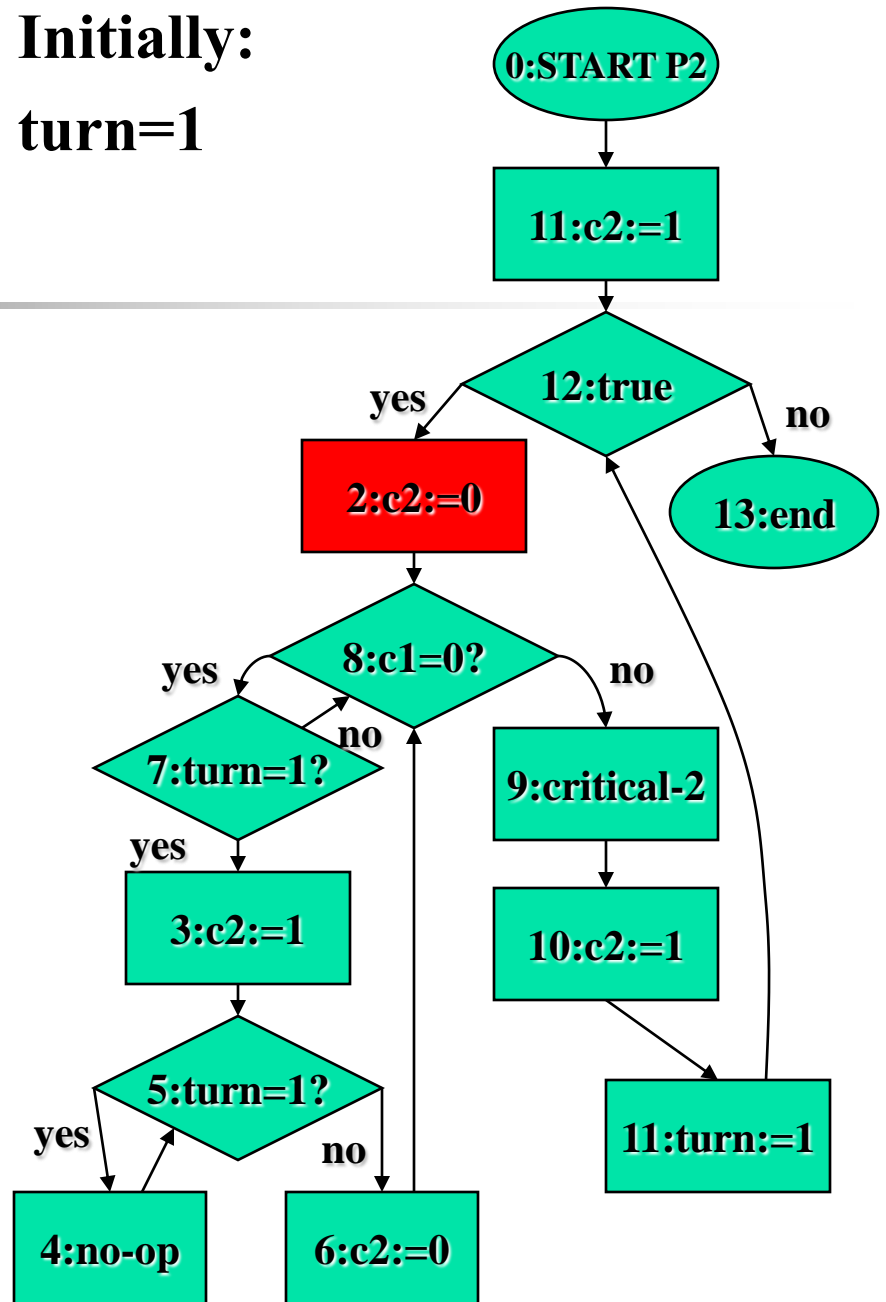
c2:=1;

turn:=1

end.



Initially:  
turn=1





# What went wrong?

---

- The execution is *unfair* to P2. It is not allowed a chance to execute.
- Such an execution is due to the interleaving model (just picking an enabled transition).
- Allowing P2 to progress, it would continue and set c2 to 0, which would allow P1 to progress.
- *Fairness* = excluding some of the executions in the interleaving model, which do not correspond to actual behavior of the system.

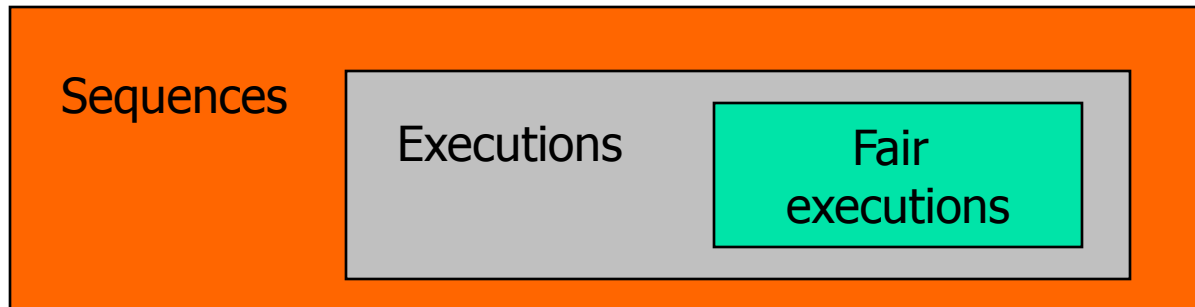
```
while c1=0 do
  begin
    if turn=1 then
      begin
        c2:=1;
        wait until turn=2;
        c2:=0;
      end
    end
  end
```

# Recall:

## The interleaving model

- An **execution** is a finite or infinite sequence of states  $s_0, s_1, s_2, \dots$
- The initial state satisfies the initial condition, I.e.,  $I(s_0)$ .
- Moving from one state  $s_i$  to  $s_{i+1}$  is by executing a transition  $e \rightarrow t$ :
  - $e(s_i)$ , I.e.,  $s_i$  satisfies  $e$ .
  - $s_{i+1}$  is obtained by applying  $t$  to  $s_i$ .

Now: consider only “fair” executions. Fairness constrains sequences that are considered to be executions.





# Some fairness definitions

---

- *Weak transition fairness:*  
It cannot happen that a transition is enabled indefinitely, but is never executed.
- *Weak process fairness:*  
It cannot happen that a process is enabled indefinitely, but none of its transitions is ever executed
- *Strong transition fairness:*  
If a transition is infinitely often enabled, it will get executed.
- *Strong process fairness:*  
If at least one transition of a process is infinitely often enabled, a transition of this process will be executed.



# Example

P1::x=1

In order for the loop to terminate (in a *deadlock*!) we need P1 to execute the assignment. But P1 may never execute, since P2 is in a loop executing *true*. Consequently,  $x==1$  never holds, and  $y$  is never assigned a 1.

Initially:  $x=0; y=0;$

P2: do

  ::  $y==0 \rightarrow$

  if

    :: *true*

    ::  $x==1 \rightarrow y=1$

  fi

od

pc1=l0  $\rightarrow$  (pc1,x):=(l1,1) /\* x=1 \*/

pc2=r0/\y=0  $\rightarrow$  pc2=r1 /\* y==0\*/

pc2=r1  $\rightarrow$  pc2=r0 /\* true \*/

pc2=r1/\x=1  $\rightarrow$  (pc2,y):=(r0,1)

/\* x==1  $\rightarrow$  y:=1 \*/

# Weak transition fairness

P1::x=1

Under **weak transition fairness**, P1 would assign 1 to x, but this does not guarantee that 1 is assigned to y and thus the P2 loop will terminate, since the transition for checking  $x==1$  is not continuously enabled (program counter not always there).

Initially:  $x=0; y=0;$

```
P2: do
  ::  $y==0 \rightarrow$ 
  if
  :: true
  ::  $x==1 \rightarrow y=1$ 
  fi
od
```

**Weak process fairness** only guarantees P1 to execute, but P2 can still choose the *true* guard.

**Strong process fairness:** same.



# Strong transition fairness

Initially:  $x=0; y=0;$

P1:: $x=1$

Under **strong transition fairness**, P1 would assign 1 to  $x$ . If the execution was infinite, the transition checking  $x==1$  was infinitely often enabled. Hence it would be eventually selected. Then assigning  $y=1$ , the main loop is not enabled anymore.

P2: do

  ::  $y==0 \rightarrow$

  if

  :: true

  ::  $x==1 \rightarrow y=1$

  fi

od



# Specifying fairness conditions

---

- Express properties over an alternating sequence of states and transitions:

$S_0 \alpha_1 S_1 \alpha_1 S_2 \dots$

- Use transition predicates  $\text{exec}_\alpha$ .



# Some fairness definitions

---

$exec_{\alpha}$   $\alpha$  is executed.

$exec_{P_i}$  some transition of  $P_i$  is executed.

$en_{\alpha}$   $\alpha$  is enabled.

$en_{P_i}$  some transition of process  $P_i$  is enabled.

$$en_{P_i} = \bigvee_{\alpha \in P_i} en_{\alpha}$$

$$exec_{P_i} = \bigvee_{\alpha \in P_i} exec_{\alpha}$$

- *Weak transition fairness:*

$$\bigwedge_{\alpha \in T} (\langle \rangle [] en_{\alpha} \rightarrow [] \langle \rangle exec_{\alpha}).$$

Equivalently:  $\bigwedge_{\alpha \in T} \neg \langle \rangle [] (en_{\alpha} \wedge \neg exec_{\alpha})$

- *Weak process fairness:*

$$\bigwedge_{P_i} (\langle \rangle [] en_{P_i} \rightarrow [] \langle \rangle exec_{P_i})$$

- *Strong transition fairness:*

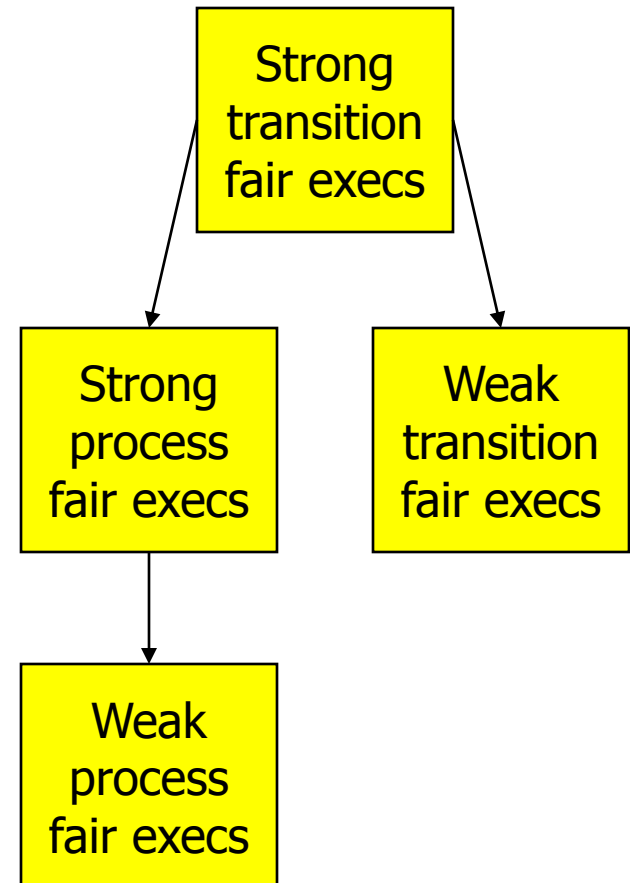
$$\bigwedge_{\alpha \in T} ( [] \langle \rangle en_{\alpha} \rightarrow [] \langle \rangle exec_{\alpha} )$$

- *Strong process fairness:*

$$\bigwedge_{P_i} ( [] \langle \rangle en_{P_i} \rightarrow [] \langle \rangle exec_{P_i} )$$

# “Weaker fairness condition”

- $A$  is **weaker** than  $B$  if  $B \rightarrow A$ .  
(Means  $A$  has more executions than  $B$ .)
- Consider the executions  $L(A)$  and  $L(B)$ . Then  $L(B) \subseteq L(A)$ .
- If an execution is strong {process/transition} fair, then it is also weak {process/transition} fair.
- There are fewer strong {process, transition} fair executions.



# Fairness is an abstraction; no scheduler can guarantee exactly all fair executions!

Initially:  $x=0, y=0$

P1:: $x=1$

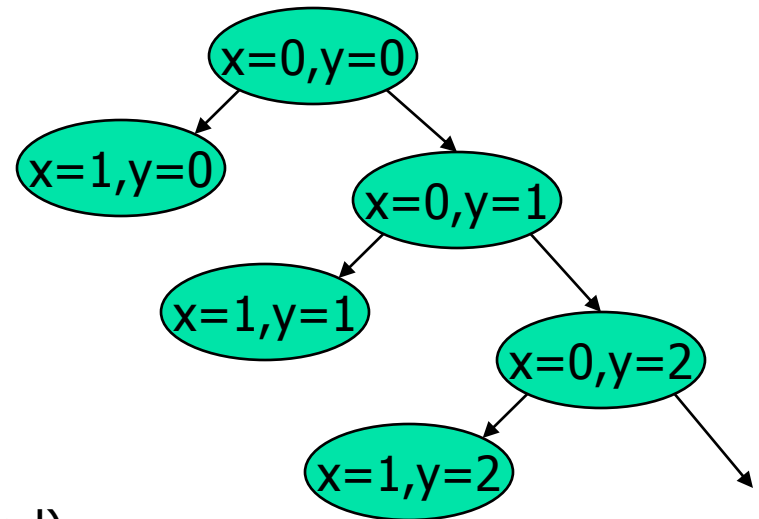
||

P2::do

  ::  $x==0 \rightarrow y=y+1$

  ::  $x==1 \rightarrow \text{break}$

od



Under fairness assumption (any of the four defined), P1 will execute the assignment, and consequently, P2 will terminate. All executions are finite and there are infinitely many of them, and infinitely many states.

Thus, an execution tree (the state space) will potentially look like the one on the right, but with infinitely many states, finite branching and only finite sequences. But according to **König's Lemma** there is no such tree!



# Model Checking under fairness

---

- Instead of verifying that the program satisfies  $\varphi$ , verify it satisfies *fair*  $\rightarrow \varphi$
- Problem: may be inefficient. Also fairness formula may involves special arrangement for specifying what exec means.
- May specialize model checking algorithm instead.



# Model Checking under Fairness

---

Specialize model checking. For weak *process fairness*: search for a reachable strongly connected component, where for each process  $P$  either

- it contains an occurrence of a transition from  $P$ , or
- it contains a state where  $P$  is disabled.
- Weak transition fairness: similar.
- Strong fairness: more difficult algorithm (graph transformation).



# Abstractions

---

(Book: Chapter 10.1)



# How to fight the complexity problem?



---

- Abstraction
- Compositionality
- Partial Order Reduction
- Symmetry
- Other model checking strategies:  
Symbolic (BDD), Bounded Model  
Checking (using SAT solving).



# Abstraction

---

- Represent the program using a smaller model.
- Pay attention to preserving the checked properties.
- Do not affect the flow of control.



# Main idea

---

- Use smaller data objects.

$x := f(m)$

$y := g(n)$

if  $x * y > 0$  then ...

else ...

$x, y$  never used again.



# How to abstract?

---

- Assign values  $\{-1, 0, 1\}$  to  $x$  and  $y$ .
- Based on the following connection:  
$$\text{sgn}(x) = \begin{cases} 1 & \text{if } x > 0, \\ 0 & \text{if } x = 0, \text{ and} \\ -1 & \text{if } x < 0. \end{cases}$$
$$\text{sgn}(x) * \text{sgn}(y) = \text{sgn}(x * y).$$



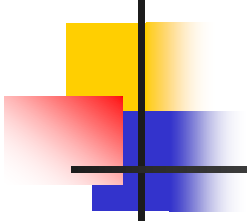
# Abstraction mapping

---

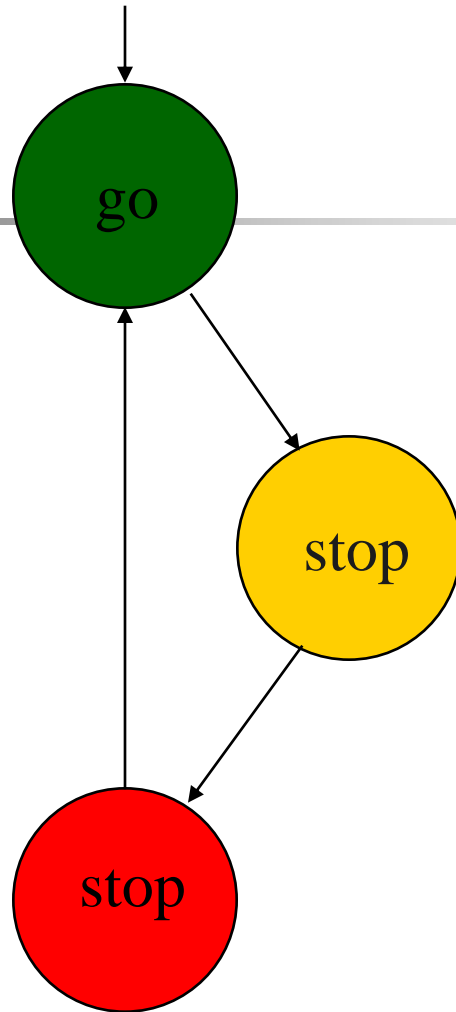
- $S$  - states,  $I$  - initial states.  $L(s)$  - labeling.
- $R(S,S)$  - transition relation.
- $h(s)$  maps  $s$  into its abstract image.

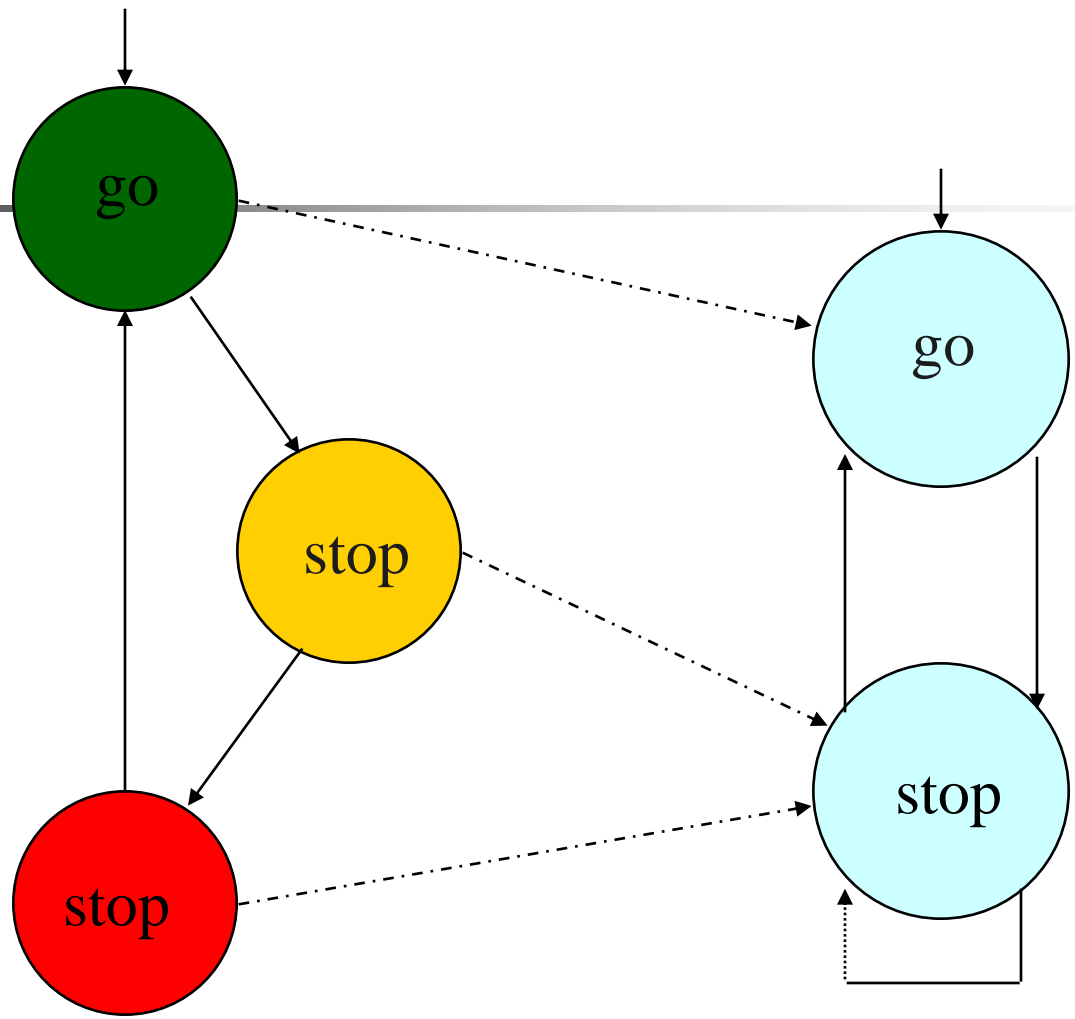
Full model	$-h \rightarrow$	Abstract model
$I(s)$	$-h \rightarrow$	$I(h(s))$
$R(s, t)$	$-h \rightarrow$	$R(h(s), h(t))$

$\text{Label}(h(s)) = \text{Label}(s)$



Traffic light  
example



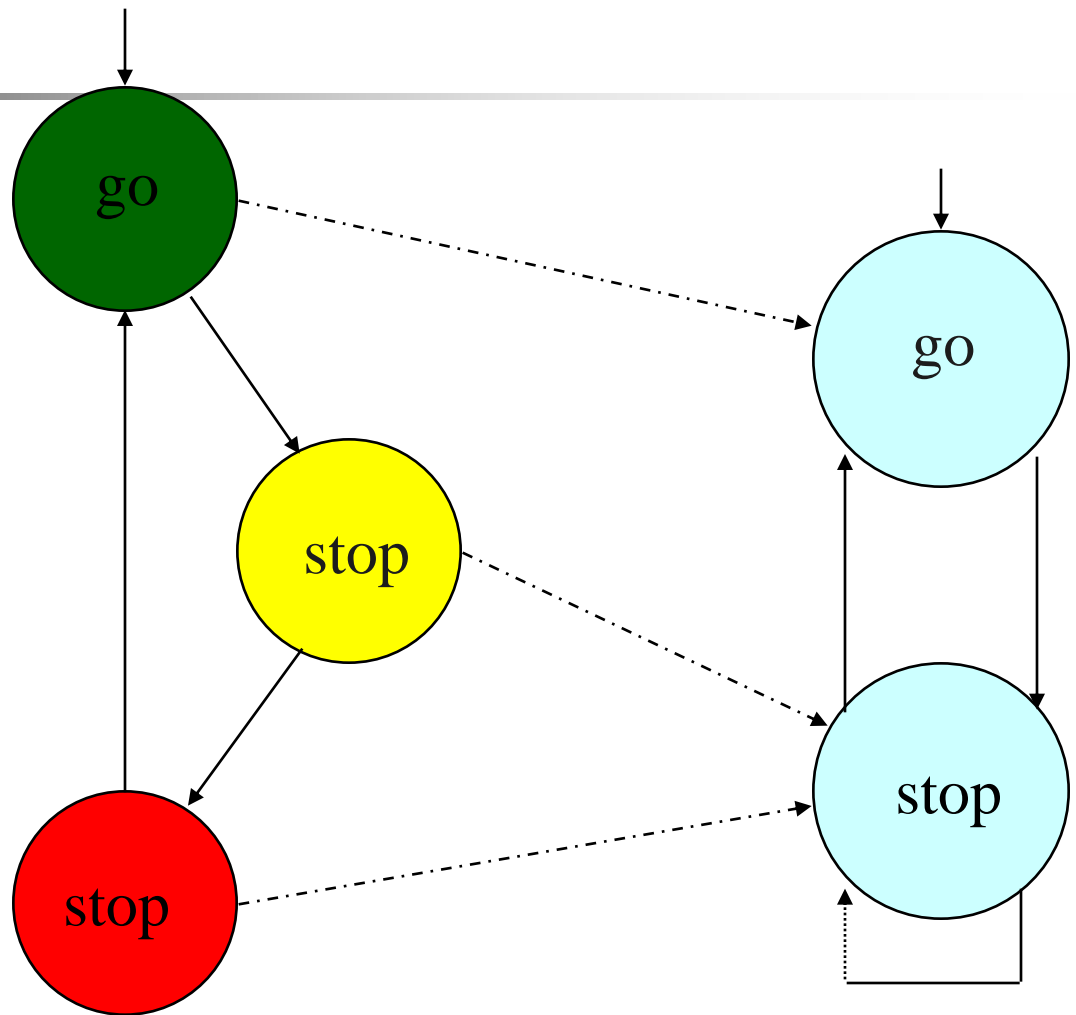


# What do we preserve?

Every execution of the full model can be simulated by an execution of the reduced one.

Every LTL property that holds in the reduced model hold in the full one.

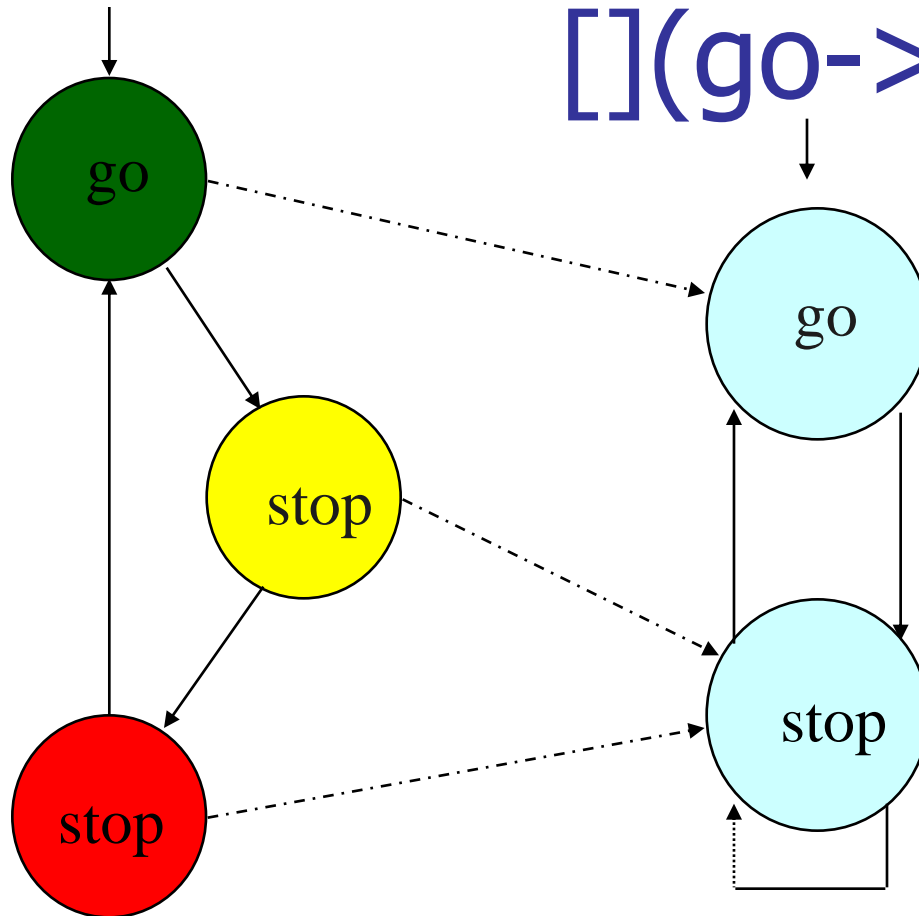
But there can be properties holding for the original model but not the abstract one.





Preserved:

$\square(\text{go} \rightarrow 0 \text{ stop})$

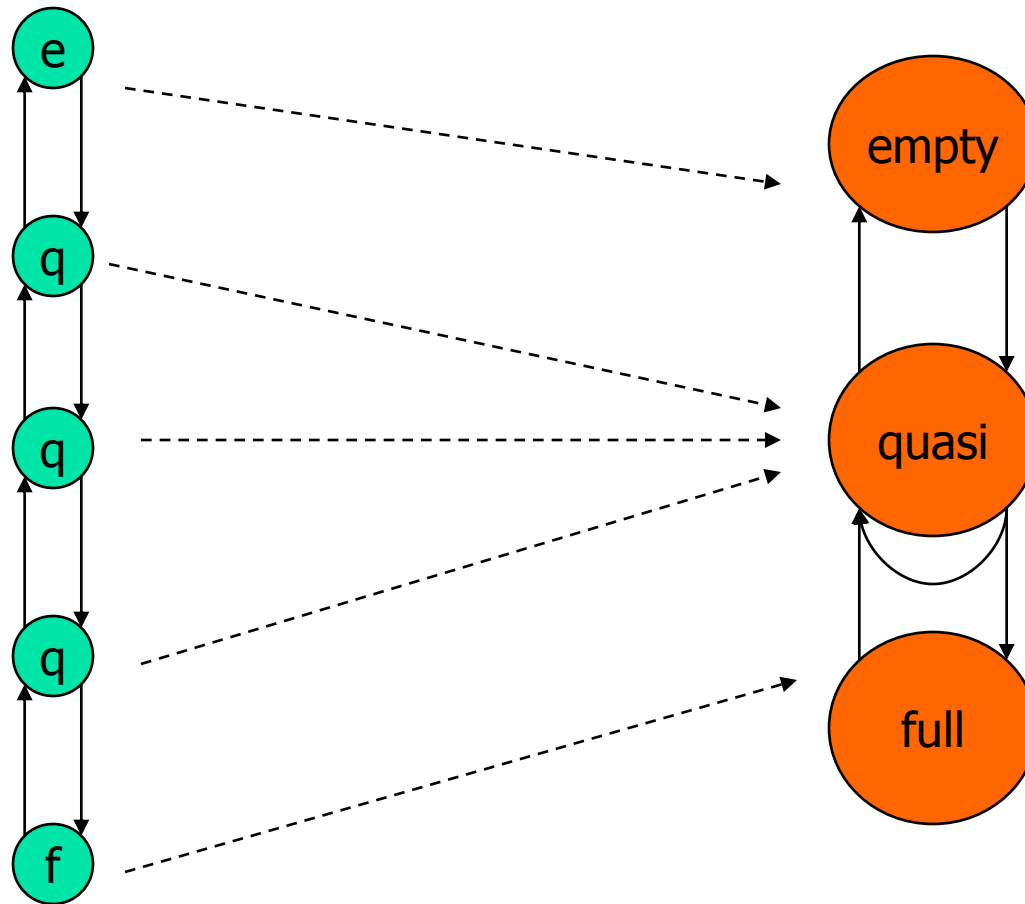


Not preserved:

$\square \langle \rangle \text{go}$

Counterexamples  
need to be  
checked.

Homework: what is preserved in the following buffer abstraction? What is not preserved?





# CTL, BDD representation Symbolic model checking

---

Why CTL?

More efficient model checking algorithm (Polynomial).  
Can express branching points (alternatives) in executions.

Why not CTL?

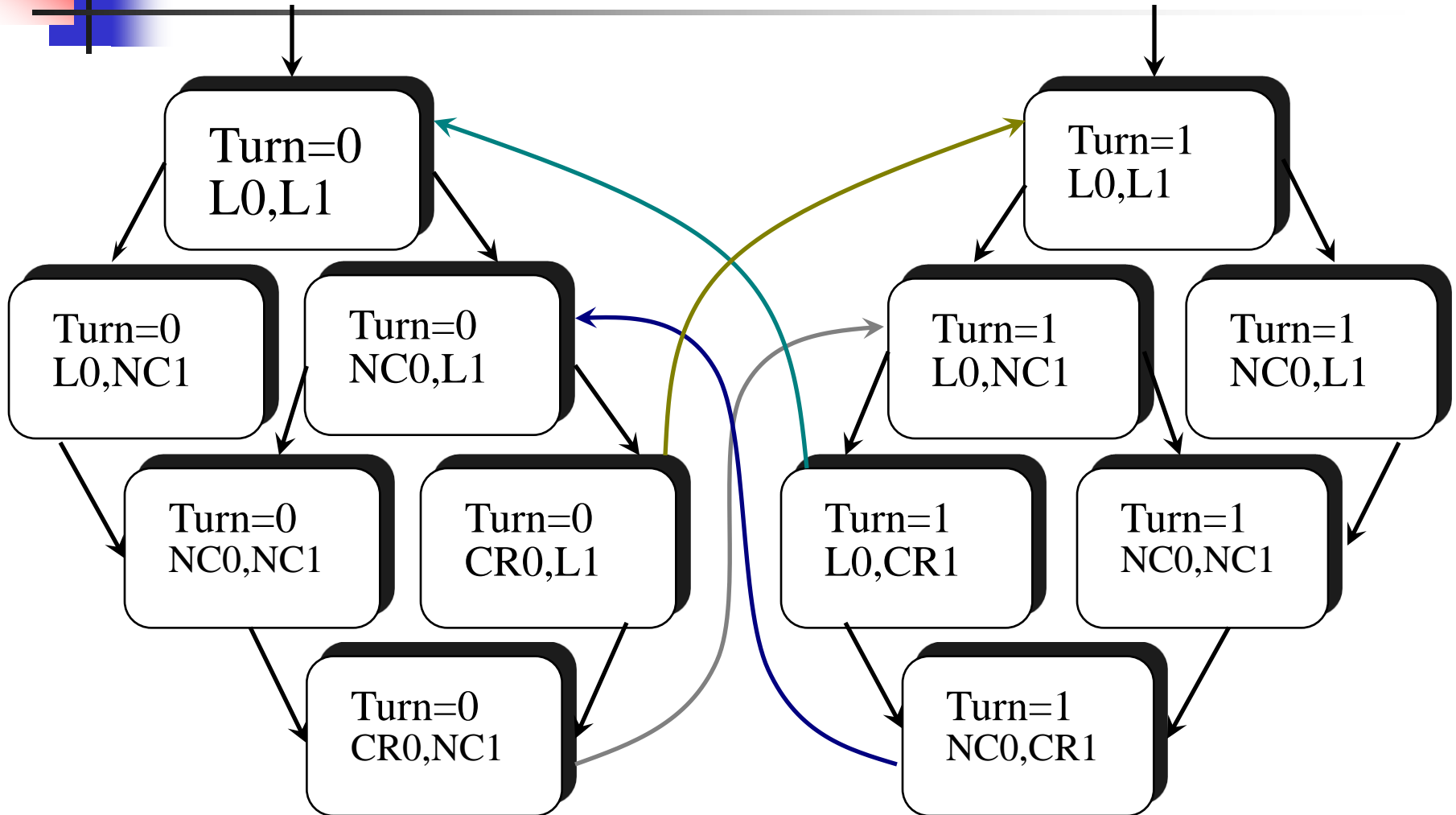
Cannot express fairness (use CTL\*, loose complexity advantage).  
Hard to give counterexamples.

The debate is still ongoing...

Recall our state graph: reachable states, labeled with the properties that hold in each state.

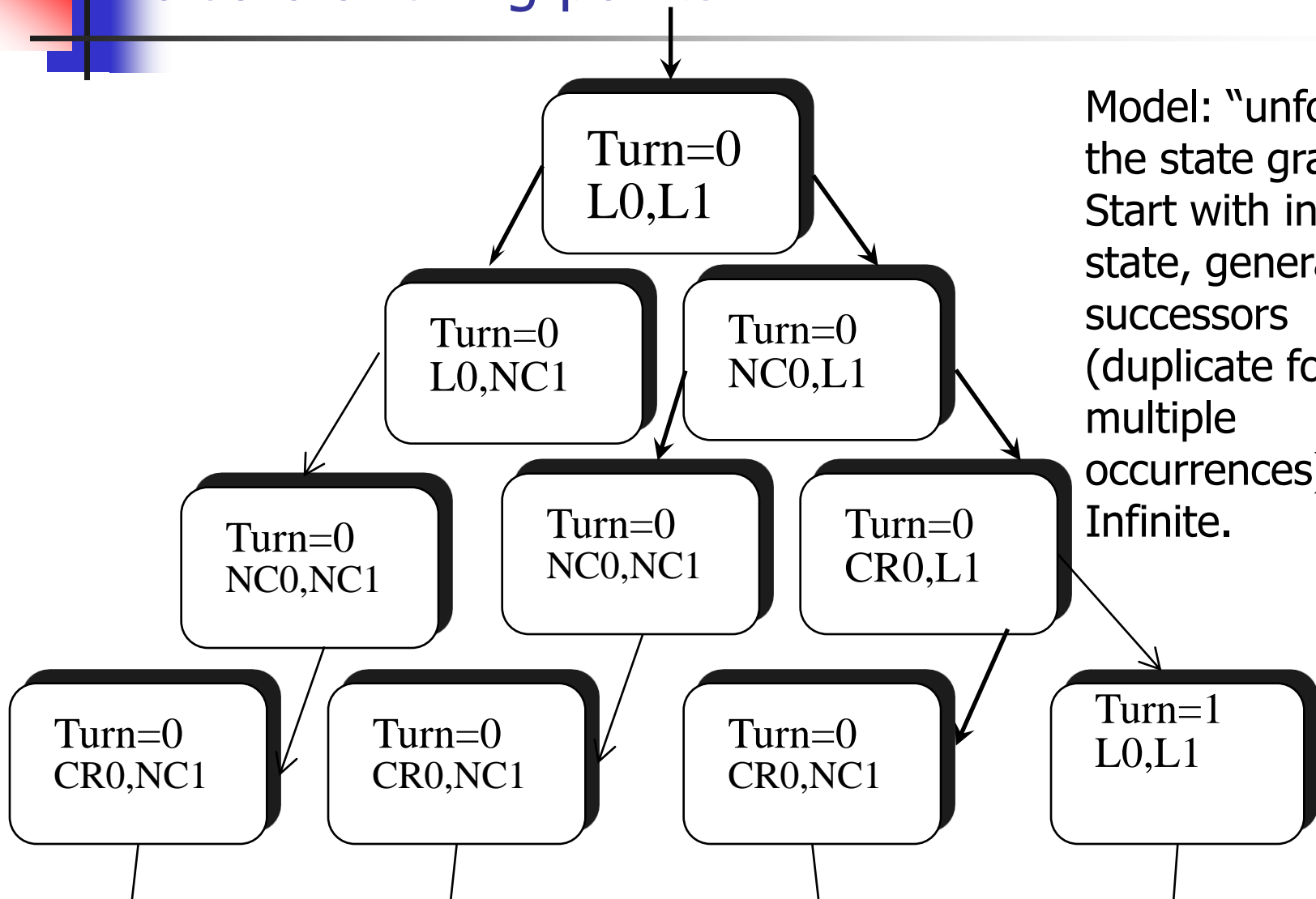
Linear view: look at all executions.

Branching View: Look at a tree that embeds also branching points



Linear view: look at all executions.

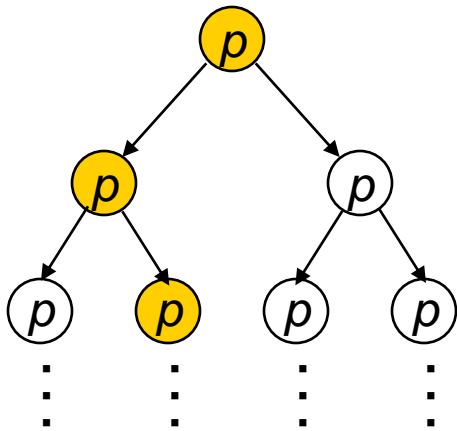
Branching View: Look at a tree that embeds also branching points



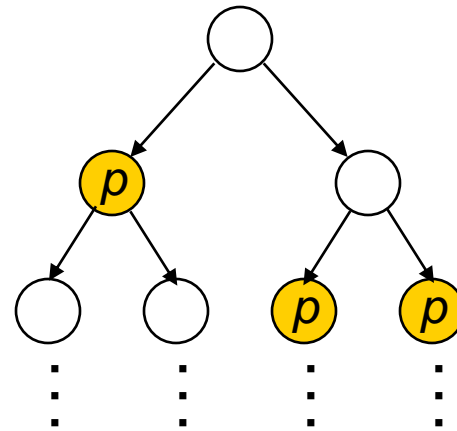
Model: "unfold" the state graph. Start with initial state, generate successors (duplicate for multiple occurrences). Infinite.

# Computation Tree Logic

$EG p$



$AF p$

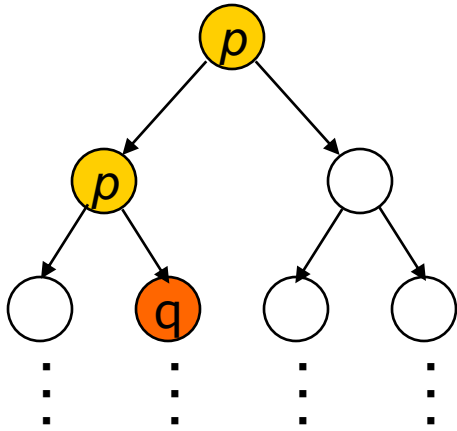


State quantifiers: A (forall), E (exists)

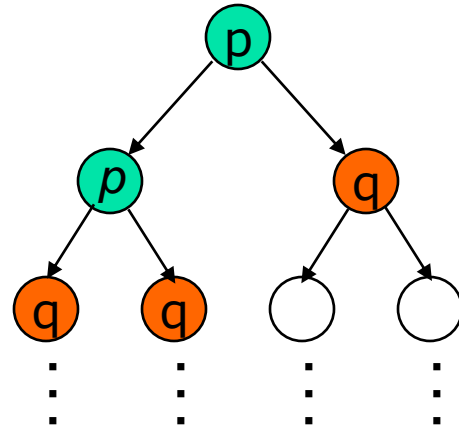
Path quantifiers: X (=0), G (=[]), F (=⟨>), U.

# Computation Tree Logic

$E p U q$



$A p U q$



# Example formulas

---

## CTL formulas:

- **mutual exclusion:**  $\mathbf{AG} \neg( CS_1 \wedge CS_2 )$
- **non starvation:**  $\mathbf{AG} (\text{request} \Rightarrow \mathbf{AF} \text{grant})$
- **The possibility of returning to recoverable state not blocked:**  
 $\mathbf{AG} \mathbf{EF} \text{rec}$





# Model Checking $M \models f$

---

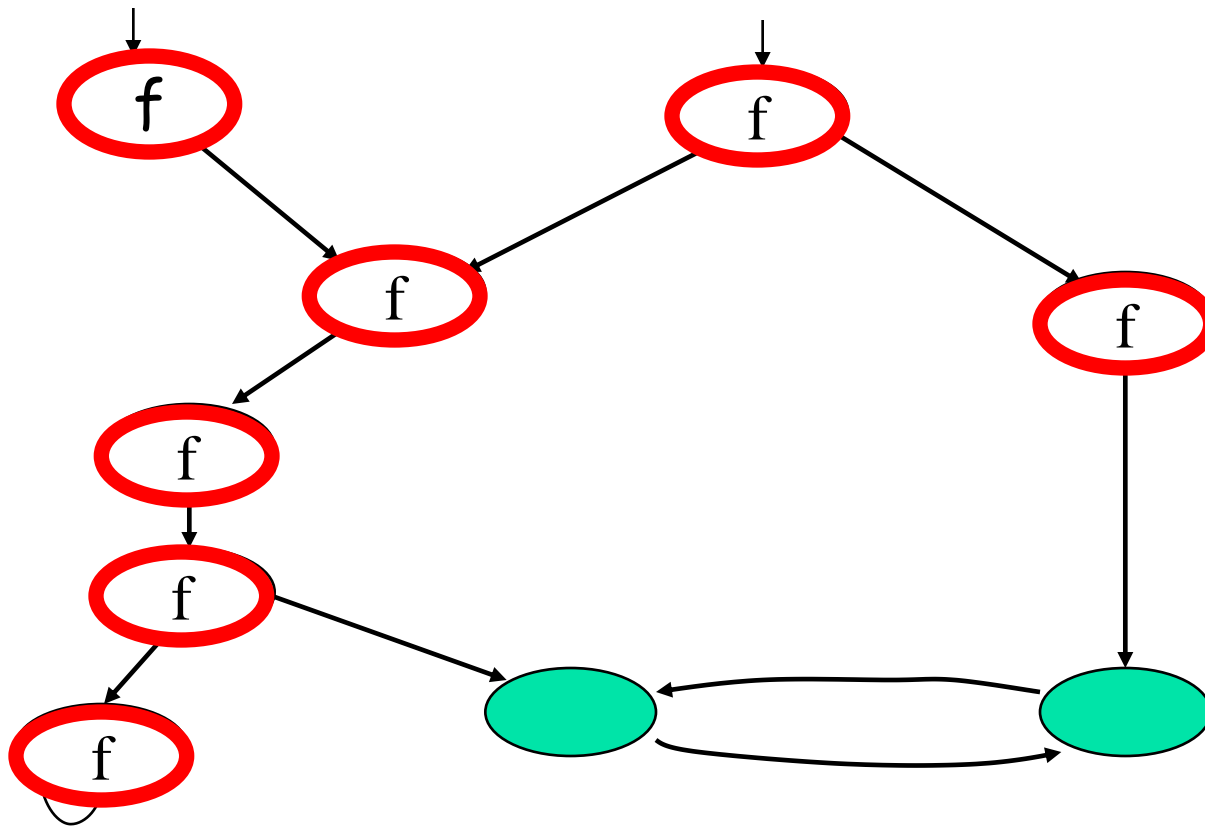
- The **Model Checking** algorithm works **iteratively on structure of formula.**  
on subformulas of  $f$ , from **simpler** subformulas to more **complex** ones
- When checking subformula  $g$  of  $f$  we assume that all subformulas of  $g$  have already been checked
- For subformula  $g$ , the algorithm returns the **set of states** that satisfy  $g$  ( $S_g$ )
- The algorithm has time complexity:  **$O(|M| \times |f|)$**  where  $|M|$  is the size of the global state space: the (exponentially big) product of the processes.

# Model checking $f = EF g$

Given a model  $M = \langle S, I, R, L \rangle$   
and  $S_g$  the sets of states satisfying  $g$  in  $M$

```
procedure CheckEF ( $S_g$ )  
   $Q := \text{emptyset}; Q' := S_g;$   
  while  $Q \neq Q'$  do  
     $Q := Q';$   
     $Q' := Q \cup \{ s \mid \exists s' [ R(s,s') \wedge Q(s') ] \}$   
  end while  
   $S_f := Q;$  return( $S_f$ )
```

Example:  $f = EF g$



# Model checking $f = EG\ g$

**CheckEG** gets  $M = \langle S, I, R, L \rangle$  and  $S_g$   
and returns  $S_f$

procedure **CheckEG** ( $S_g$ )

$Q := S$  ;  $Q' := S_g$  ;

while  $Q \neq Q'$  do

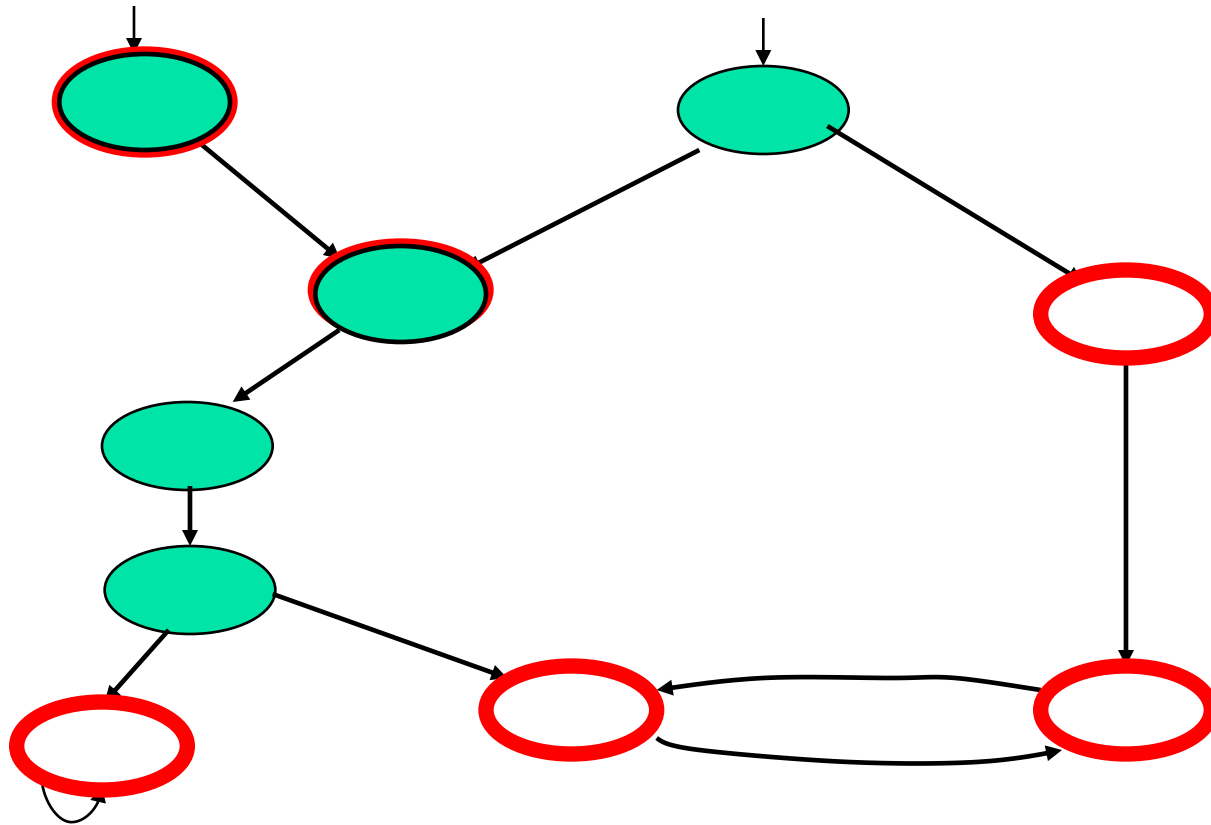
$Q := Q'$  ;

$Q' := Q \cap \{ s \mid \exists s' [ R(s, s') \wedge Q(s') ] \}$

end while

$S_f := Q$  ; return( $S_f$ )

**Example:  $f = EG g$**



# Binary decision diagrams

(BDDs) [Bryant 86]

---

- Data structure for representing Boolean functions.
- Can represent **sets of states** and perform operations on them.
- **Boolean operations** on BDDs can be done in **polynomial time** in the BDD size.



# BDDs in model checking

---

- Assume that **states** in model  $M$  are **encoded by  $\{0,1\}^n$**  and described by Boolean variables  $\mathbf{v}_1 \dots \mathbf{v}_n$
- A set of states can be represented by a BDD over  $\mathbf{v}_1 \dots \mathbf{v}_n$
- $\mathbf{R}$  (a set of pairs of states  **$(s, s')$** ) can be represented by a BDD over  $\mathbf{v}_1 \dots \mathbf{v}_n \quad \mathbf{v}_1' \dots \mathbf{v}_n'$



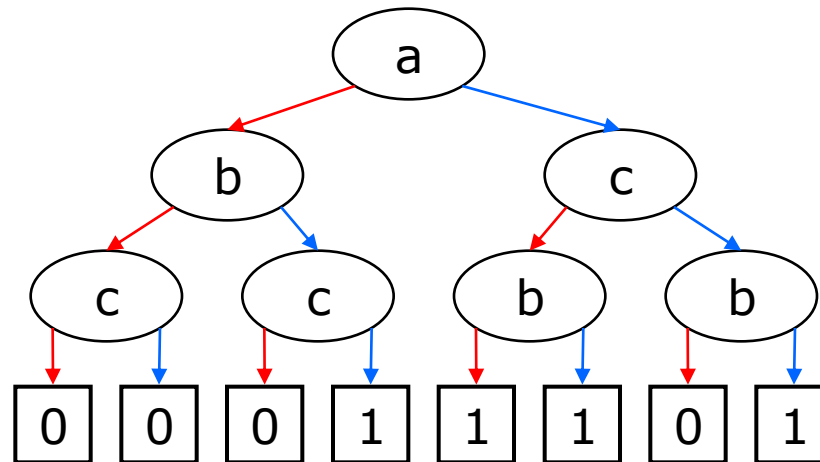
# BDD definition

---

- A tree representation of a Boolean formula.
- Each leaf represents 0 (false) or 1 (true).
- Each internal leaf represents a node.
- If we follow a path in the tree and go from a node **left (low)** on 0 and **right (high)** on 1, we obtain a leaf that corresponds to the value of the formula under this truth assignment.

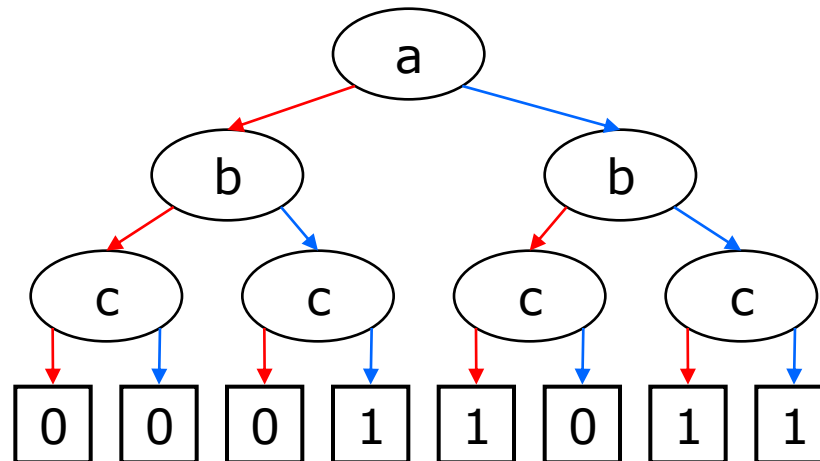


# Example



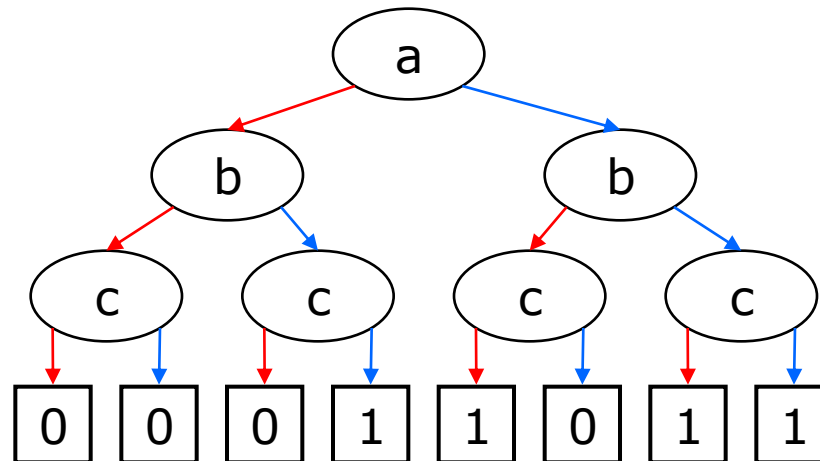
$$(a \wedge (b \vee \neg c)) \vee (\neg a \wedge (b \wedge c))$$

OBDD: there is some fixed appearance order between variables, e.g.,  $a < b < c$



$$(a \wedge (b \vee \neg c)) \vee (\neg a \wedge (b \wedge c))$$

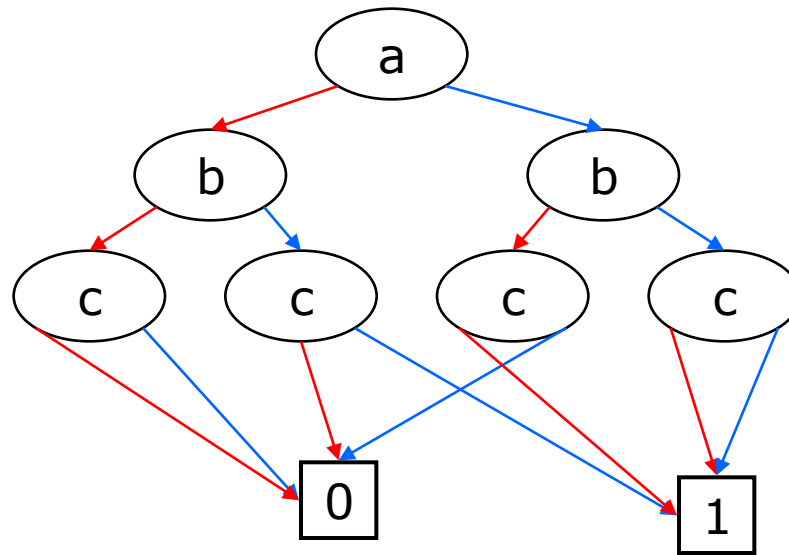
In reduced form: combine all leafs with same values, all isomorphic subgraph.



$$(a \wedge (b \vee \neg c)) \vee (\neg a \wedge (b \wedge c))$$

In addition, remove nodes with identical children ( $\text{low}(x) = \text{high}(x)$ ).

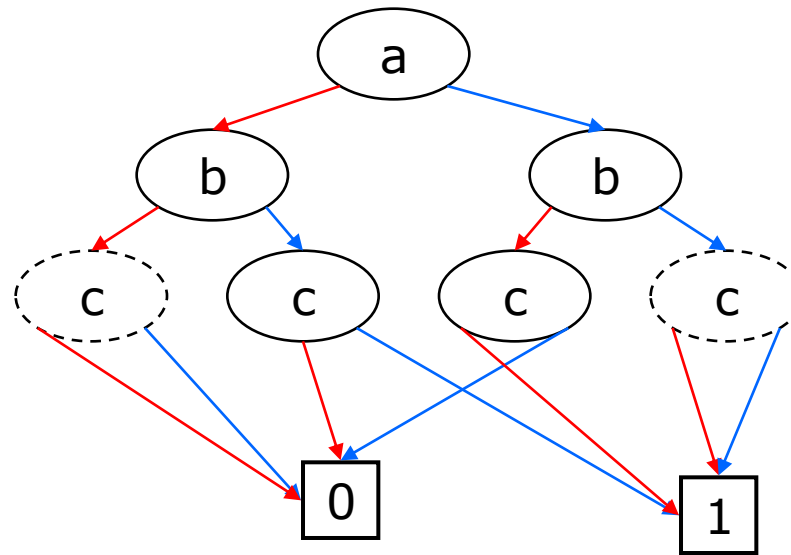
In reduced form: combine all leafs with same values, all isomorphic subgraph.



$$(a \wedge (b \vee \neg c)) \vee (\neg a \wedge (b \wedge c))$$

In addition, remove (shortcut) nodes with identical children ( $\text{low}(x) = \text{high}(x)$ ). Apply bottom up until not possible.

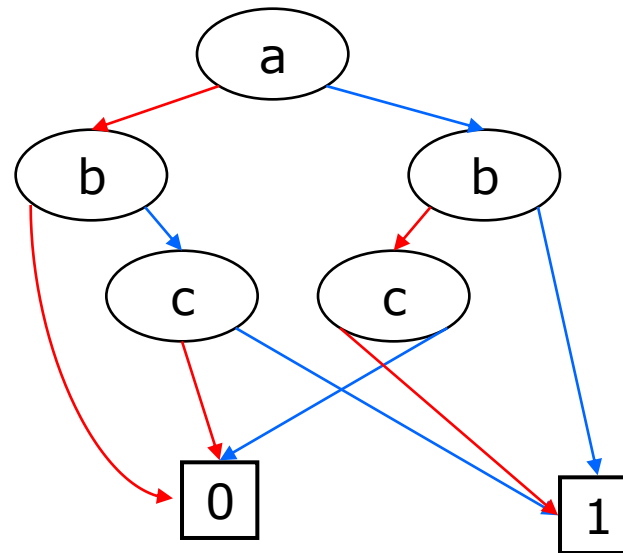
In reduced form: combine all leafs with same values, all isomorphic subgraph.



$$(a \wedge (b \vee \neg c)) \vee (\neg a \wedge (b \wedge c))$$

In addition, **remove (shortcut) nodes with identical children** ( $\text{low}(x) = \text{high}(x)$ ).

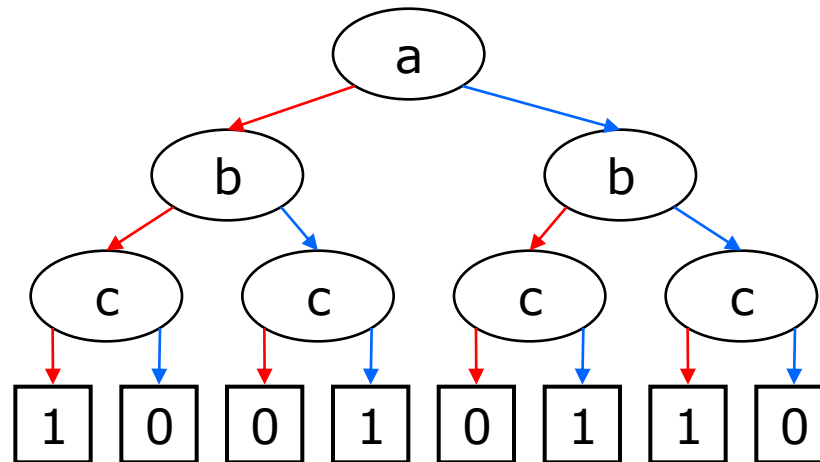
In reduced form: combine all leafs with same values, all isomorphic subgraph.



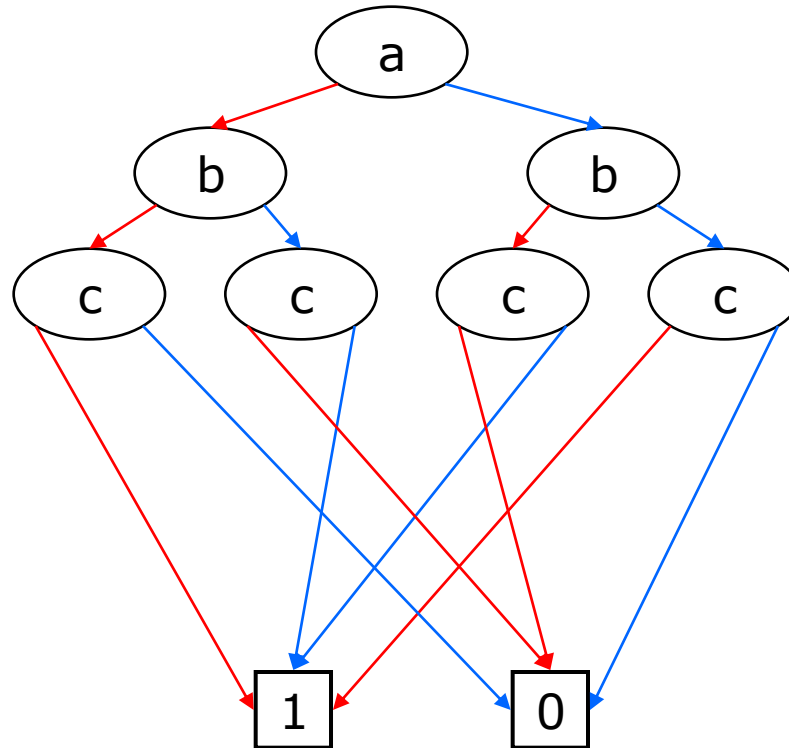
$$(a \wedge (b \vee \neg c)) \vee (\neg a \wedge (b \wedge c))$$

In addition, **remove (shortcut) nodes with identical children** ( $\text{low}(x) = \text{high}(x)$ ).

# Example, even parity, 3 bits

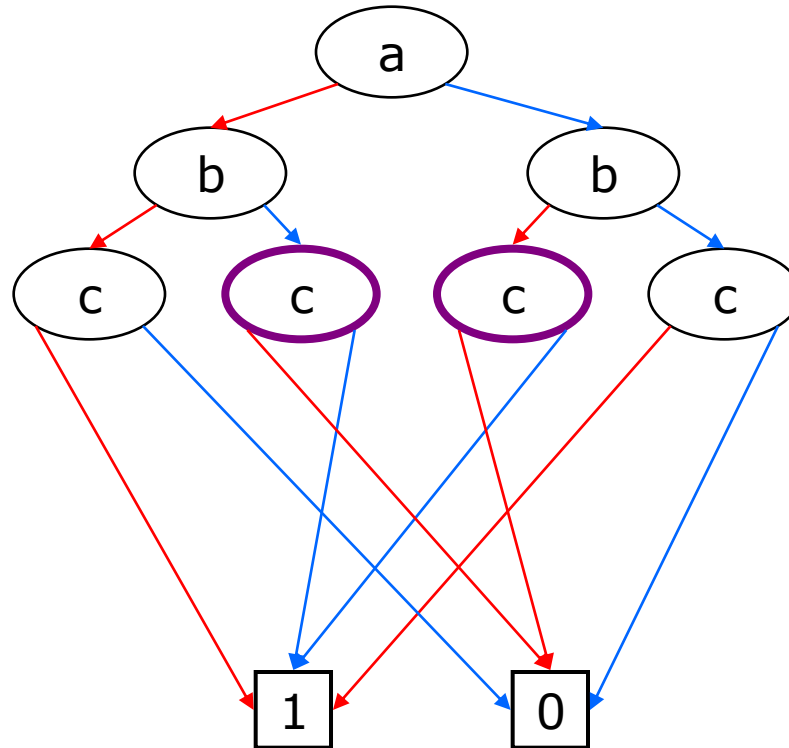


# Apply reduce

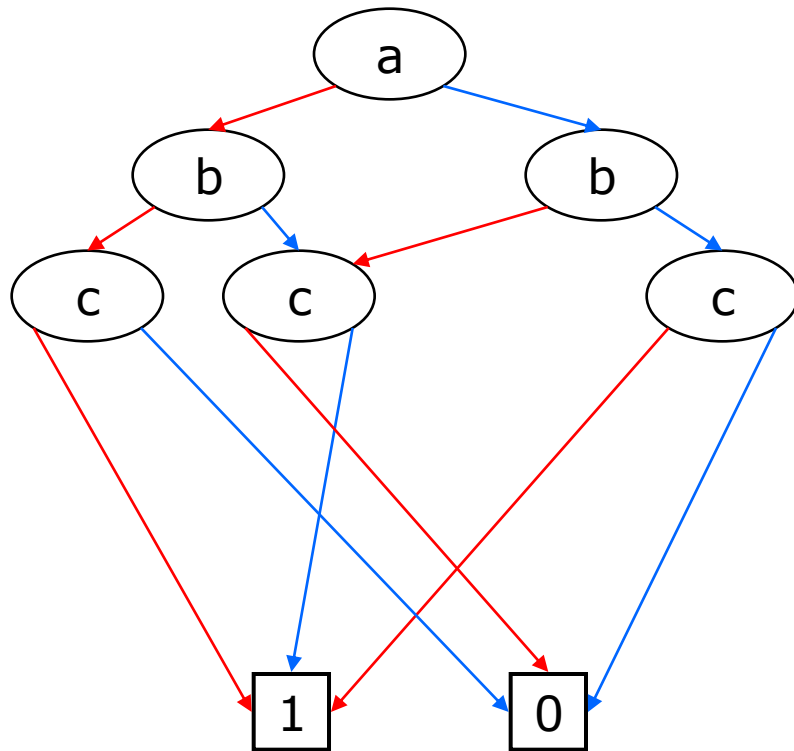




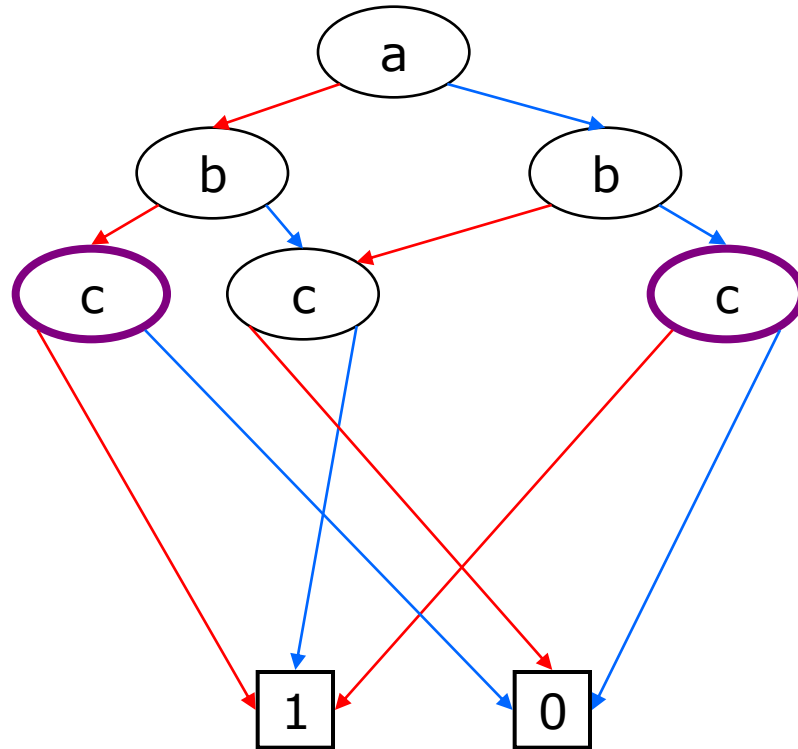
# Apply reduce



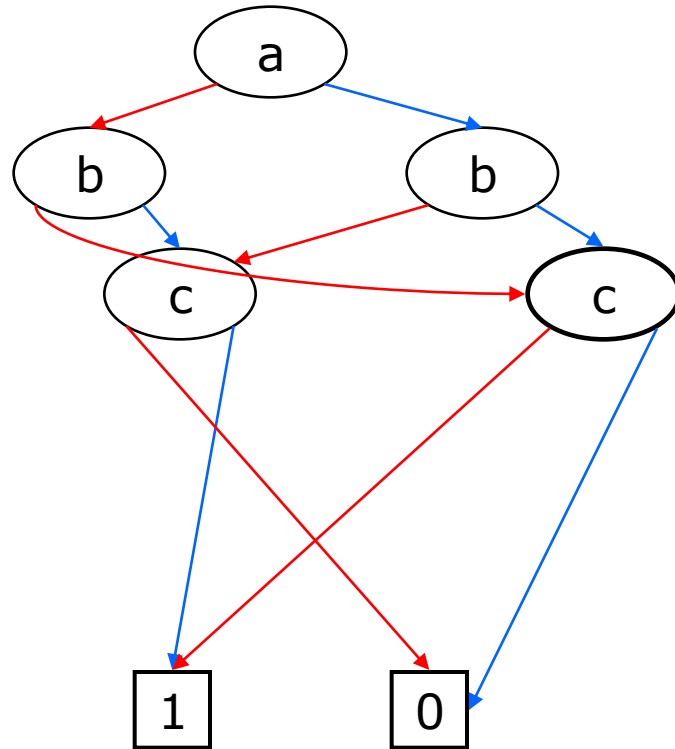
# Apply reduce



# Apply reduce



# Apply reduce





## $f[0/x]$ , $f[1/x]$ (“restrict” algorithm)

---

- Goal: Obtain the replacement of a variable  $x$  by 0 or 1, in formula  $f$ , respectively.
- For  $f[0/x]$ , incoming edges to node  $x$  are redirected to **low**( $x$ ), and  $x$  is removed.
- For  $f[1/x]$ , incoming edges to node  $x$  are redirected to **high**( $x$ ), and  $x$  is removed.
- Then we reduce the OBBD.



# Calculate $\exists x\varphi$

---

- $\exists x\varphi = \varphi[0/x] \vee \varphi[1/x]$
- Thus, we apply “restrict” twice to  $\varphi$  and then “apply” the disjunction.



# Shannon expansion of Boolean expression f.

---

- $f = (\neg x \wedge f[0/x]) \vee (x \wedge f[1/x])$
- Thus,  $f \# g$ , for some logical operator # is  $f \# g = (\neg x \wedge f \# g [0/x]) \vee (x \wedge f \# g [1/x]) = (\neg x \wedge f [0/x] \# g [0/x]) \vee (x \wedge f [1/x] \# g [1/x])$



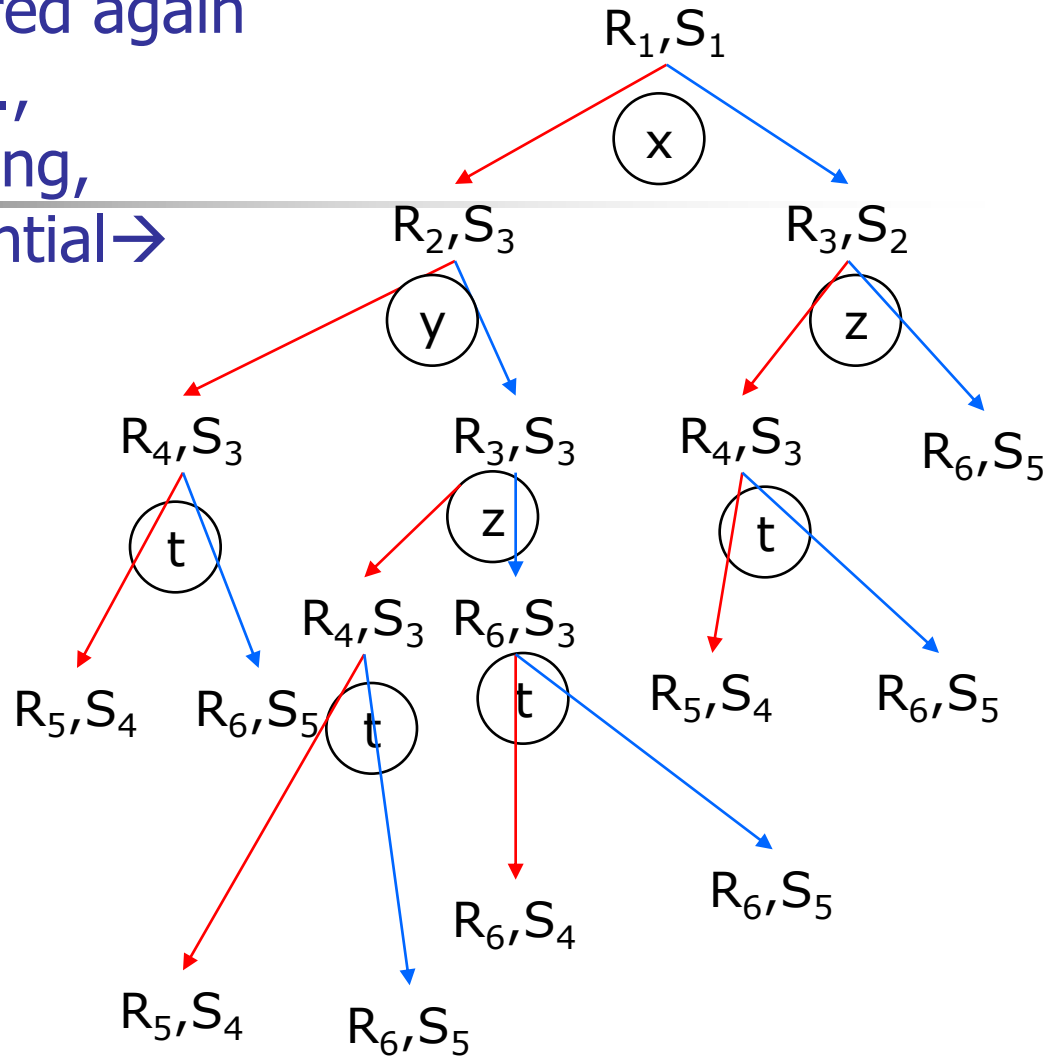
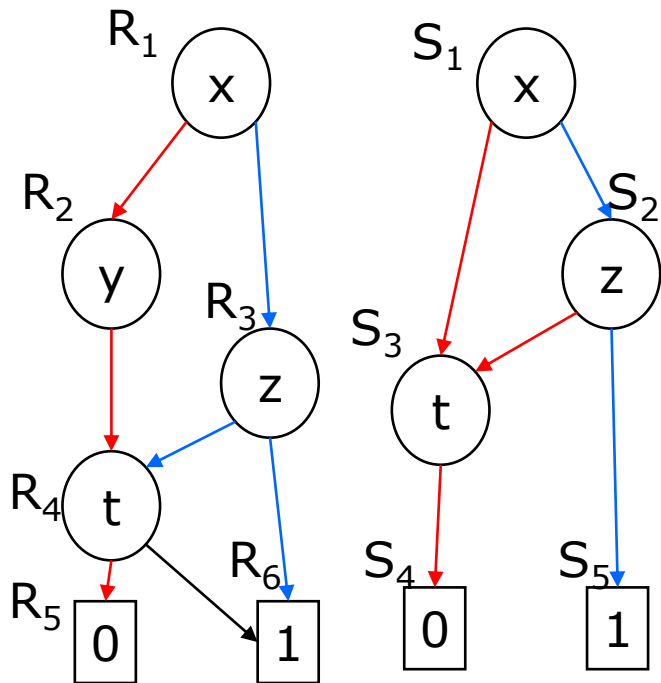
# Now compute $f \# g$ recursively:

---

- Let  $r_f$  be the root of the OBDD for  $f$ , and  $r_g$  be the root of the OBDD for  $g$ .
- If  $r_f$  and  $r_g$  are terminals, then apply  $r_f \# r_g$ .
- If both roots are same node (say  $x$ ), then create a **low** edge to  $\text{low}(r_f) \# \text{low}(r_g)$ , and a high edge to  $\text{high}(r_f) \# \text{high}(r_g)$ .
- If  $r_f$  is  $x$  and  $r_g$  is  $y$ , and  $x < y$ , there is no  $x$  node in  $g$ , so  $g = g[0/x] = g[1/x]$ . So we create a **low** edge to  $\text{low}(r_f) \# g$  and a **high** edge to  $\text{high}(r_f) \# g$ . The symmetric case is handled similarly.
- Reduce.

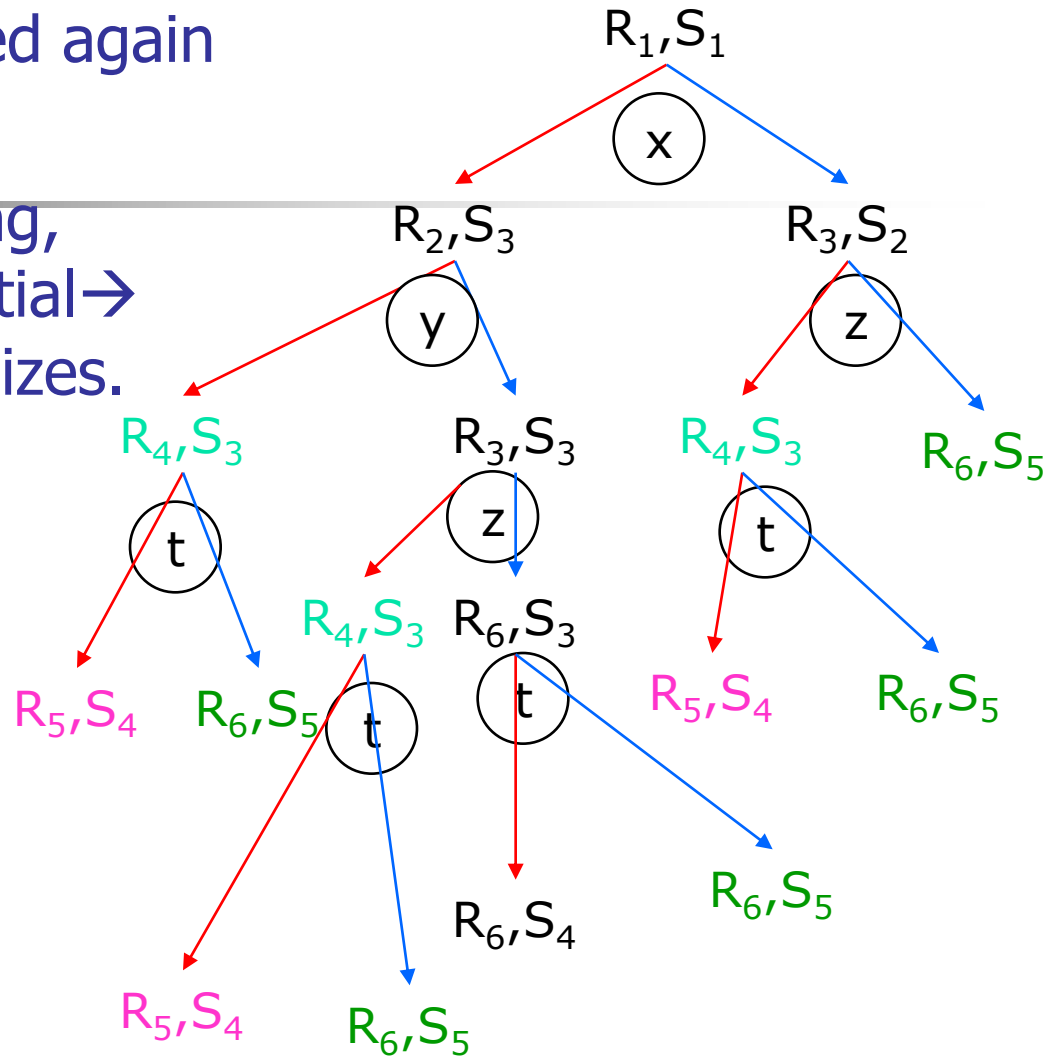
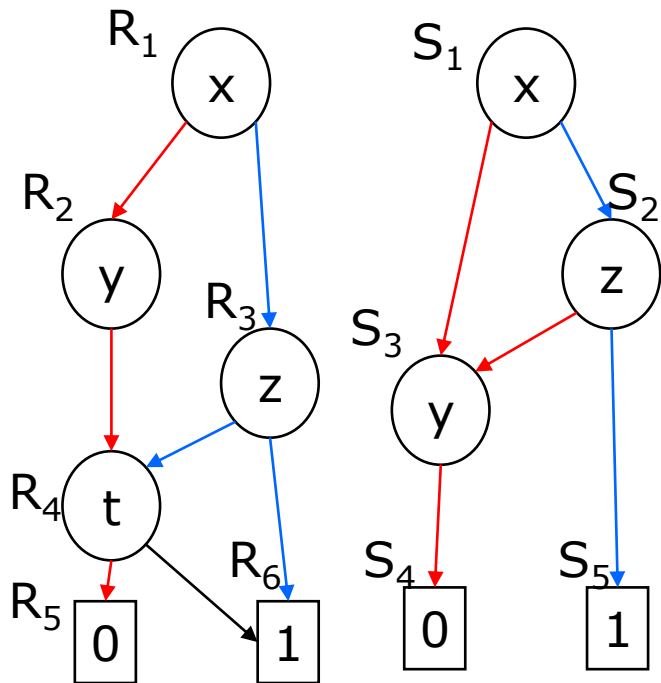


Same subgraphs are not needed to be explored again (use memoising, i.e., dynamic programming, complexity: exponential  $\rightarrow$  2 x mult of sizes.



Same subgraphs are not needed to be explored again

(use memoising, i.e., dynamic programming, complexity: exponential  $\rightarrow$  2x multiplications of sizes.)

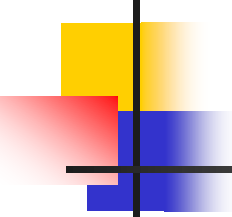




# Symbolic Model Checking

---

- Characterize CTL formulas using fixpoints.
- $AF\phi = \phi \vee AX AF\phi$
- $EF\phi = \phi \vee \mathbf{EX EF\phi} \rightarrow \mu \mathbf{Z}.\phi \vee \mathbf{EX Z}$
- $AG\phi = \phi \wedge AX AG\phi$
- $EG\phi = \phi \wedge \mathbf{EX EG\phi} \rightarrow \nu \mathbf{Z}.\phi \wedge \mathbf{EX Z}$
- $A\phi U\psi = \psi \vee (\phi \wedge AX A\phi U\psi)$
- $E\phi U\psi = \psi \vee (\phi \wedge \mathbf{EX E\phi U\psi}) \rightarrow \mu \mathbf{Z}.\psi \vee (\phi \wedge \mathbf{EX Z})$
- $A\phi R\psi = \psi \wedge (\phi \vee AX A\phi R\psi)$
- $E\phi R\psi = \psi \wedge (\phi \vee \mathbf{EX E\phi R\psi}) \rightarrow \nu \mathbf{Z}.\psi \wedge (\phi \vee \mathbf{EX Z})$



# Representing the successor relation formula **R**

---

- A relation between the current state and the next state can be represented as a BDD with prime variables representing the variables at next states.
- For example:  
 $p \wedge \neg q \wedge r \wedge \neg p' \wedge q' \wedge r'$  says that the current state satisfies  $p \wedge \neg q \wedge r$  and the next state satisfies  $\neg p \wedge q \wedge r$ . (typically, for one transition, represented as a Boolean relation).
- If  $t_i$  represents this relation for transition  $i$ , we can write for the entire code  $R = \bigvee_i t_i$ .

# Calculating $\tau(Z)$ for

$$\tau(\mathbf{Z}) = \varphi \vee \mathbf{E} \mathbf{X} \mathbf{Z}$$

---

- $Z$  is a BDD.
- Rename variables in  $Z$  by their primed version to obtain BDD  $Z'$ .
- Calculate the BDD  $R/\backslash Z'$ .
- Let  $y_1' \dots y_n'$  be the primed variables, Then calculate the BDD  $X = \exists y_1' \dots \exists y_n' R/\backslash Z'$  to remove primed variables.
- Calculate the BDD  $\varphi \vee X$ .

Model checking  $\mu Z \tau$  (least fixed point)

For example,  $\tau(Z) = \varphi \vee EX Z$

For formulas with main operator  $\vee$ .

---

procedure **Check LFP** ( $\tau$ )

$Q := \text{False}; Q' := \tau(Q);$

while  $Q \neq Q'$  do

$Q := Q';$

$Q' := \tau(Q);$

end while

return( $Q$ )

Model checking  $\mu\mathbf{Z} \tau$  (**Greatest fixed point**)

For example,  $\tau(\mathbf{Z}) = \psi \wedge (\varphi \vee \mathbf{EX} \mathbf{Z})$

For formulas with main operator  $\wedge$ .

---

procedure **Check GFP** ( $\tau$ )

$Q := \text{True}; Q' := \tau(Q);$

**while**  $Q \neq Q'$  **do**

$Q := Q';$

$Q' := \tau(Q);$

**end while**

**return**( $Q$ )



# Conclusions

---

- Automatic verification:  
Model + Specification + Model checking
- Explicit state space model checking,  
based on automata theory.
- Extensions: Model checking with real  
time, probability, direct on model,  
partial order reduction...