Mini course on Model Checking

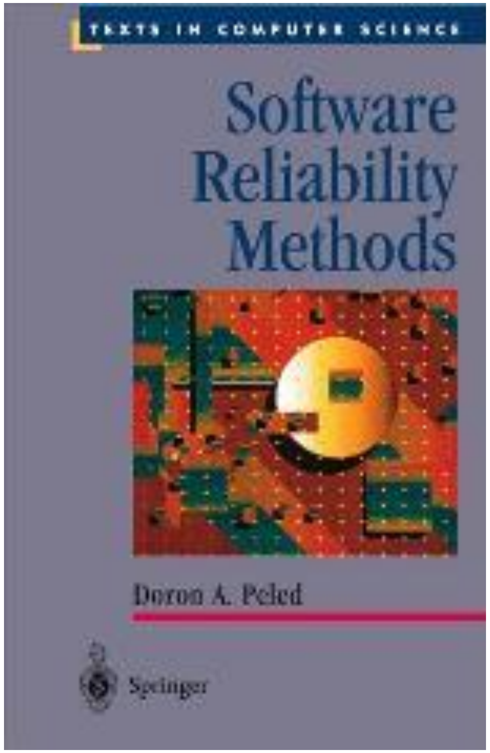
MarktOberdorf Summer School

Prof. Doron A. Peled
Bar Ilan University, Israel

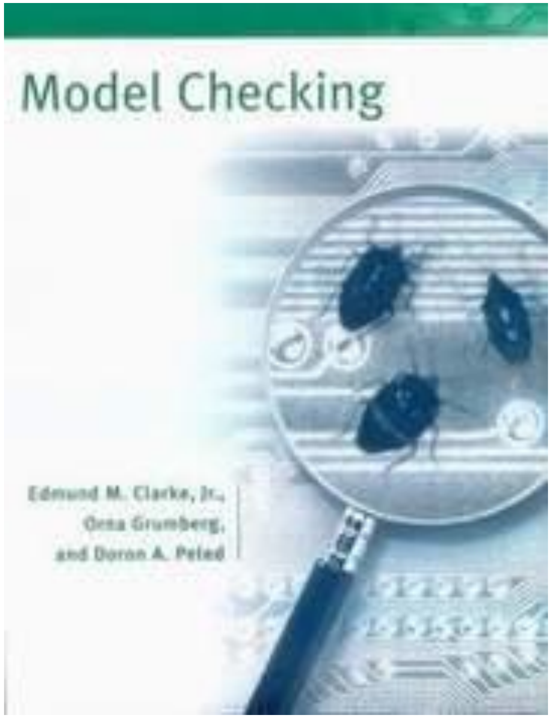
Version 2012

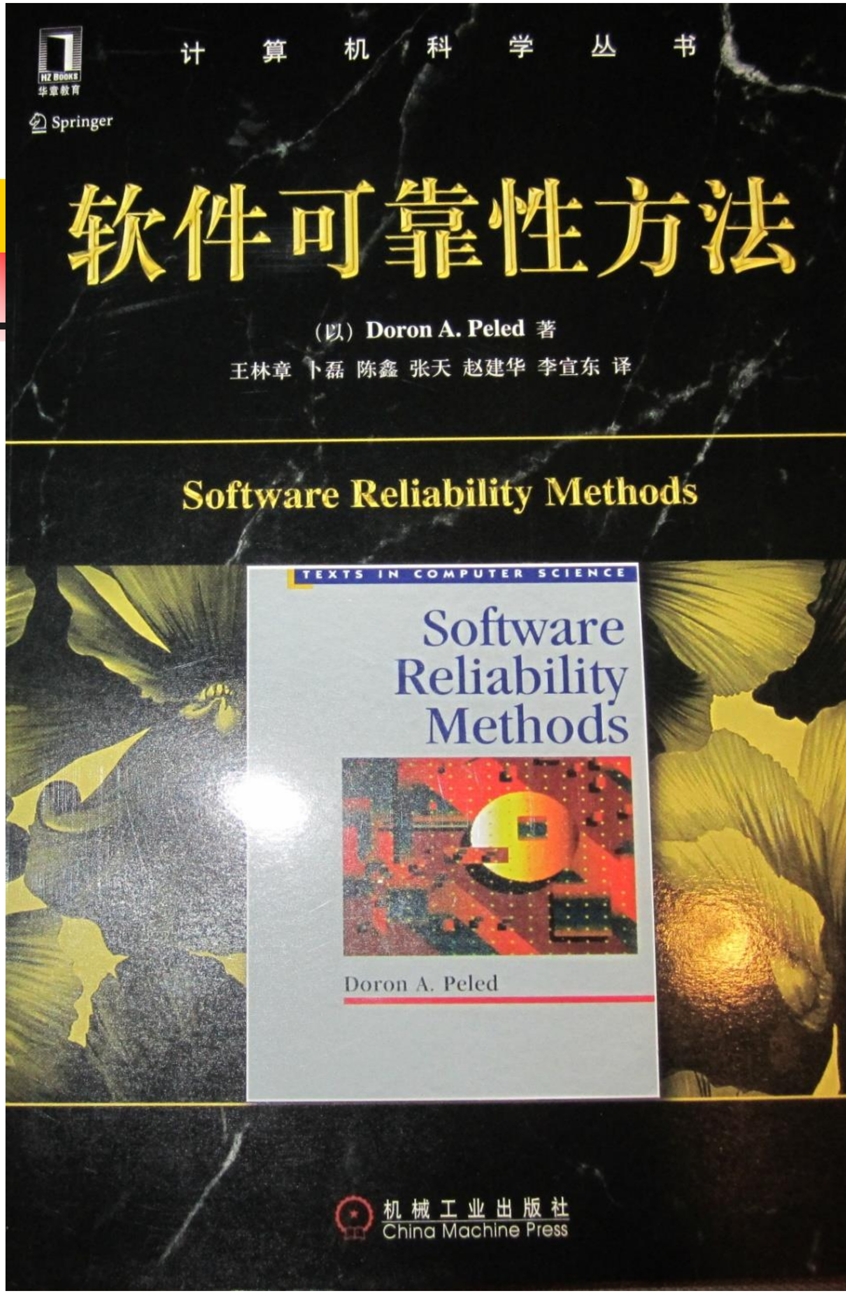
Some related books:

Mainly:



Also:

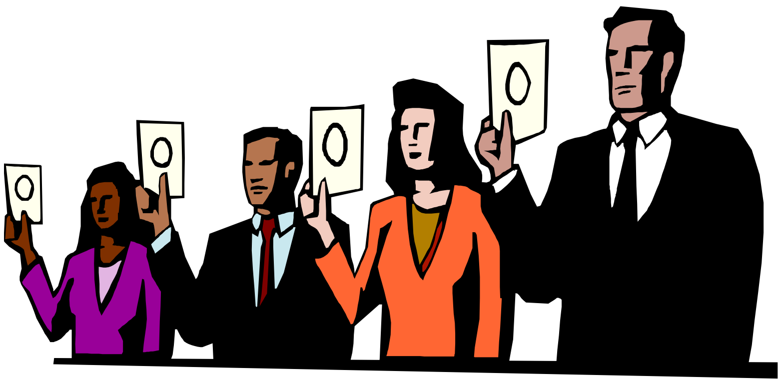



计 算 机 科 学 丛 书
Springer
软件可靠性方法
（以）Doron A. Peled 著
王林章 卜磊 陈鑫 张天 赵建华 李宣东 译
Software Reliability Methods
TEXTS IN COMPUTER SCIENCE
Software Reliability Methods
Doron A. Peled
机械工业出版社
China Machine Press

Goal: software reliability

Use software engineering methodologies to develop the code.

Use formal methods during code development



What are formal methods?

Techniques for analyzing systems, based on some mathematics.

This does not mean that the user must be a mathematician (but here we study the math).

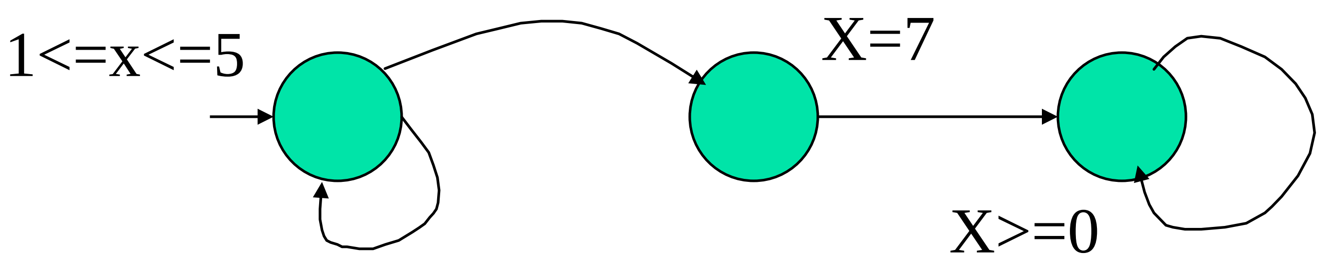
(Ambitious) Plan

- How to model (concurrent) systems?
- How to write a specification using temporal logic and automata on infinite words.
- How to translate TL to automata.
- How to check consistency between model and specification (model checking).
- The SPIN tool (you can us it!)
- Branching time model checking, BDD and CTL.

Specification: Informal, textual, visual

The value of x will be between 1 and 5, until some point where it will become 7. In any case it will never be negative.

(1<=x<=5 U (x=7/\ [] x>=0))



Modeling Software Systems for Analysis

(Book: Chapter 4)

Modelling and specification for verification and validation

- How to specify what the software is supposed to do?
- How to model it in a way that allows us to check it?

Systems of interest

- Sequential systems.
- Concurrent systems (multi-threaded).
  1. Distributive systems.
  2. Reactive systems.
  3. Embedded systems (software + hardware).

Sequential systems.

- Perform some computational task.
- Have some *initial condition*, e.g.,
  $\forall 0 \le i \le n\ A[i]$ integer.
- Have some *final assertion*, e.g.,
  $\forall 0 \le i \le n-1\ A[i] \le A[i+1]$.
  (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.

- $V=\{v_0,v_1,v_2, \ldots\}$ - a set of variables, over some domain.
- $p(v_0, v_1, \ldots, v_n)$ - a parametrized assertion, e.g., $v_0=v_1+v_2$ /\ $v_3>v_4$.
- A state is an assignment of values to the program variables. For example: s=<v_0=1,v1=3,v_3=7,…,v_{18}=2>
- For predicate (first order assertion) *p*: *p*(*s*) is *p* under the assignment *s*. Example: *p* is x>y /\ y>z. *s*=<x=4, y=3, z=5>. Then we have 4>3 /\ 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:
  <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 Transitions

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

Non atomicity

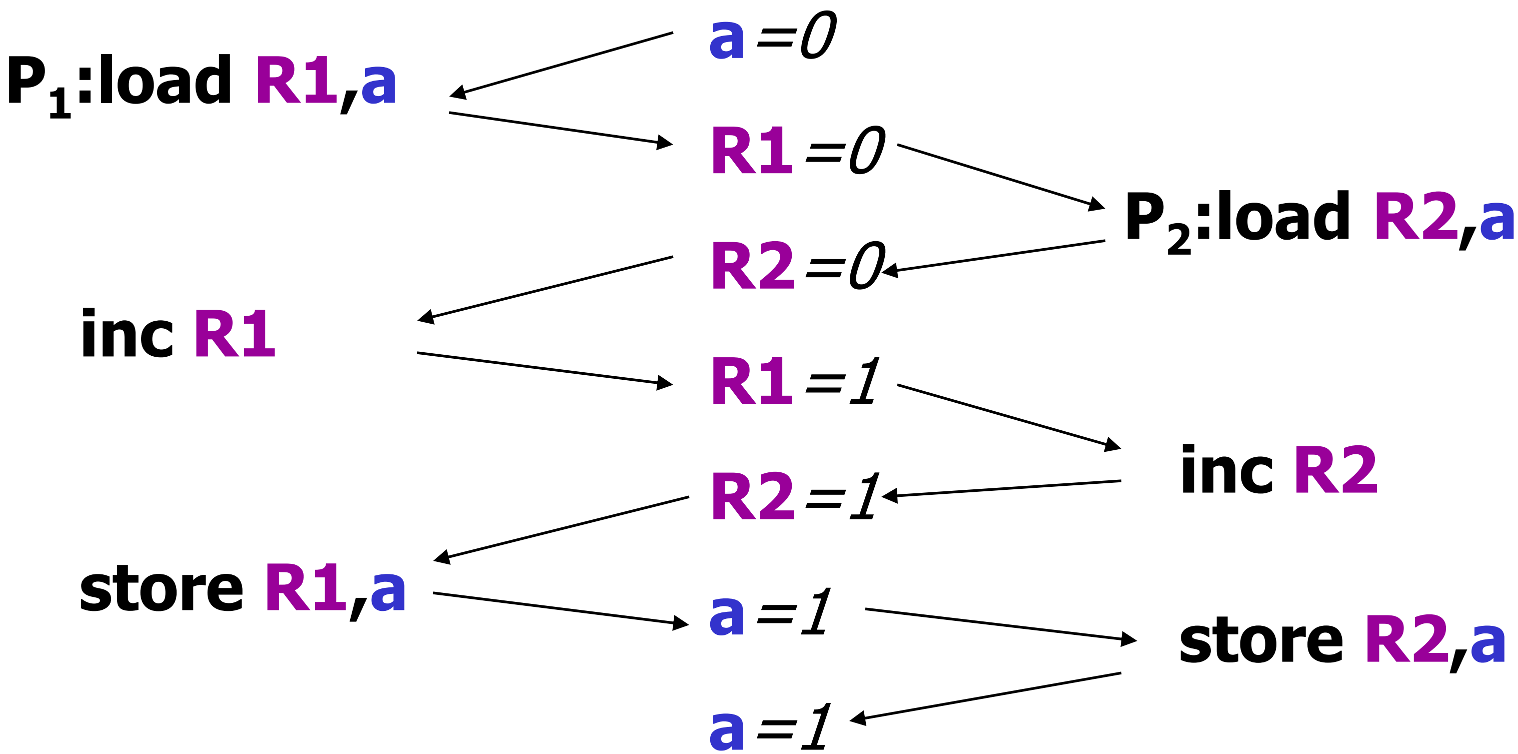
- Execute the following when **a**=0 in two concurrent processes:
- **P_1:a=a+1**
- **P_2:a=a+1**
- Result: **a**=2.
- Is this always the case?

- 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



a=0
P1:load R1,a
R1=0
P2:load R2,a
R2=0
inc R1
R1=1
inc R2
R2=1
store R1,a
a=1
store R2,a
a=1

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 Σ.
- 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

- $V=\{a, b, c, d, e\}$.
- Σ: 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 $s_0, s_1, s_2, \ldots$
  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➔(*c*,*e*):=(*c* -1,*e* +1),

d>0➔(d,e):=(d-1,e+1)}

I: c=a /\ d=b /\ e=0

- s0=<a=2, b=1, c=2, d=1, e=0>
- s1=<a=2, b=1, c=1, d=1, e=1>
- s2=<a=2, b=1, c=1, d=0, e=2>
- s3=<a=2, b=1 ,c=0, d=0, e=3>

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

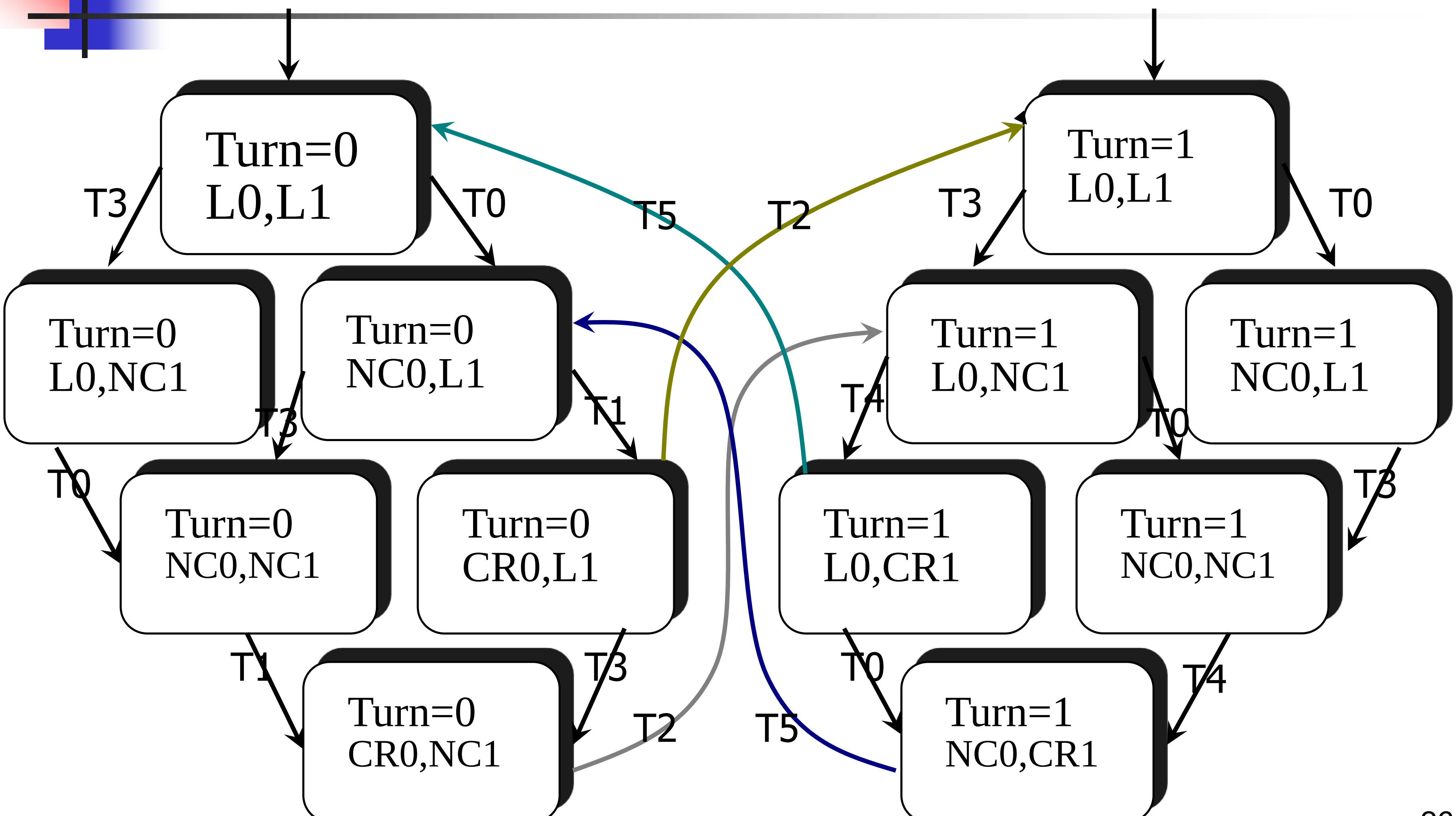
T4:PC1=NC1/\Turn=1➔
PC1:=CR1

T5:PC1=CR1➔
(PC1,Turn):=(L1,0)

Initially: PC0=L0/\PC1=L1

Is this the only reasonable way to model this program?

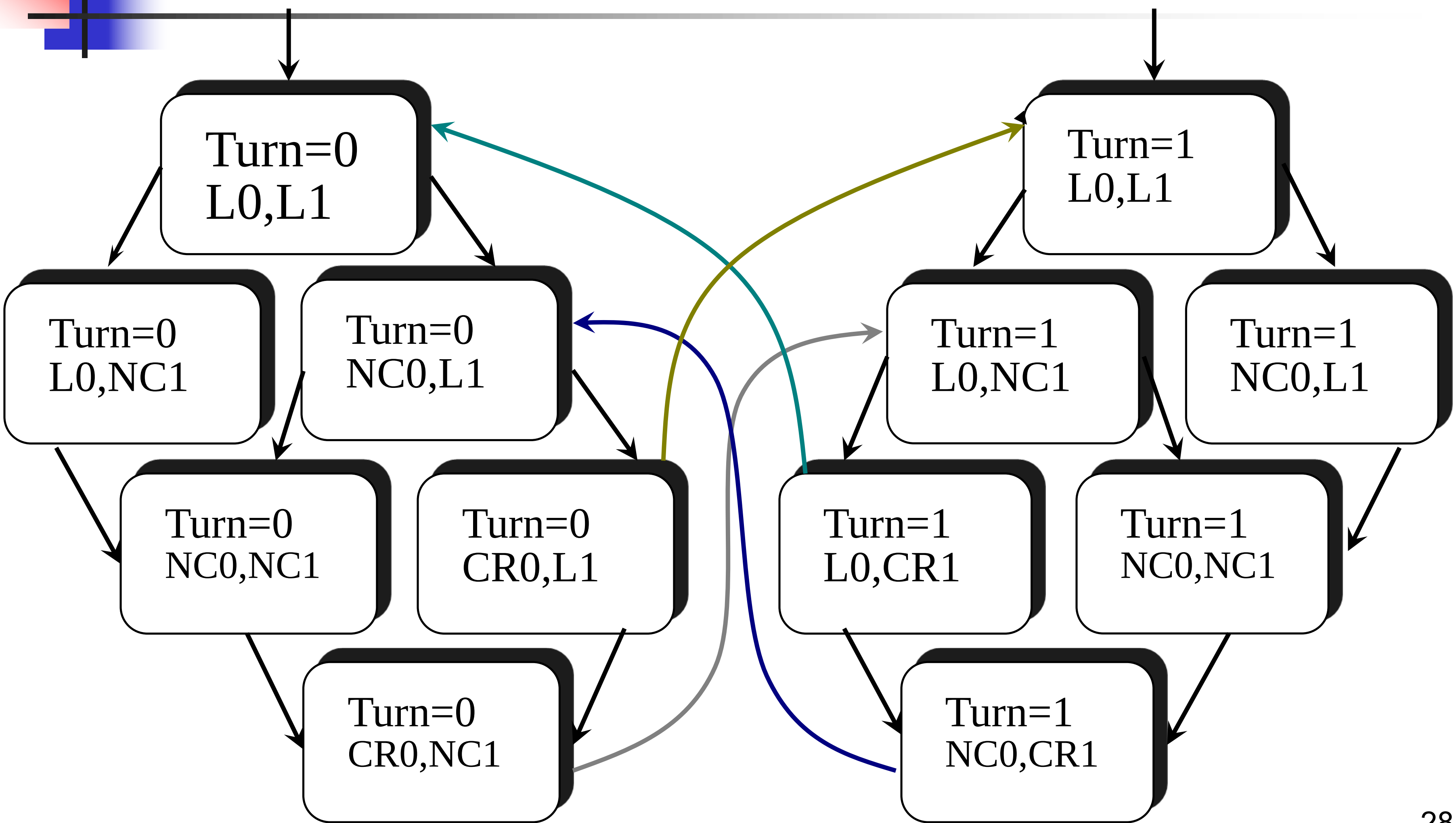
The state graph:Successor relation between *reachable* states.



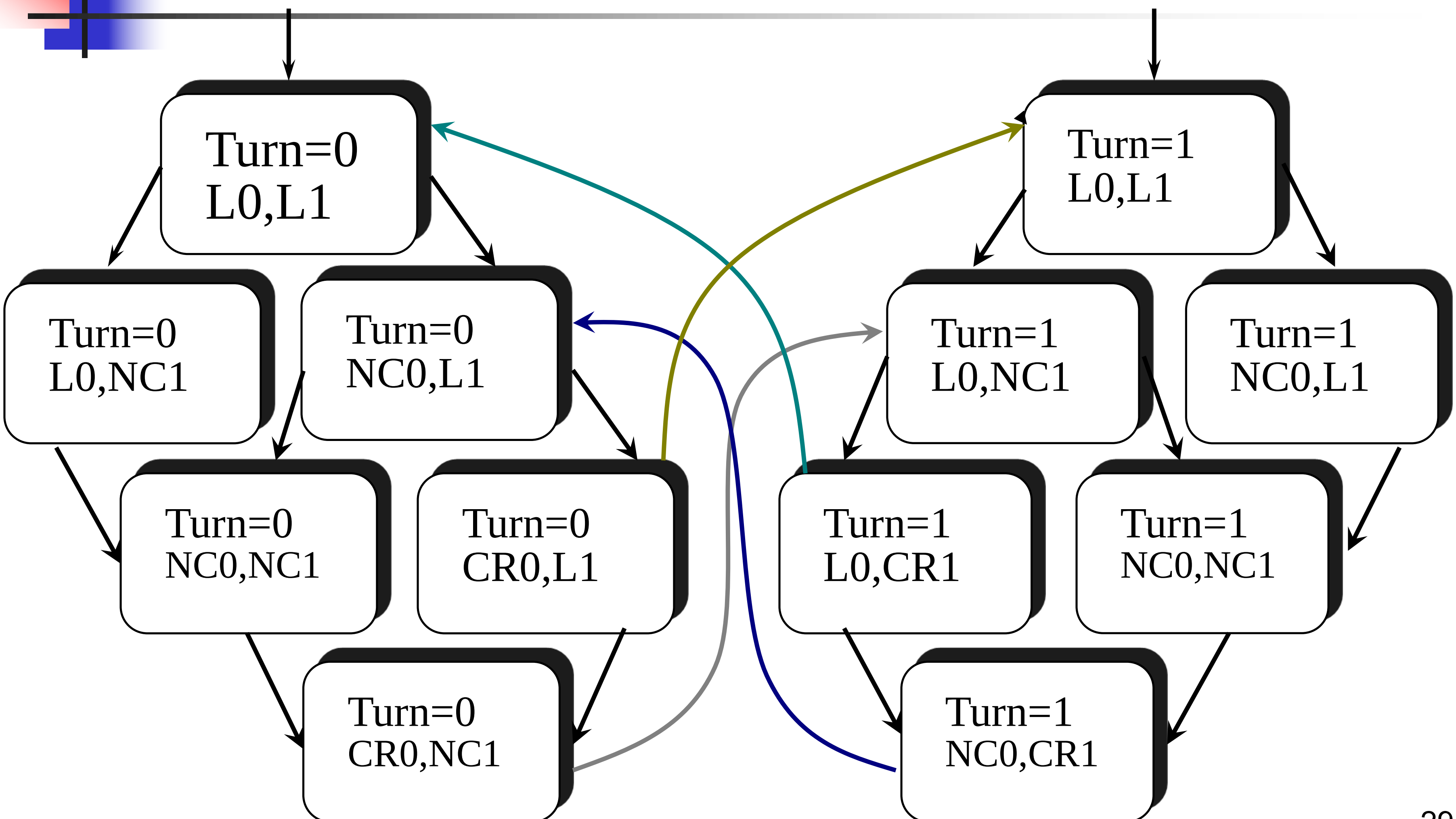
Some important points

- *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.

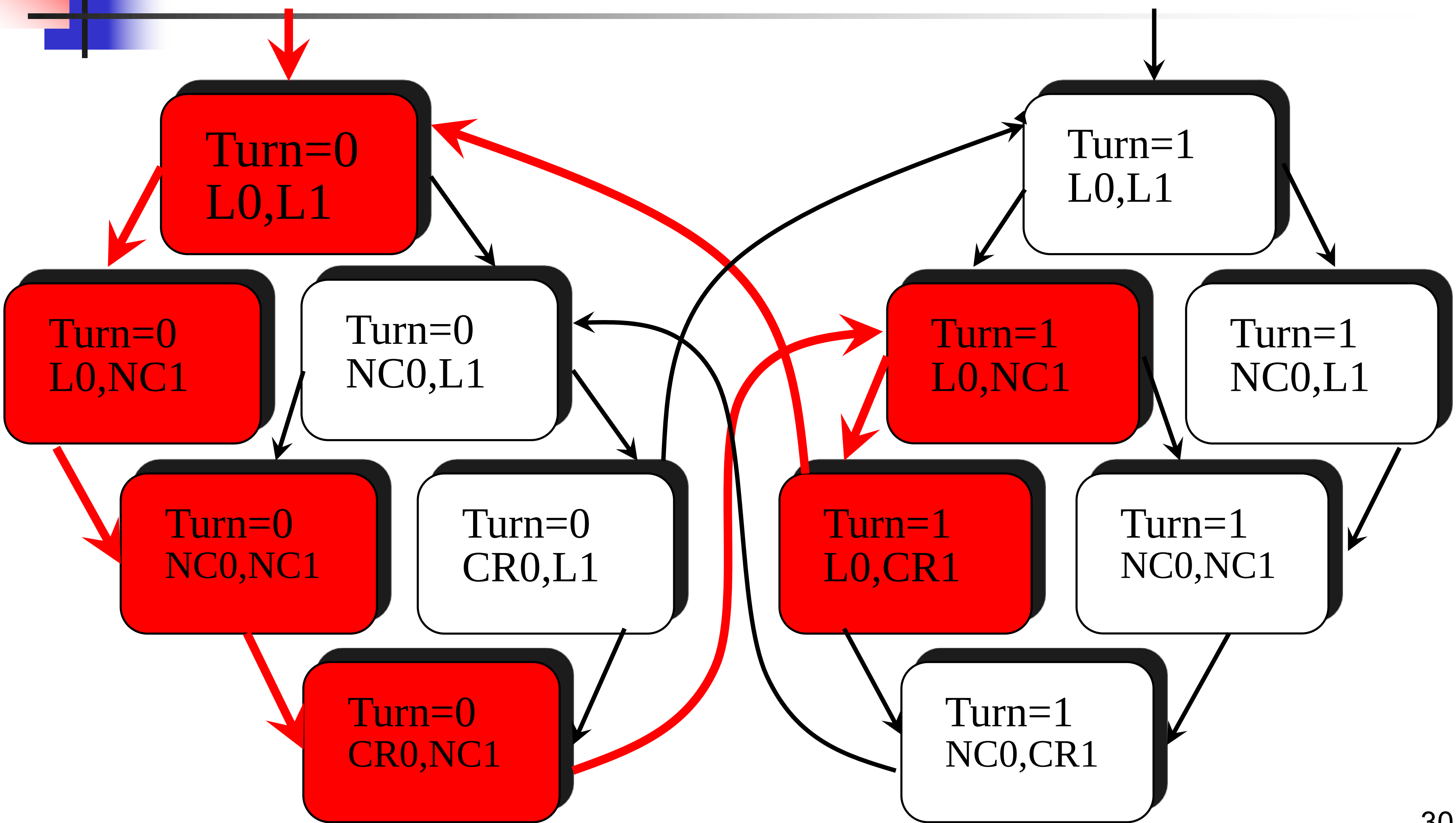
Always ¬(PC0=CR0/\PC1=CR1) *(Mutual exclusion)*



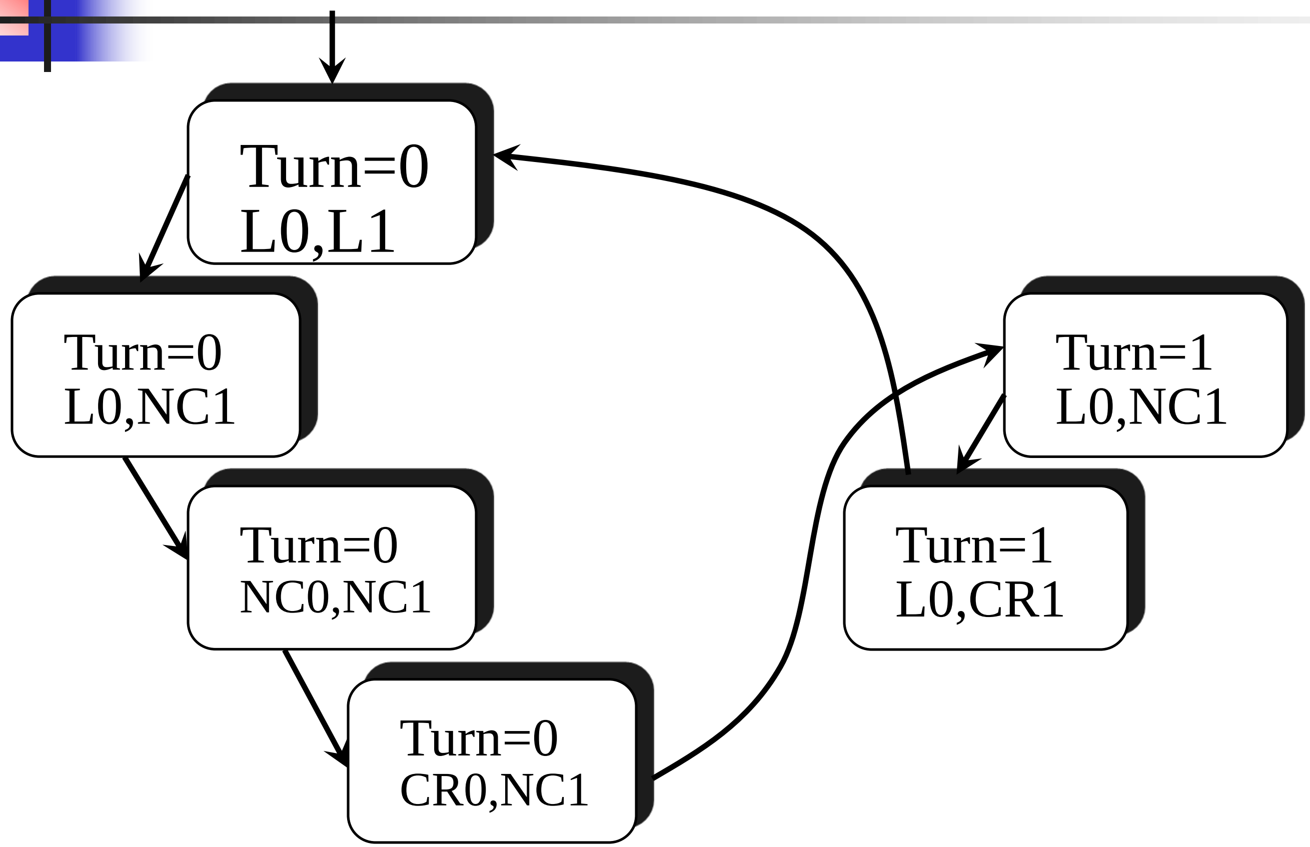
Always if Turn=0 then at some point Turn=1



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



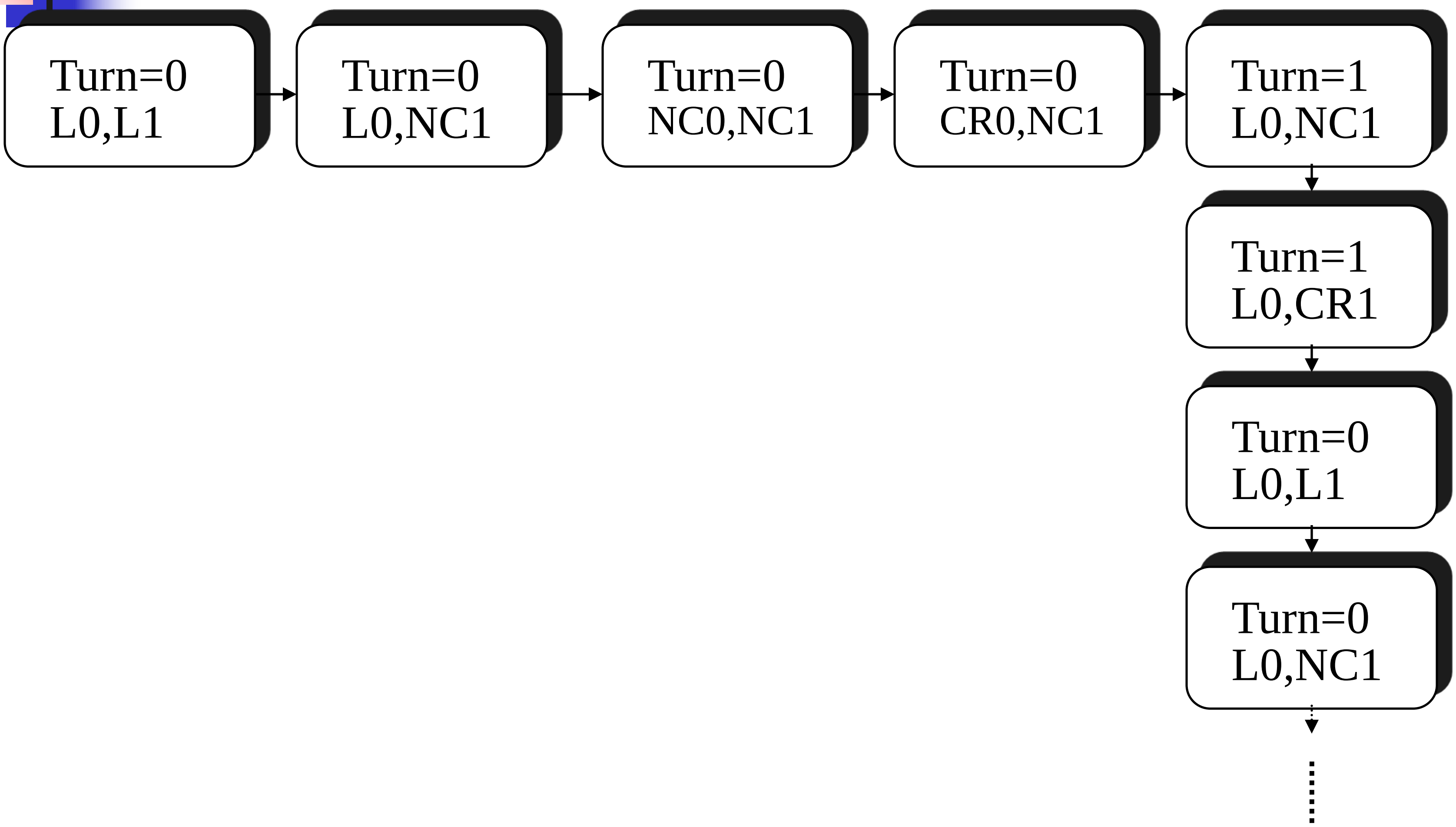
Interleaving semantics: Execute one transition at a time.



Need to check the property

for every possible interleaving!

Interleaving semantics



Busy waiting

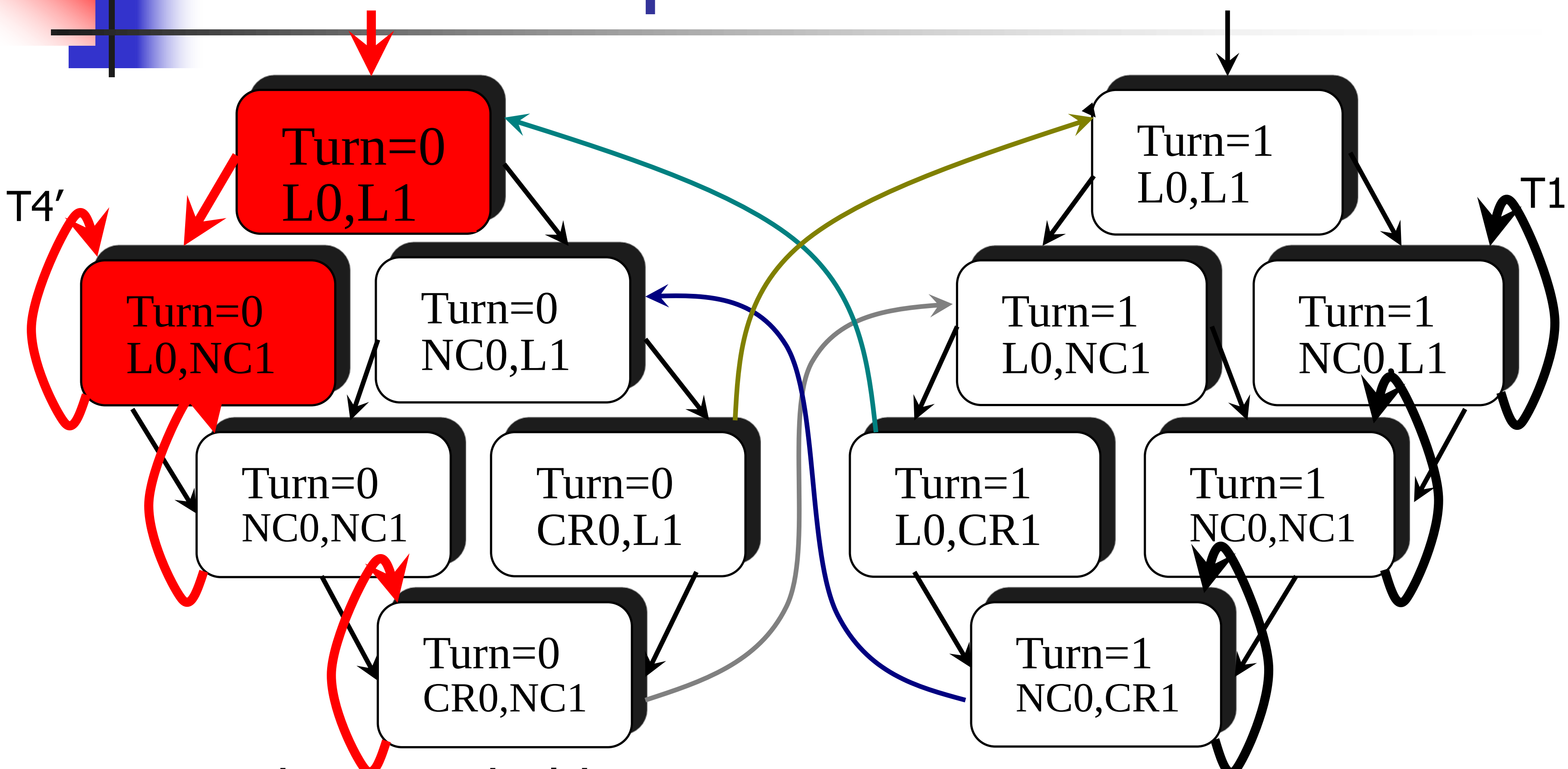
L0:While True do
NC0:wait(Turn=0);
CR0:Turn=1
endwhile ||
L1:While True do
NC1:wait(Turn=1);
CR1:Turn=0
endwhile

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

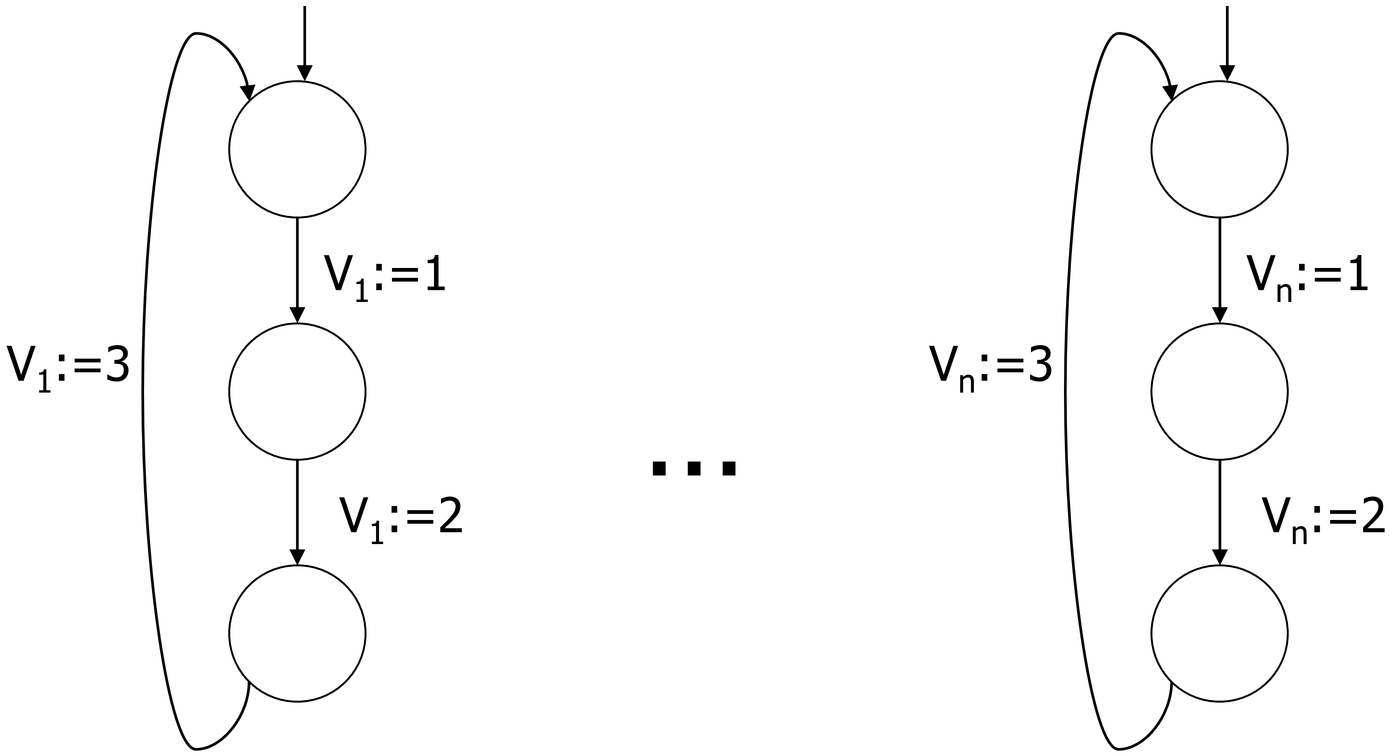
Always when Turn=0 then at some point Turn=1



Now it does not hold!

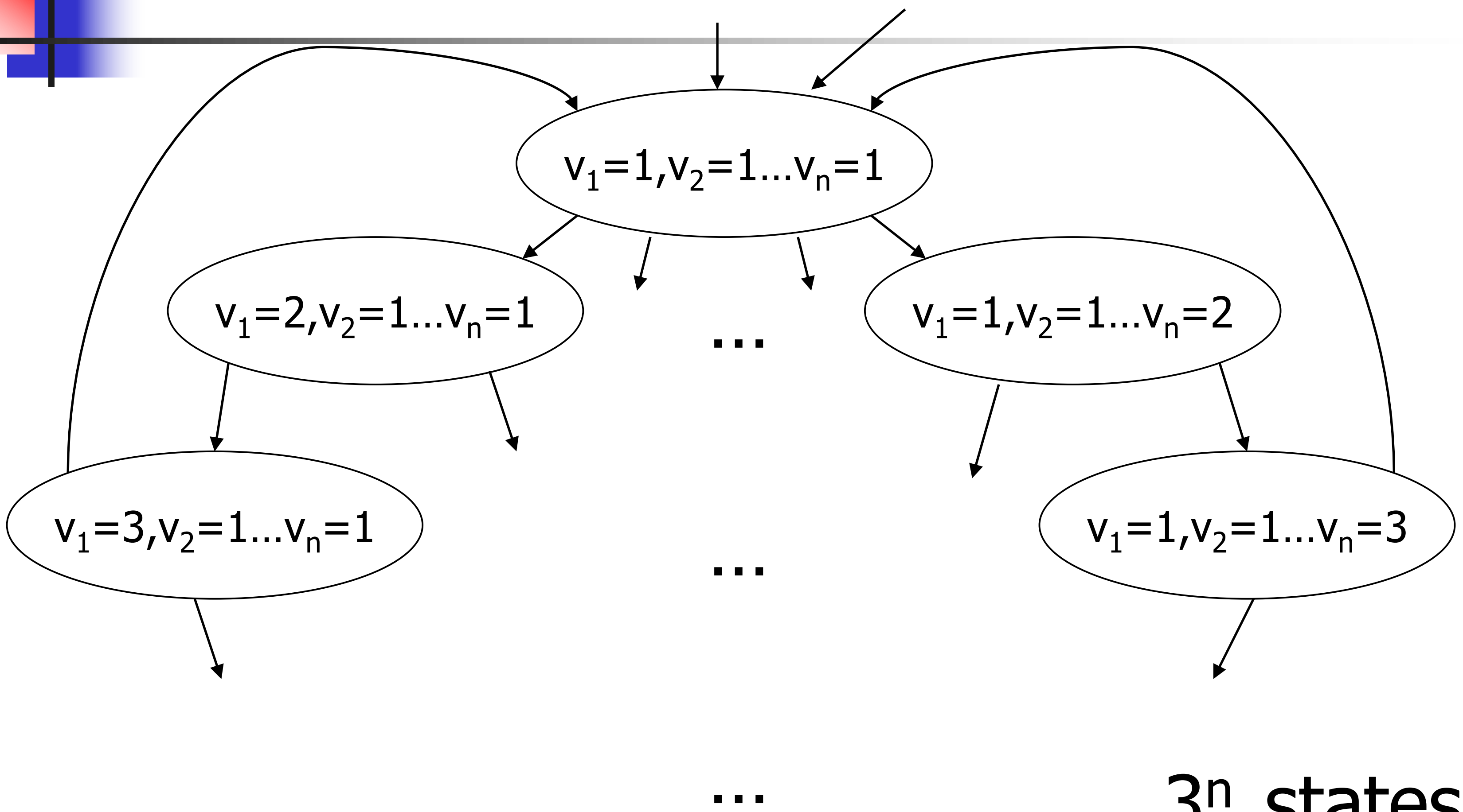
(Red subgraph generates a counterexample execution.)

Combinatorial explosion



How many states?

Global states



3^n 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*?

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

φ ::= (φ) | ¬φ | φ /\ φ | φ \/ φ | φ U φ | [] φ | <>φ | O φ | p

[] φ –– "box", "always", "forever"

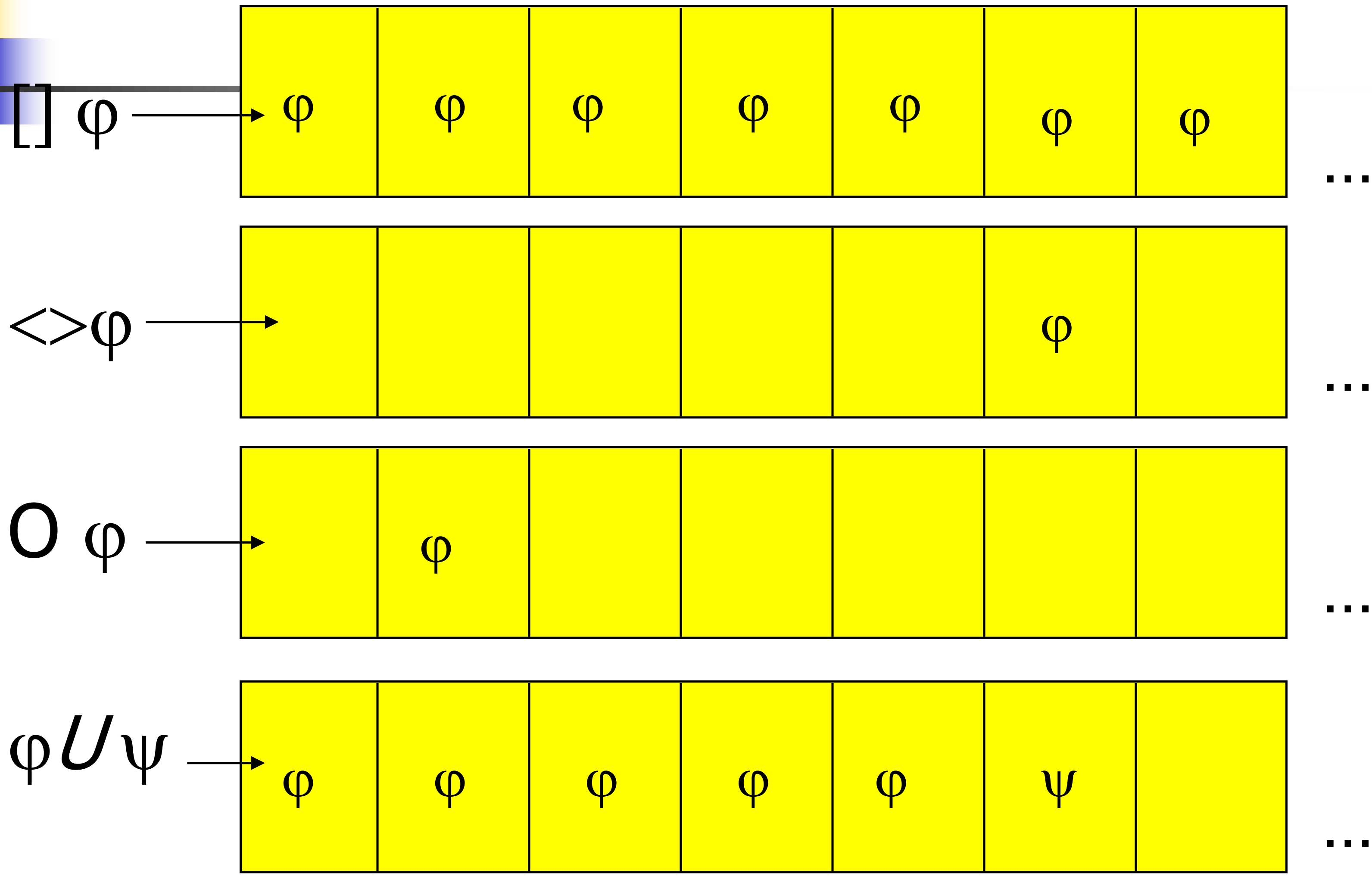
<>φ –– "diamond", "eventually", "sometimes"

O φ –– "nexttime"

φ U ψ –– "until"

Propositions *p*, *q*, *r*, … Each represents some state property (x>y+1, z=t, at_CR, etc.)

Semantics *over **suffixes** of execution*



Can discard some operators

- Instead of $<>p$, write $true\ U\ p$.
- Instead of $[]p$, we can write $\neg(<>\neg p)$, 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 ¬p holds or $<>\neg p$.

Combinations

- $[]<>p$ “*p* will happen infinitely often”
- $<>[]p$ “*p* will happen from some point forever”.
- $([]<>p) \rightarrow ([]<>q)$ “If *p* happens infinitely often, then *q* also happens infinitely often”.

Some relations:

- $[](\varphi /\backslash \psi)=([]\varphi)/\backslash([]\psi)$
- But $<>(\varphi /\backslash \psi)\neq(<>\varphi)/\backslash(<>\psi)$

		ψ		φ		

…

- $<>(\varphi \vee \psi)=(<>\varphi)\vee(<>\psi)$
- But $[](\varphi \vee \psi)\neq([]\varphi)\vee([]\psi)$

ψ	φ ψ	ψ	φ ψ	φ	ψ	φ

…

What about

- $([]<>\varphi) \wedge ([]<>\psi) = []<>(\varphi \wedge \psi)$? No, just ←
- $([]<>\varphi) \vee ([]<>\psi) = []<>(\varphi \vee \psi)$? Yes!!!
- $(<>[]\varphi) \wedge (<>[]\psi) = <>[](\varphi \wedge \psi)$? Yes!!!
- $(<>[]\varphi) \vee (<>[]\psi) = <>[](\varphi \vee \psi)$? No, just →

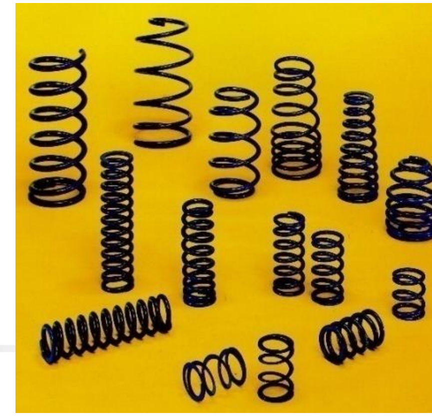
Formal semantic definition

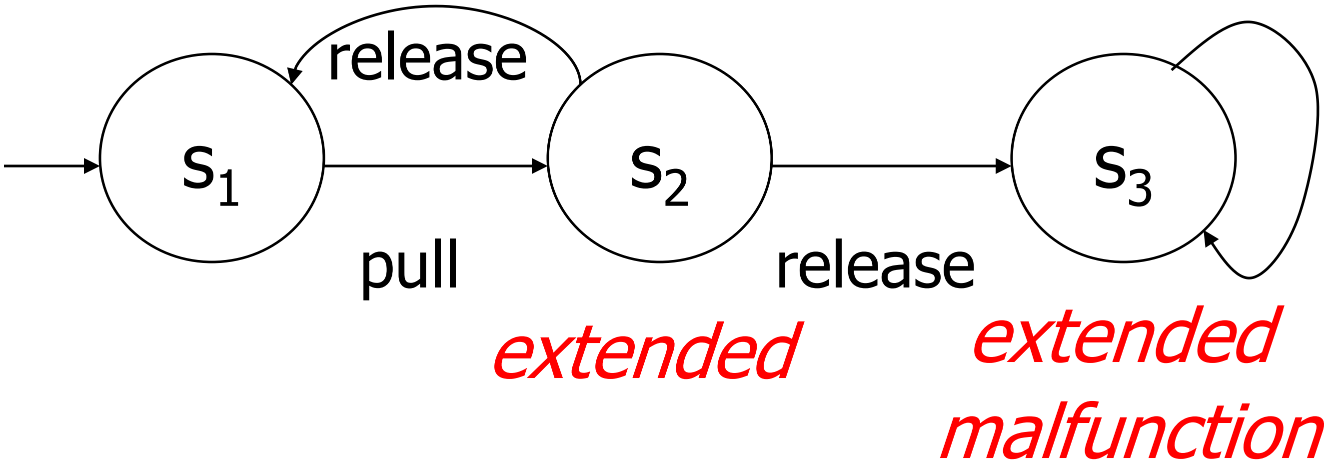
- Let σ be a sequence $s_0\ s_1\ s_2\ \ldots$
- Let σ^i be a suffix of σ: $s_i\ s_{i+1}\ s_{i+2}\ \ldots$ ($\sigma^0 = \sigma$)
- $\sigma^i \models p$, where p a proposition, if $s_i \models p$.
- $\sigma^i \models \varphi /\backslash \psi$ if $\sigma^i \models \varphi$ and $\sigma^i \models \psi$.
- $\sigma^i \models \varphi \backslash/ \psi$ if $\sigma^i \models \varphi$ or $\sigma^i \models \psi$.
- $\sigma^i \models \neg\varphi$ if it is not the case that $\sigma^i \models \varphi$.
- $\sigma^i \models <>\varphi$ 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 $i \leq k < j$, $\sigma^k \models \varphi$.
- How to define $\sigma^i \models O\varphi$?

Then we interpret:

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

Spring Example





$r_0 = s_1\ s_2\ s_1\ s_2\ s_1\ s_2\ s_1\ \ldots$

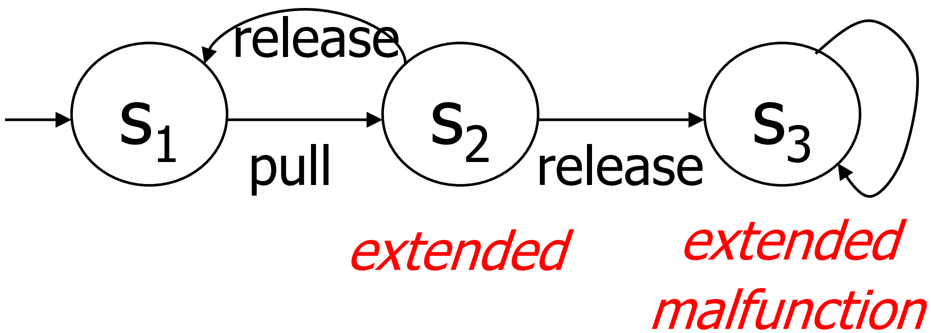
$r_1 = s_1\ s_2\ s_3\ s_3\ s_3\ s_3\ s_3\ \ldots$

$r_2 = s_1\ s_2\ s_1\ s_2\ s_3\ s_3\ s_3\ \ldots$

…

LTL satisfaction by a single sequence

$r_2 = s_1\ s_2\ s_1\ s_2\ s_3\ s_3\ s_3\ \ldots$



r_2 |= extended ??

r_2 |= O extended ??

r_2 |= O O extended ??

r_2 |= <> extended ??

r_2 |= [] extended ??

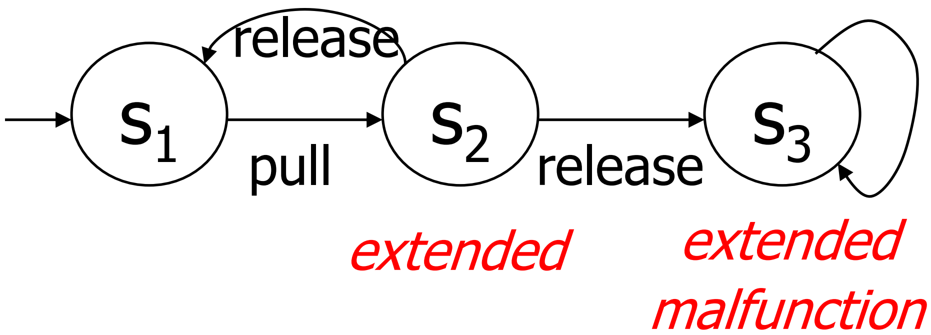
r_2 |= <>[] extended ??

r_2 |= ¬ <>[] extended ??

r_2 |= (¬extended) *U* malfunction ??

r_2 |= [](¬extended->O extended) ??

LTL satisfaction by a system



P |= extended ??

P |= O extended ??

P |= O O extended ??

P |= <> extended ??

P|= [] extended ??

P |= <>[] extended ??

P |= ¬ <>[] extended ??

P |= (¬extended) *U* malfunction ??

P |= [](¬extended->O extended) ??

More specifications

- [] (PC0=NC0 → <> PC0=CR0)
- [] (PC0=NC0 *U* Turn=0)
- Try at home:
  - The processes alternate in entering their critical sections.
  - Each process enters its critical section infinitely often.

Proof system

- $\neg\langle\rangle p \leftrightarrow [\,]\neg p$
- $[\,](p \rightarrow q) \rightarrow ([\,]p \rightarrow [\,]q)$
- $[\,]p \rightarrow (p \wedge O[\,]p)$
- $O\neg p \leftrightarrow \neg Op$
- $[\,](p \rightarrow Op) \rightarrow (p \rightarrow [\,]p)$
- $(pUq) \leftrightarrow (q \vee (p \wedge O(pUq)))$
- $(pUq) \rightarrow \langle\rangle q$

- + propositional logic axiomatization.
- + proof rule:

$$\frac{p}{[\,]p}$$

- But, there is actually no need to do proofs!! Use algorithms instead

Traffic light example

Green → Yellow → Red

Always has exactly one light:

$$[\,](\neg(gr \wedge ye) \wedge \neg(ye \wedge re) \wedge \neg(re \wedge gr) \wedge (gr \vee ye \vee re))$$

Correct change of color:

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

Another kind of traffic light

Green → Yellow → Red → Yellow

First attempt:

$\cancel{[\,]((gr \lor re \rightarrow (gr \lor re)\ U\ ye) \lor (ye \rightarrow ye\ U\ (gr \lor re)))}$

Correct specification:

$[\,](\ (gr \rightarrow (gr\ U\ (ye \land (\ ye\ U\ re\ ))))$

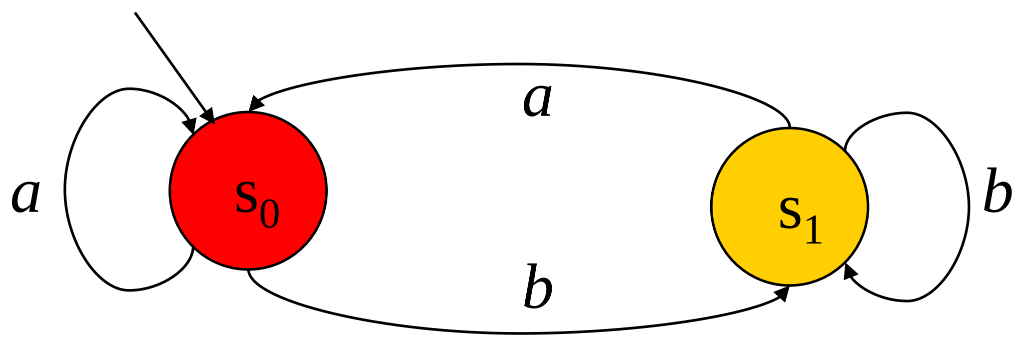
$\land(re \rightarrow (re\ U\ (ye \land (\ ye\ U\ gr\ ))))$

$\land(ye \rightarrow (ye\ U\ (gr \lor re))))$

Needed only when we can start with yellow

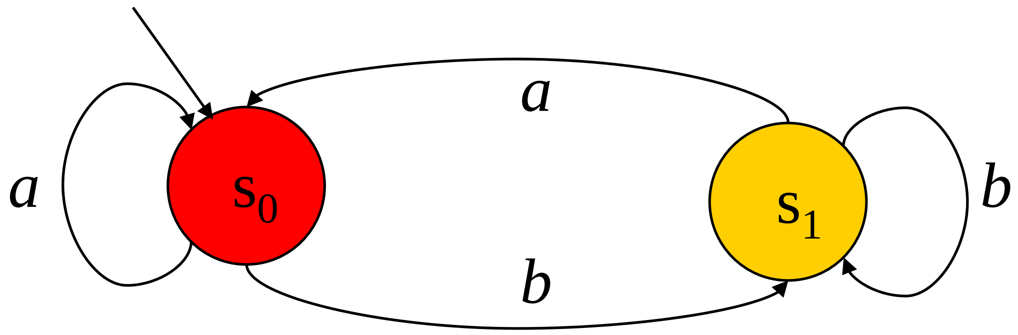
Automata over finite words

- $A=<\Sigma, S, \Delta, I, F>$
- Σ (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.



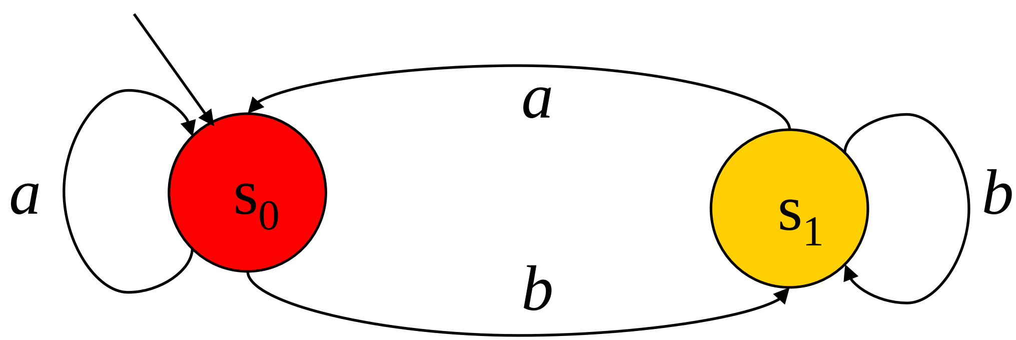
The transition relation

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



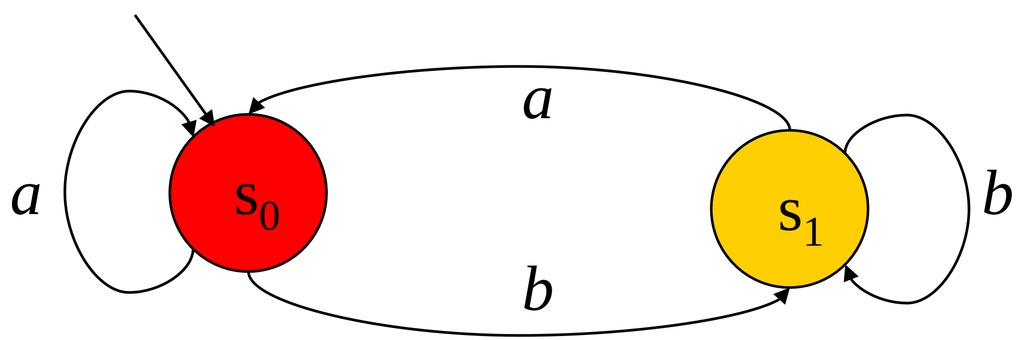
A *run* over a word

- A word over Σ, 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_i, s_{i+1}).
- Accepting if ends at accepting state.



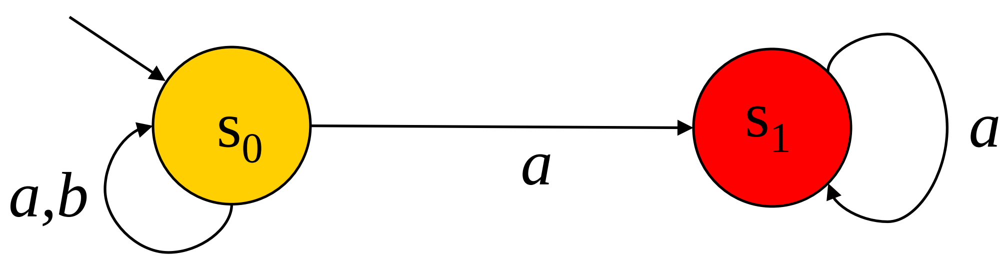
The *language* of an automaton

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

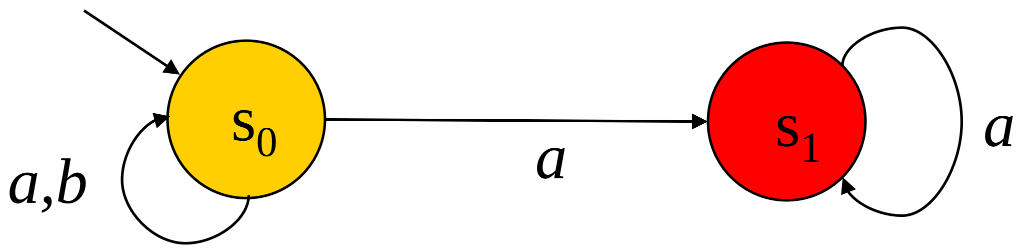


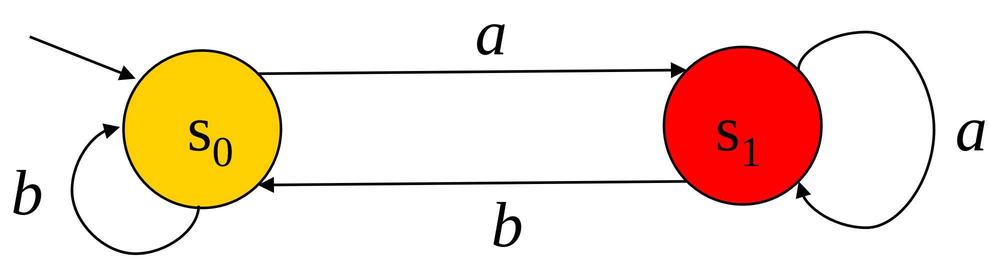
Nondeterministic automaton

- 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?



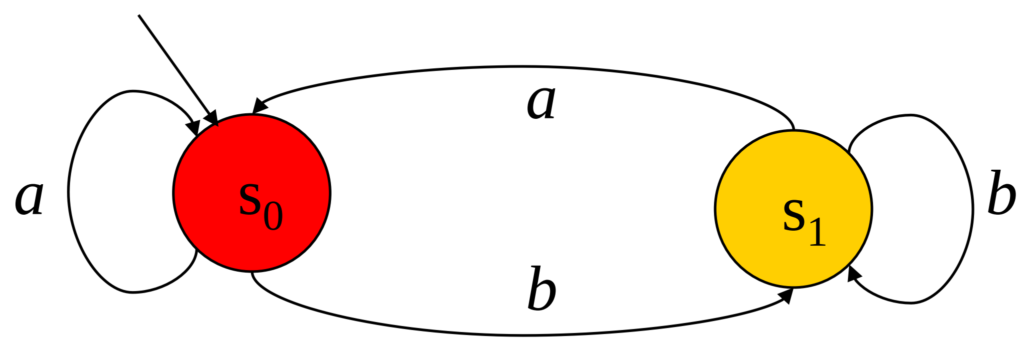
Equivalent deterministic automaton





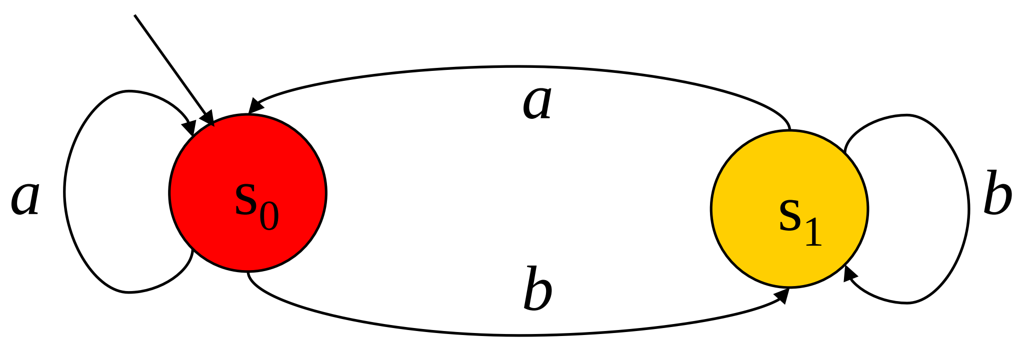
Automata over infinite words

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



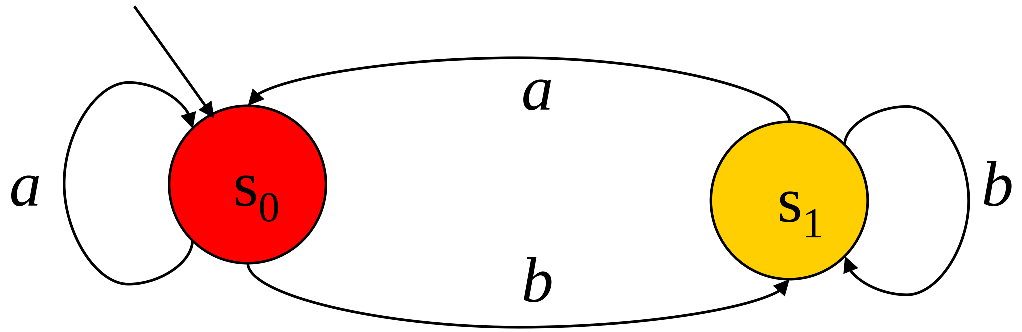
Automata over infinite words (Büchi automata,ω-automata)

- Consider the word *abababab*…
- There is a run $s_0 s_0 s_1 s_0 s_1 s_0 s_1$ …
- This run in accepting, since s_0 appears infinitely many times.



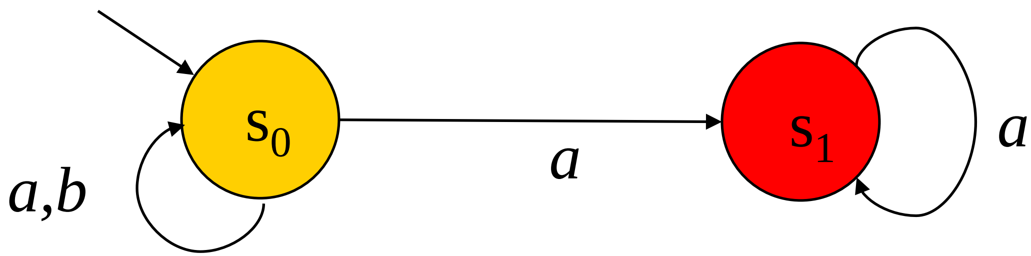
Other runs

- For the word *bbbbb*... the run is $s_0\ s_1\ s_1\ s_1\ s_1$... and is not accepting.
- For the word *aaabbbbb* ..., the run is $s_0\ s_0\ s_0\ s_0\ s_1\ s_1\ s_1\ s_1$ ...
- What is the run for *ababbabbb* ...?



Nondeterministic automaton

- What is the language of this automaton?
- What is the LTL specification if
  b -- PC0=CR0, *a* =¬*b*?



•Can you find a deterministic automaton with same language?

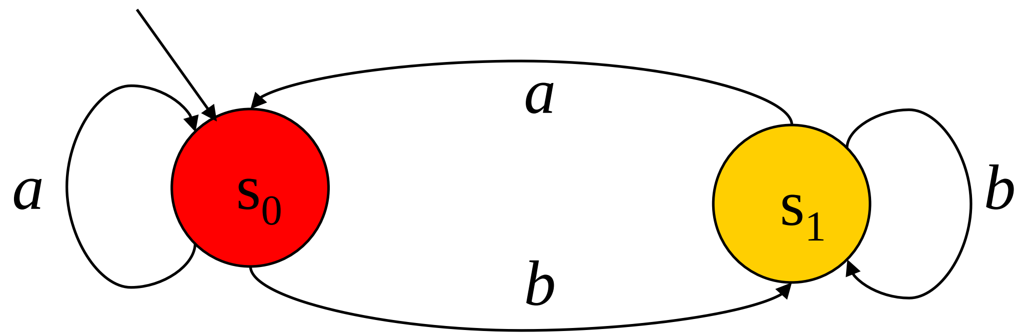
•Can you prove there is no such deterministic automaton?

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.

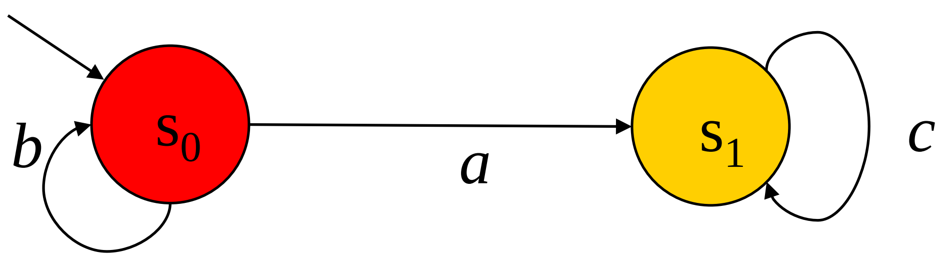
Specification using Automata

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



Mutual Exclusion

- *a* -- PC0=CR0/\PC1=CR1
- *b* -- ¬(PC0=CR0/\PC1=CR1)
- *c* -- true
- []¬(PC0=CR0/\PC1=CR1)



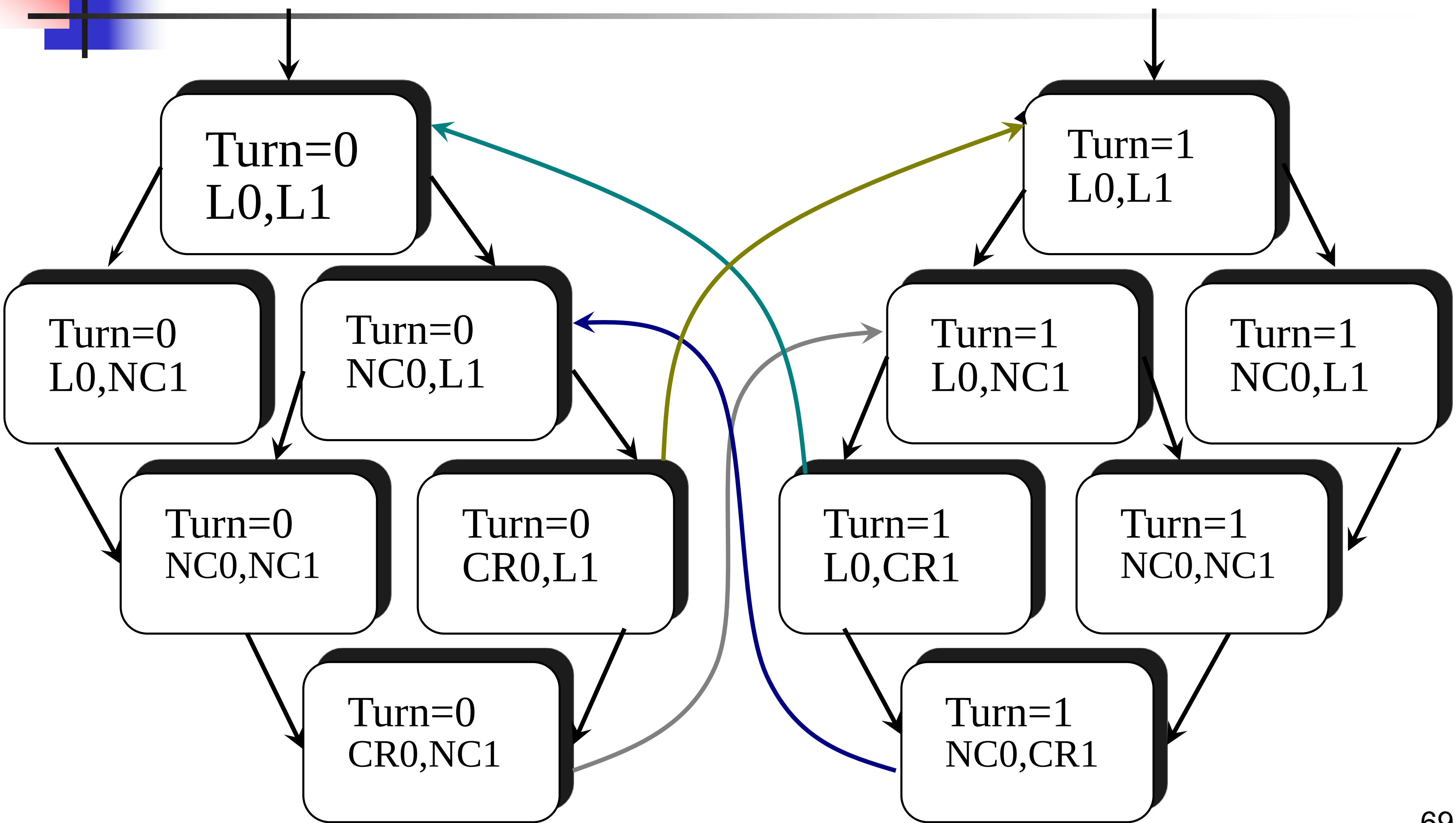
Apply now to our program:

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

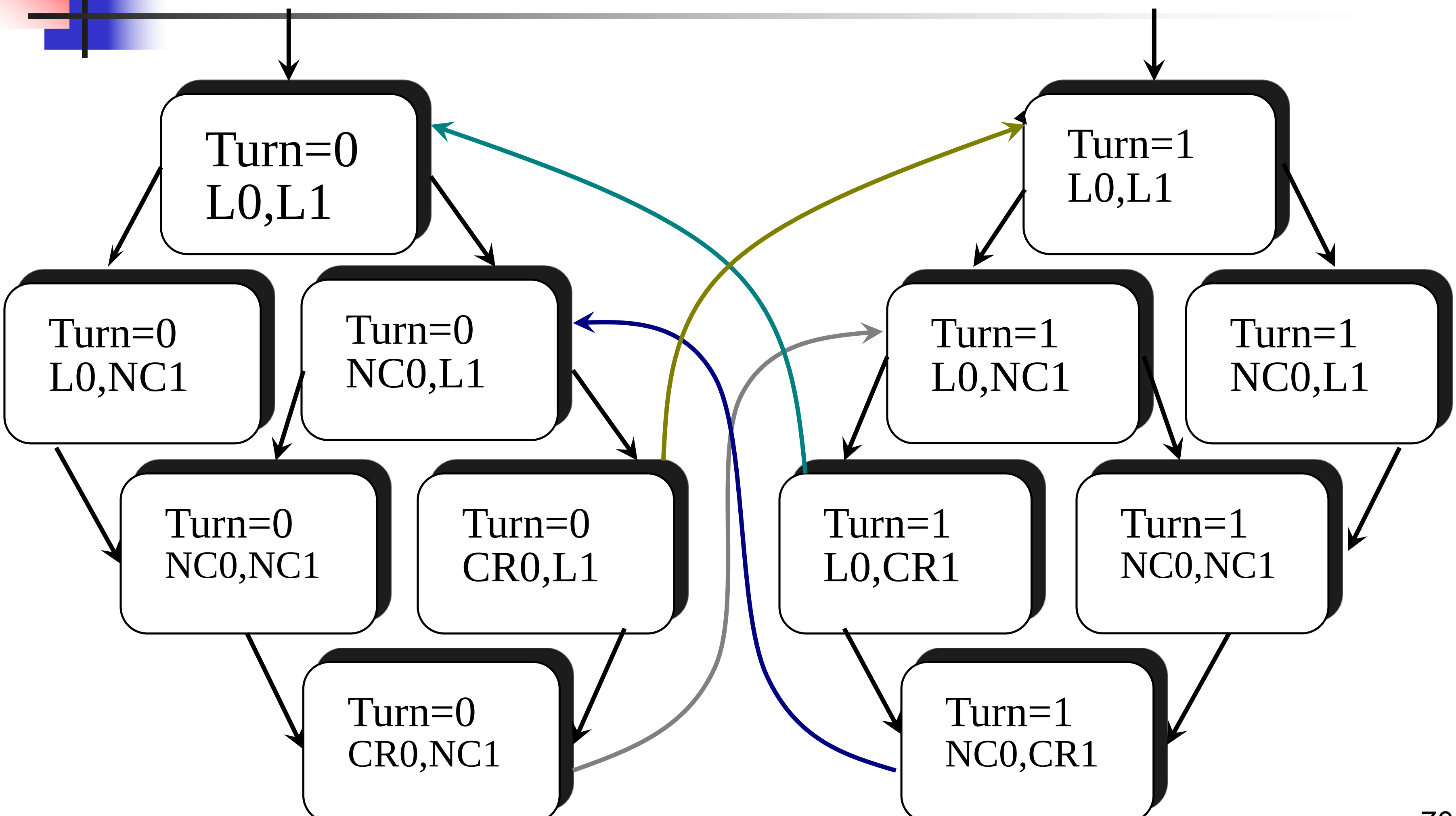
T0:PC0=L0➔PC0=NC0
T1:PC0=NC0/\Turn=0➔
  PC0:=CR0
T2:PC0=CR0➔
  (PC0,Turn):=(L0,1)
T3:PC1==L1➔PC1=NC1
T4:PC1=NC1/\Turn=1➔
  PC1:=CR1
T5:PC1=CR1➔
  (PC1,Turn):=(L1,0)

Initially: PC0=L0/\PC1=L1

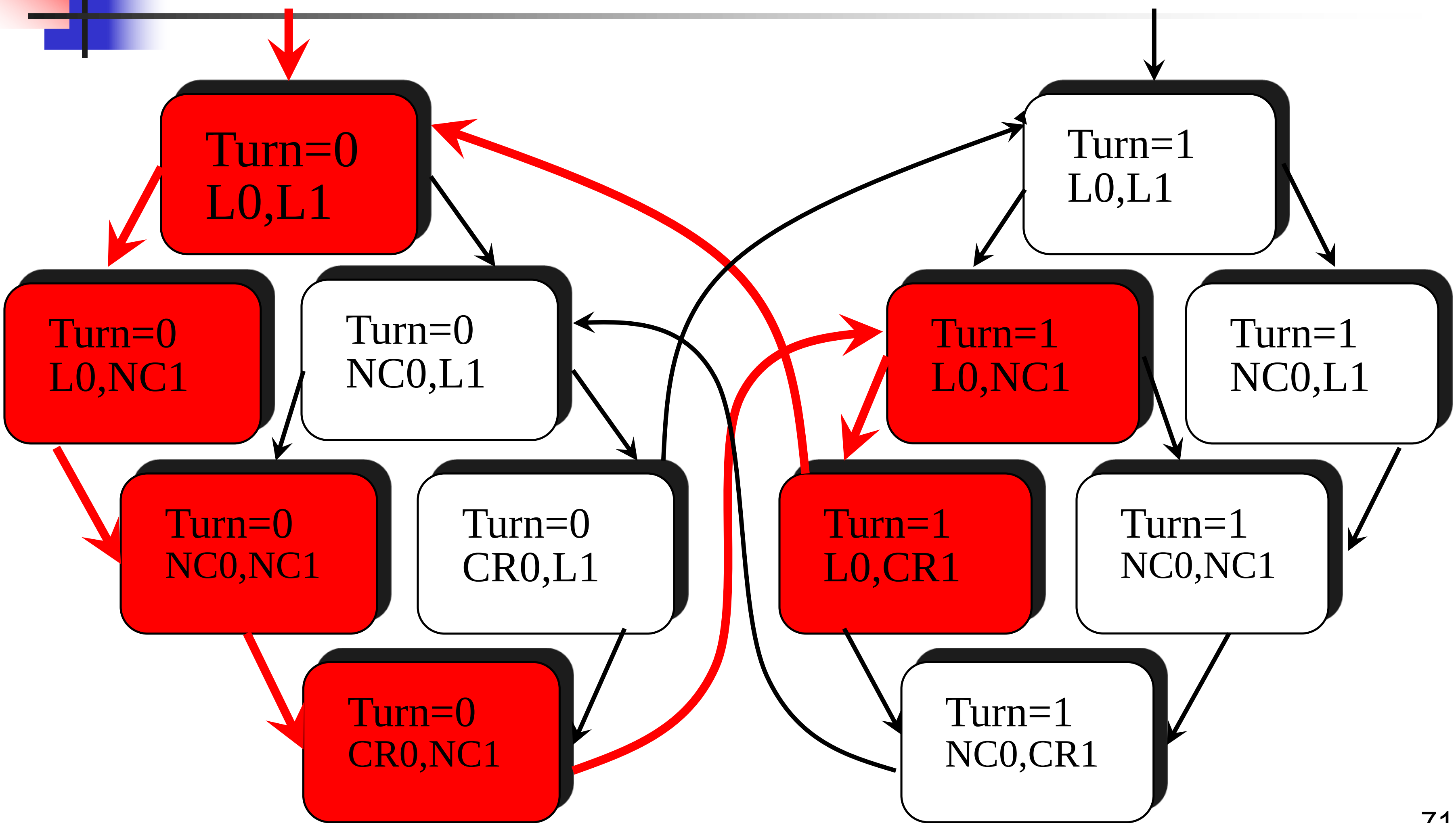
The state space



[]¬(PC0=CR0/\PC1=CR1) *(Mutual exclusion)*

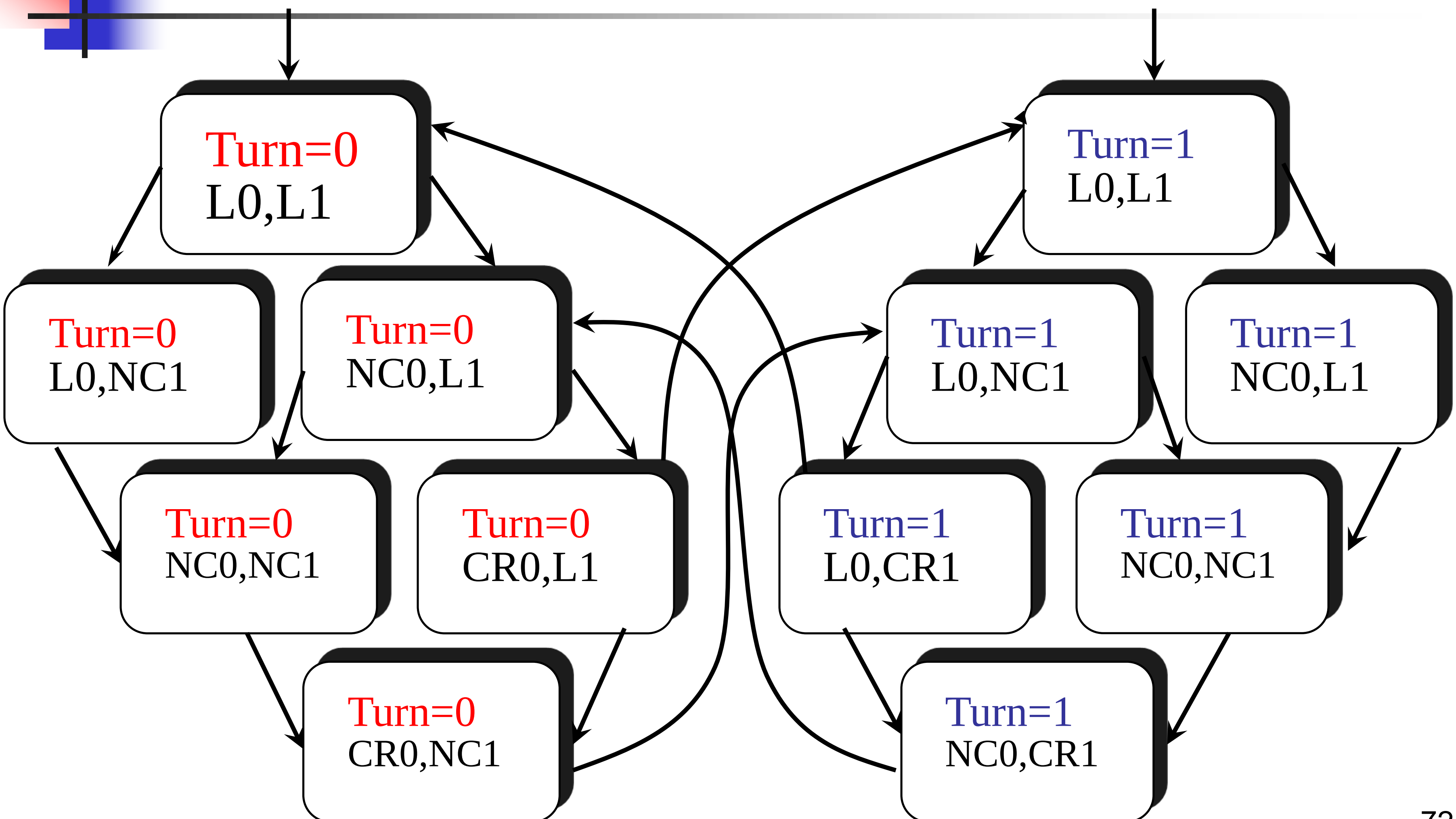


[](Turn=0 →<>Turn=1)



Turn=0
L0,L1
Turn=0
L0,NC1
Turn=0
NC0,L1
Turn=0
NC0,NC1
Turn=0
CR0,L1
Turn=0
CR0,NC1
Turn=1
L0,L1
Turn=1
L0,NC1
Turn=1
NC0,L1
Turn=1
L0,CR1
Turn=1
NC0,NC1
Turn=1
NC0,CR1

[](Turn=0 → <>Turn=1)

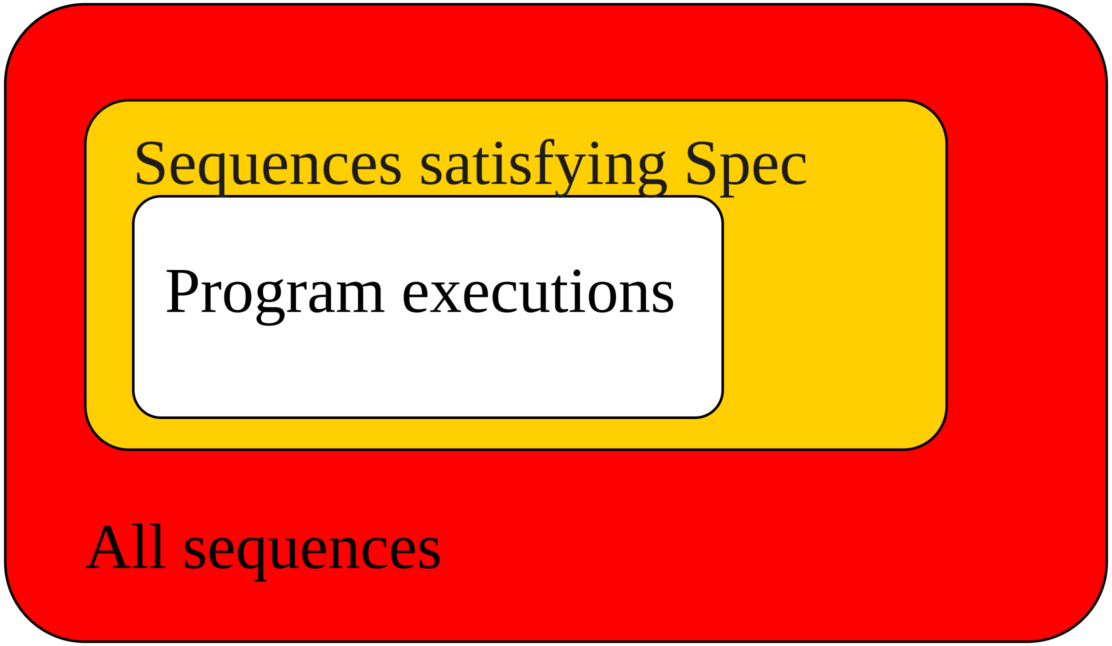


Turn=0
L0,L1
Turn=0
L0,NC1
Turn=0
NC0,L1
Turn=0
NC0,NC1
Turn=0
CR0,L1
Turn=0
CR0,NC1
Turn=1
L0,L1
Turn=1
L0,NC1
Turn=1
NC0,L1
Turn=1
L0,CR1
Turn=1
NC0,NC1
Turn=1
NC0,CR1

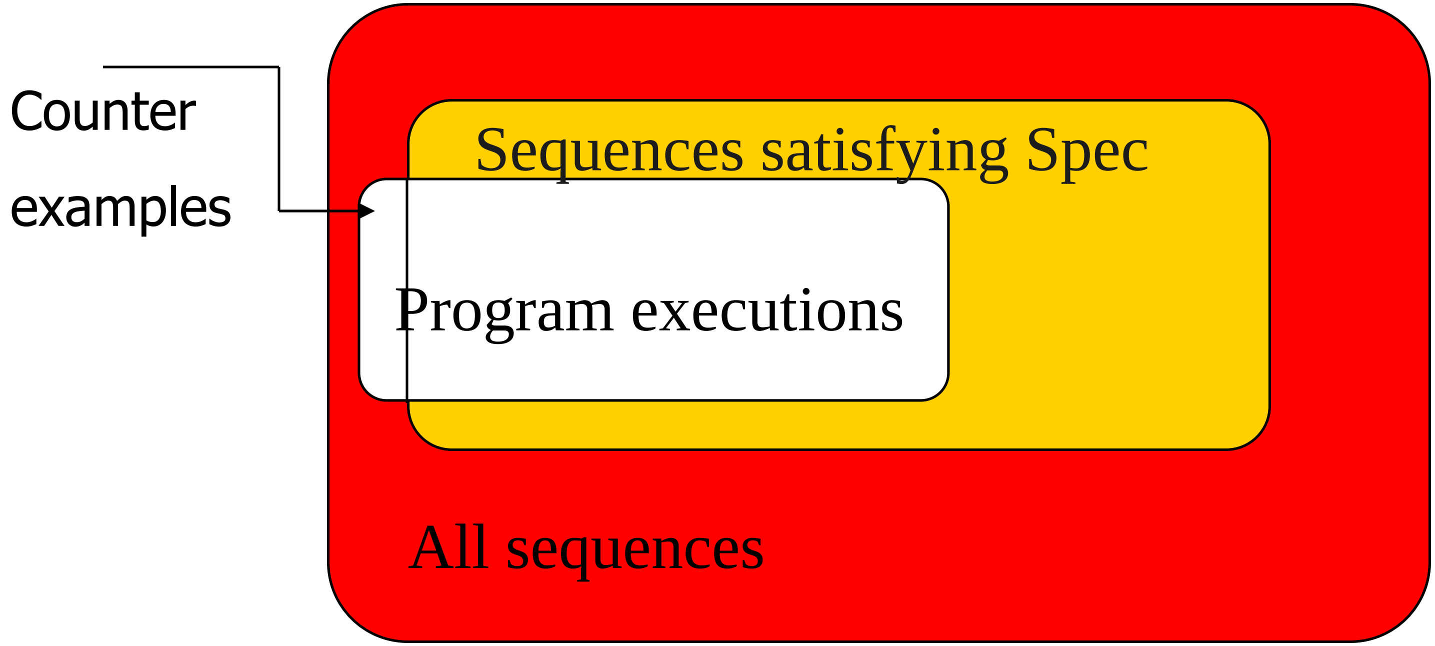
Correctness condition

- We need to define a correctness condition for a model to satisfy a specification.
- Language of a model: L(Model)
- Language of a specification: L(Spec).
- We need: L(Model) $\subseteq$ L(Spec).

Correctness



Incorrectness



Automatic Verification

(Book: Chapter 6)

How can we check the model?

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

What properties can we check?

- Invariant: a property that needs to hold in each 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 for finite state* systems. Would not work with
  - Unconstrained integers.
  - Unbounded message queues.
  - General data structures:

    queues, trees, stacks…
  - parametric algorithms and systems.

---

* But new MC methods make use of decidable logic theories (SMT).

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 usually grows exponentially with the number of processes.

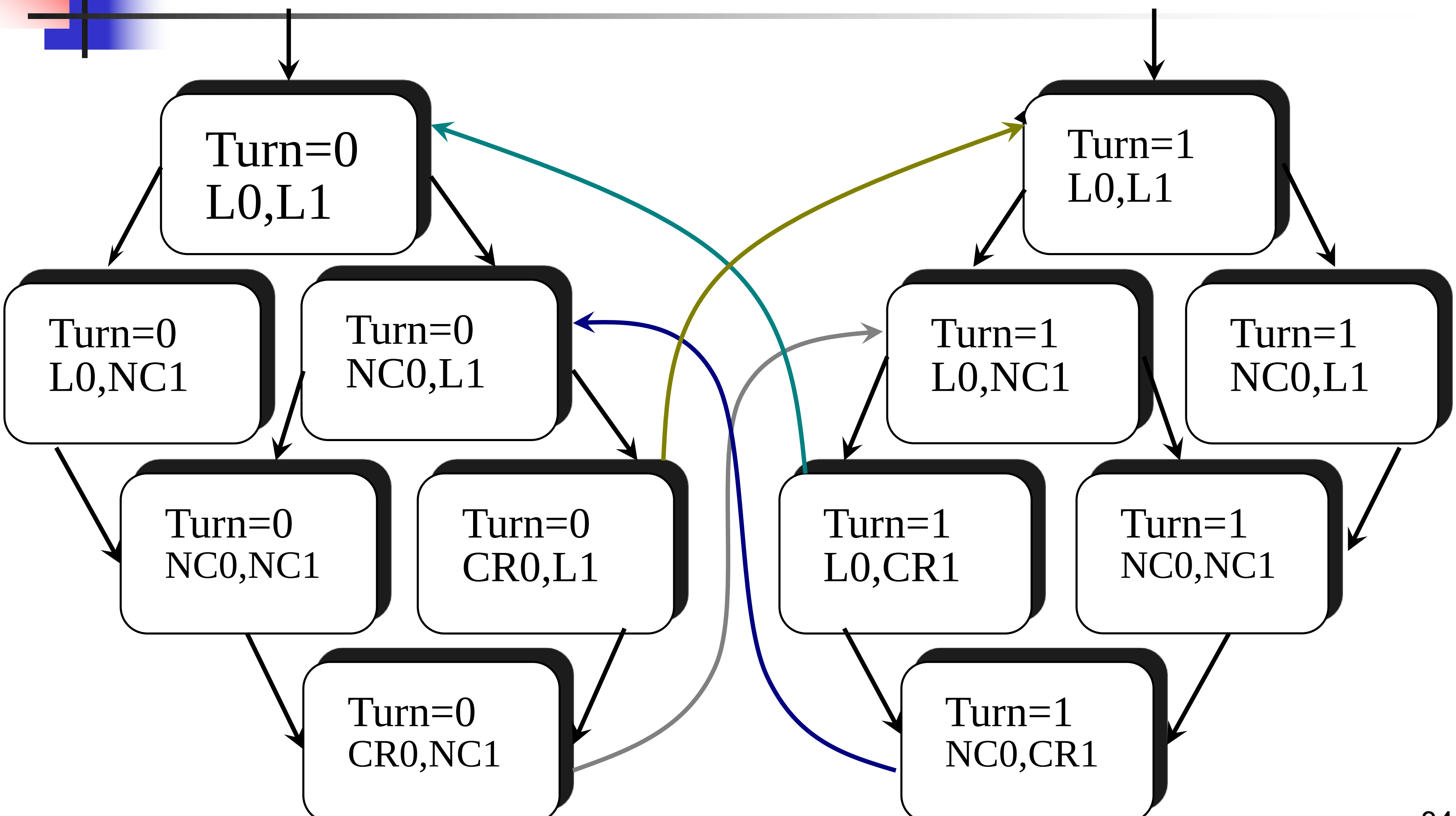
If it is so constrained, is it of any use?

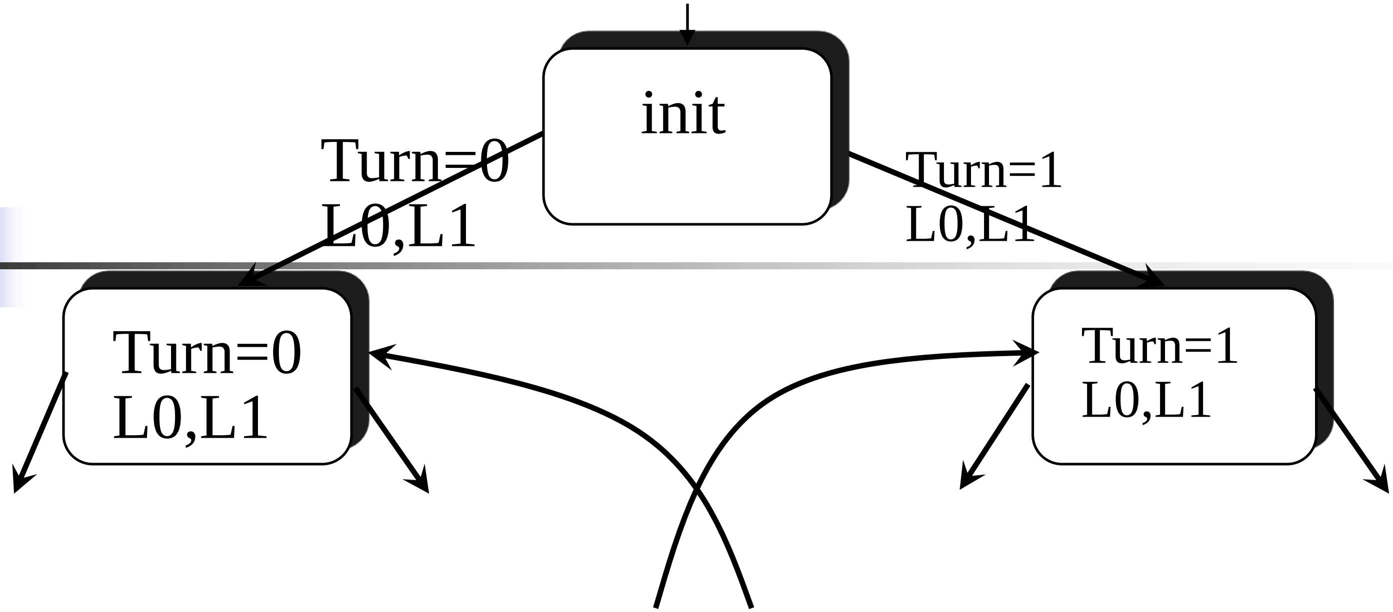
- Many protocols are finite state.
- Many programs or procedure are finite state in nature. Can use abstraction techniques.
- Sometimes it is possible to decompose a program, and prove part of it by model checking and part by theorem proving.
- Many techniques for reducing the state space explosion.

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.

¬(PC0=CR0/\PC1=CR1) is an invariant!





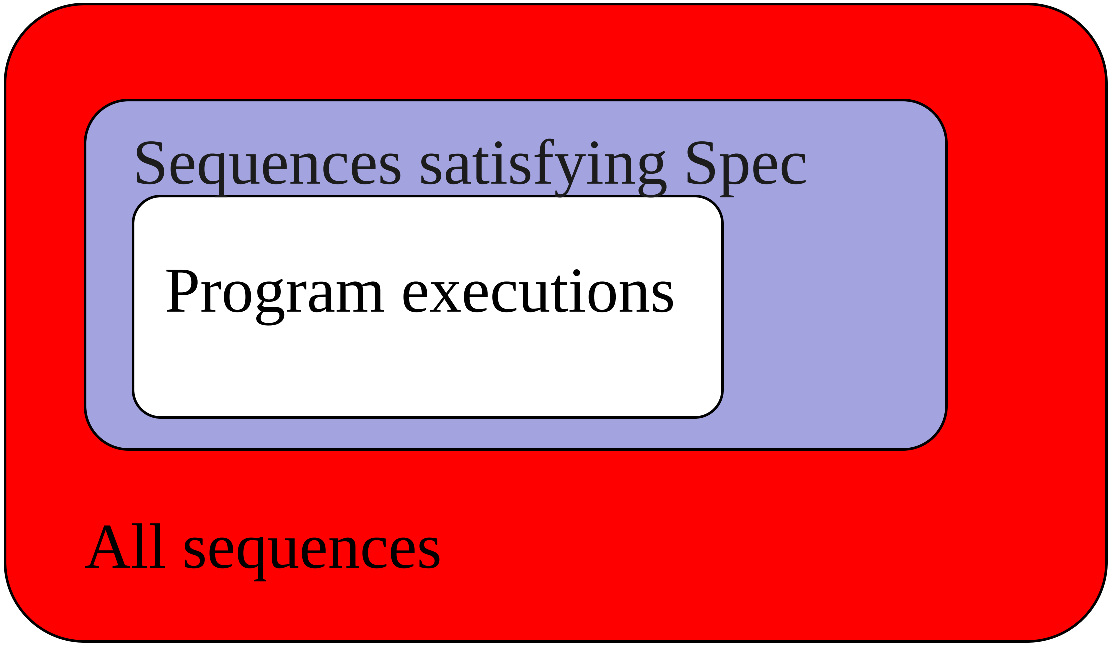
- Propositions are attached to incoming nodes.
- **All nodes are accepting**.

Correctness condition

- We want to find a correctness condition for a model to satisfy a specification.
- Language of a model: L(Model)
- Language of a specification: L(Spec).

- We need: $L(Model) \subseteq L(Spec)$.

Correctness



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!

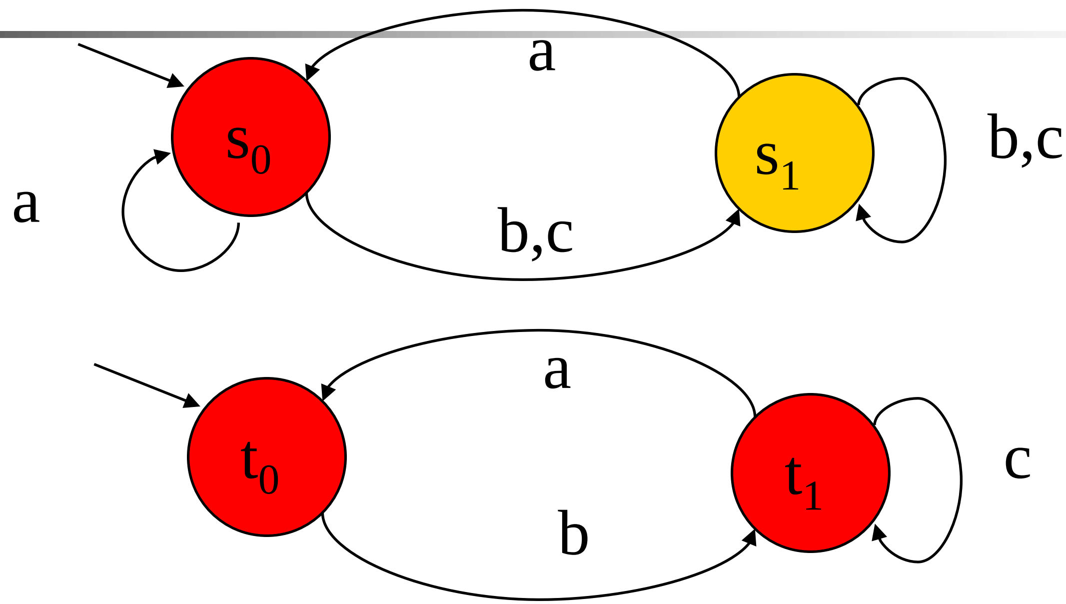
What do we need to know?

- 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 $M_2=(S_2,\Sigma,T_2,I_2,S_2)$

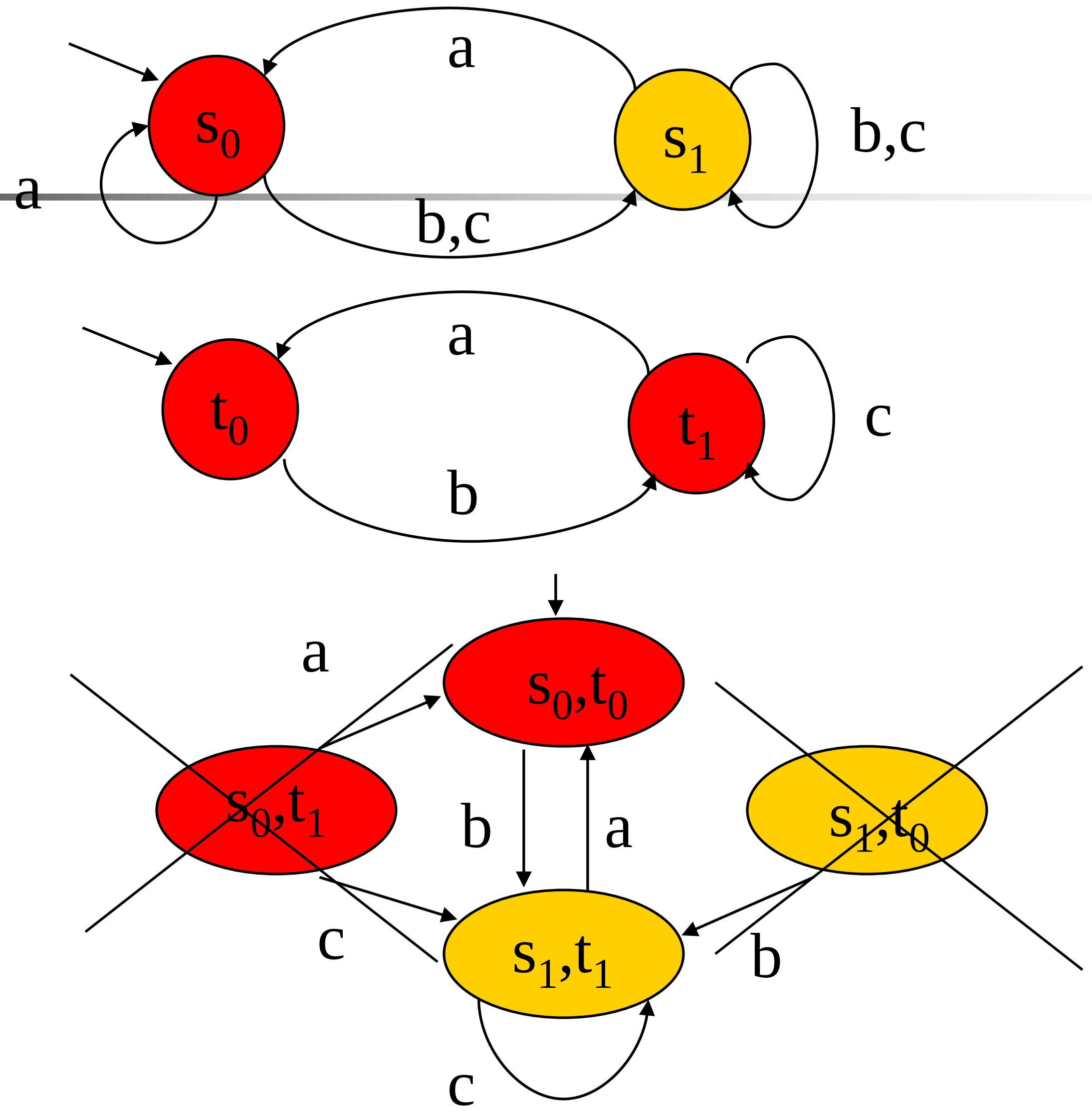
- Run the two automata in parallel.
- Each state is a pair of states: $S_1 \times S_2$
- Initial states are pairs of initials: $I_1 \times I_2$
- Acceptance depends on first component: $A_1 \times S_2$
- Conforms with transition relation: (x_1,y_1)-a->(x_2,y_2) when x_1-a->x_2 and y_1-a->y_2.

Example (all states of second automaton accepting!)



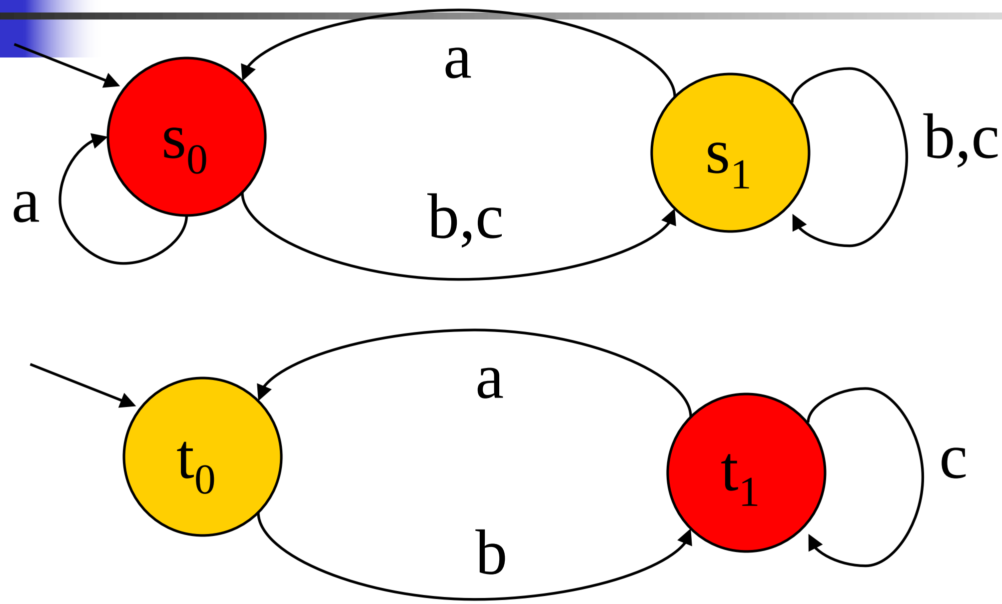
States: (s_0,t_0), (s_0,t_1), (s_1,t_0), (s_1,t_1).

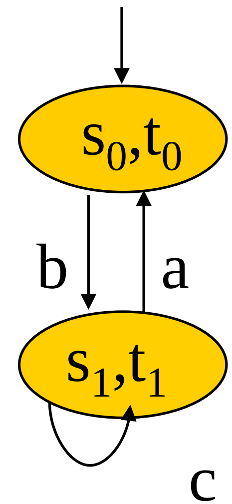
Accepting: (s_0,t_0), (s_0,t_1). Initial: (s_0,t_0).



a
s_0
s_1
b,c
a
b,c
a
t_0
t_1
c
b
a
s_0,t_0
s_0,t_1
b
a
s_1,t_0
c
s_1,t_1
b
c

More complicated when $A_2 \neq S_2$

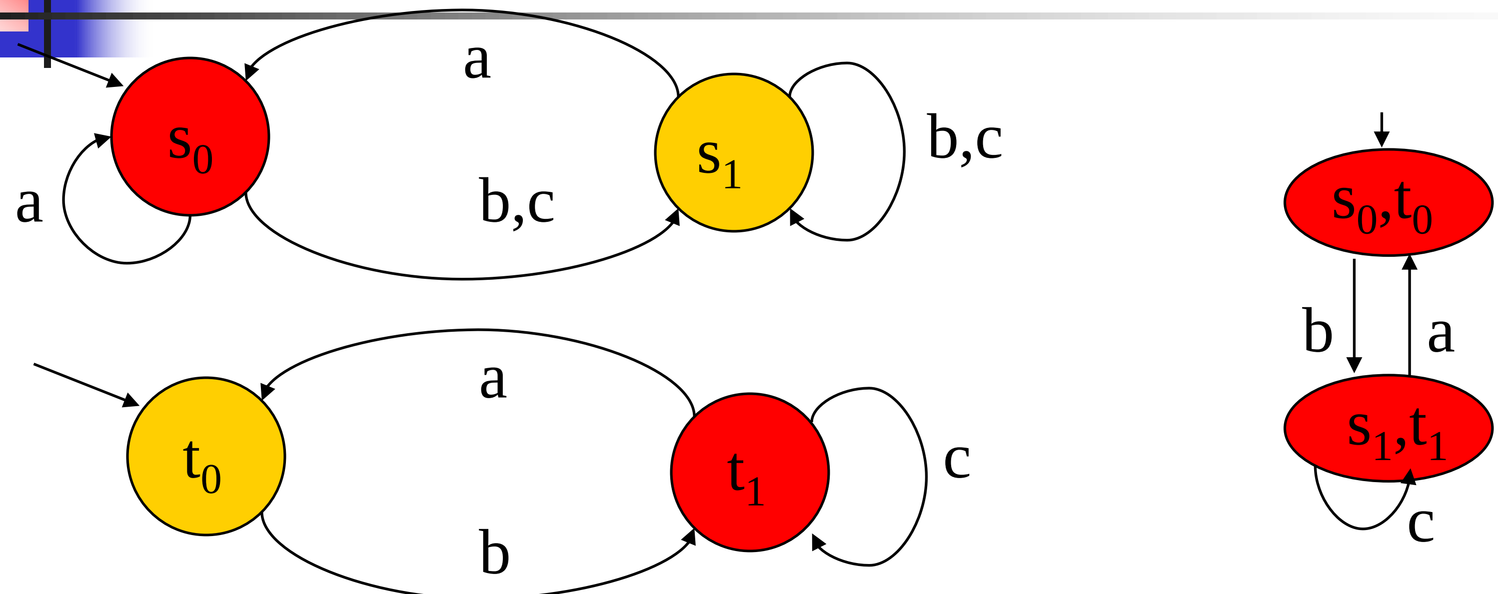




Should we have acceptance when both components accepting? I.e., $\{(s_0,t_1)\}$?

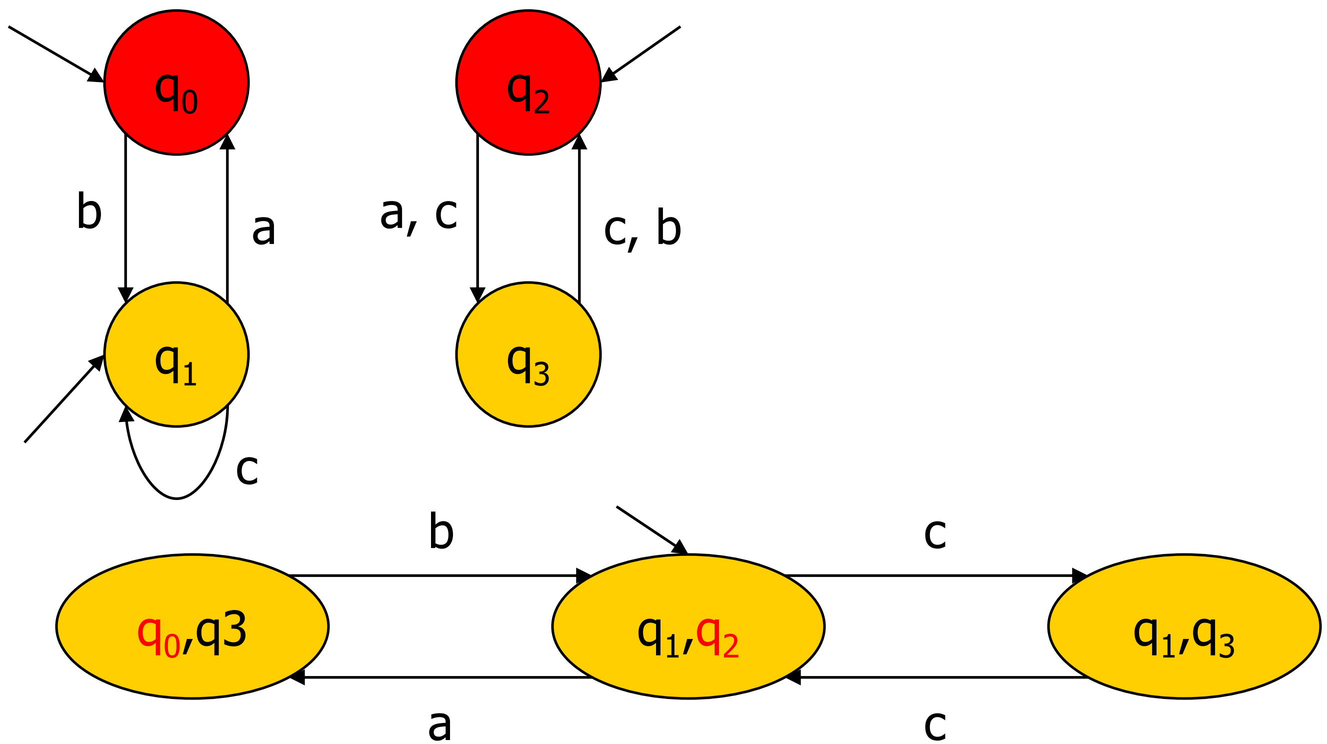
No, consider $(ba)^\omega$
It should be accepted, but never passes that state.

More complicated when $A_2 \neq S_2$

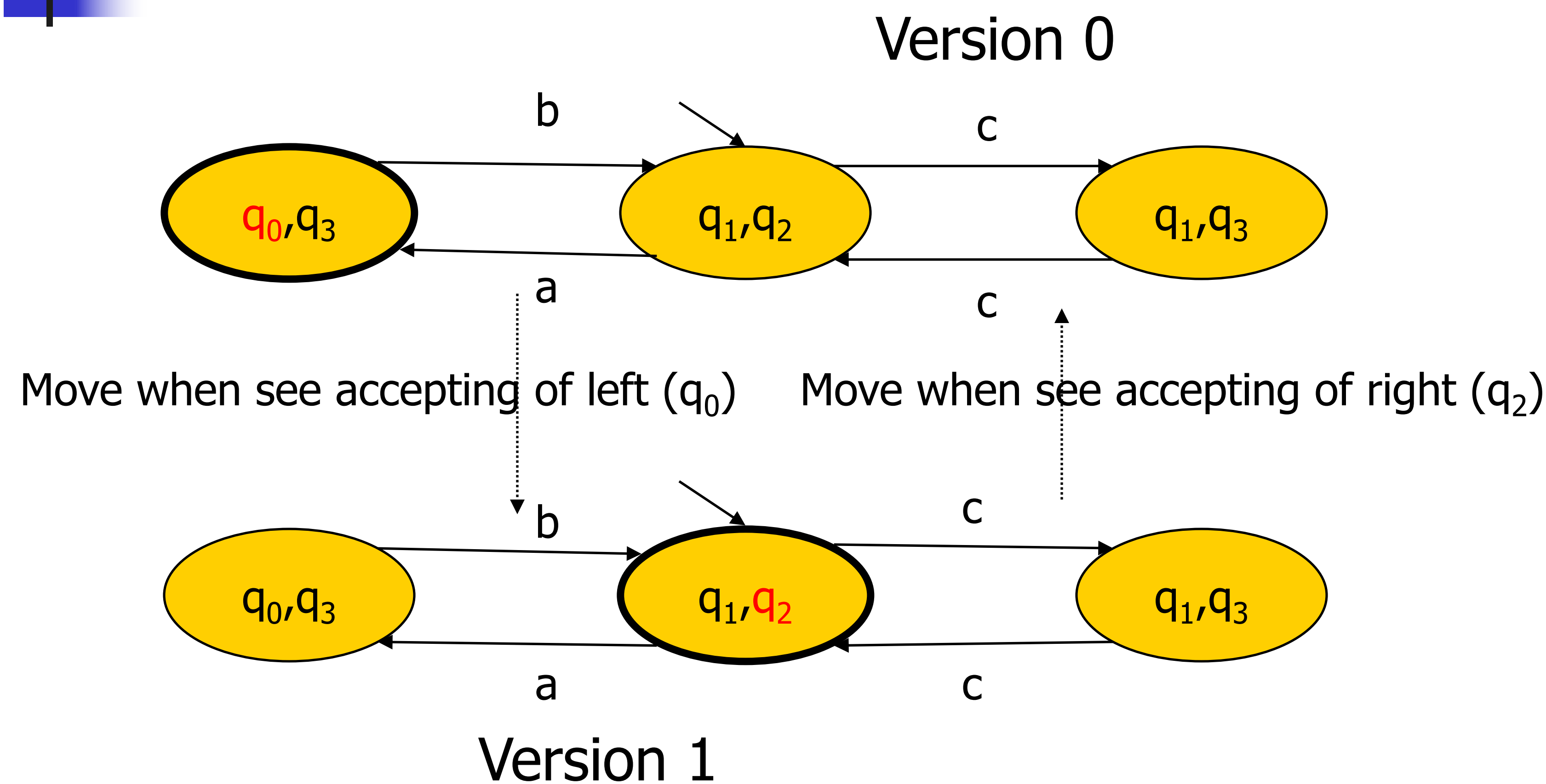


Should we have acceptance when at least one components is accepting? I.e., $\{(s_0,t_0),(s_0,t_1),(s_1,t_1)\}$?
No, consider b c^ω
It should not be accepted, but here will loop through (s_1,t_1)

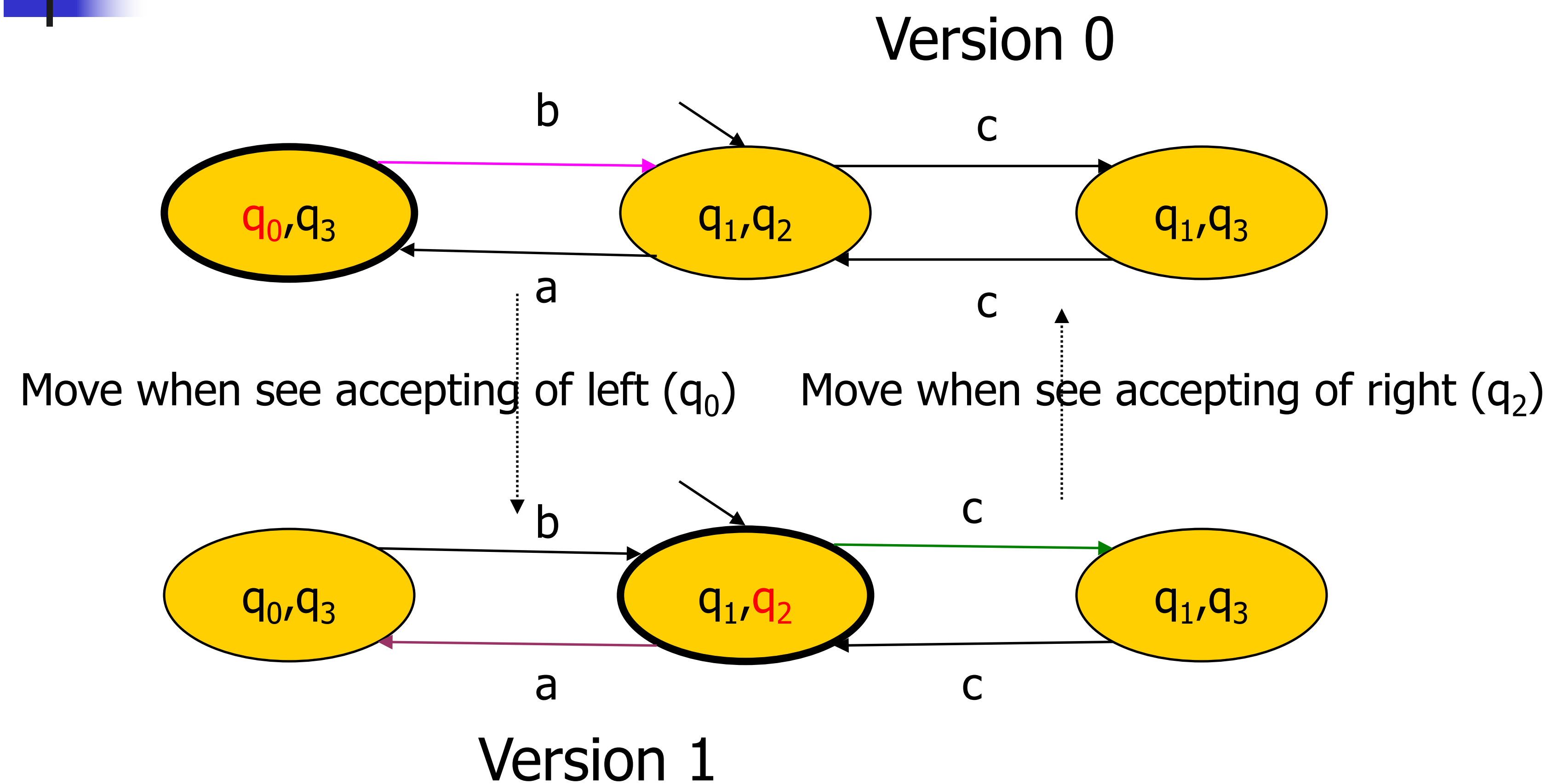
Intersection - general case



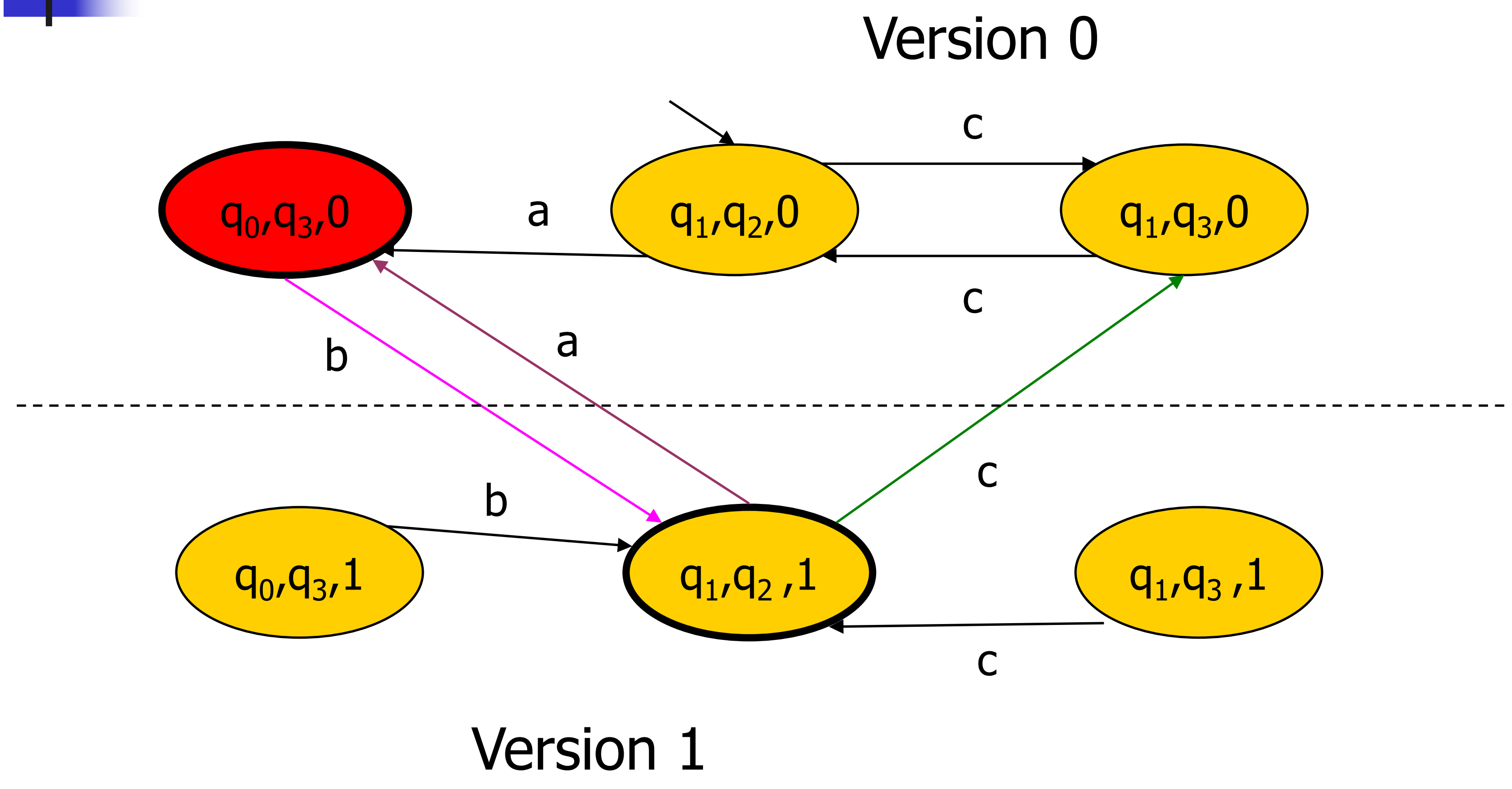
Version 0: to catch accepting state q_0
Version 1: to catch accepting state q_2



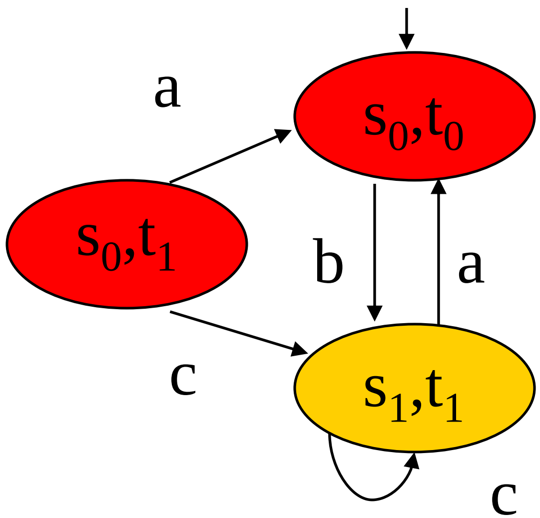
Version 0: to catch accepting state q_0
Version 1: to catch accepting state q_2



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

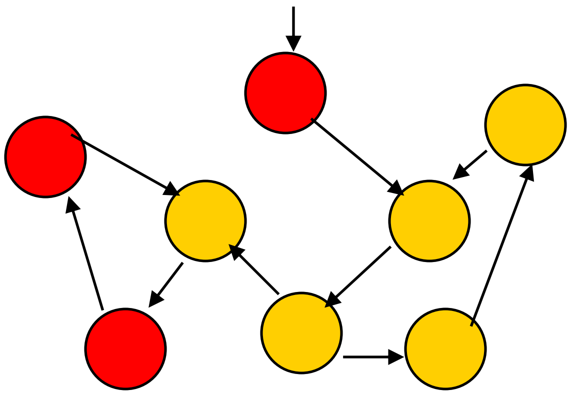


How to check for emptiness?



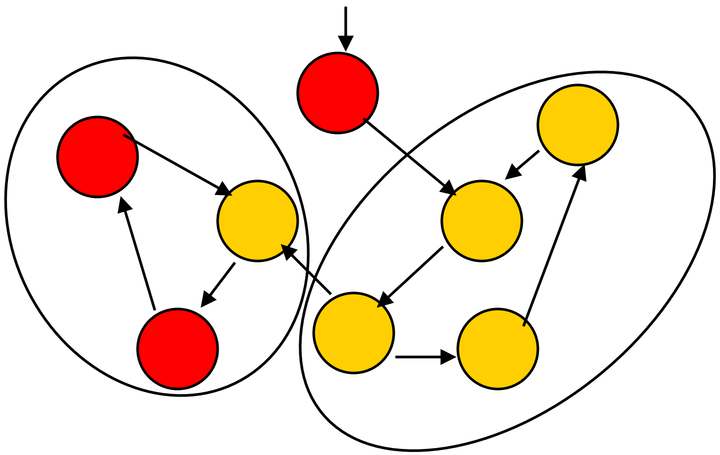
Emptiness...

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.

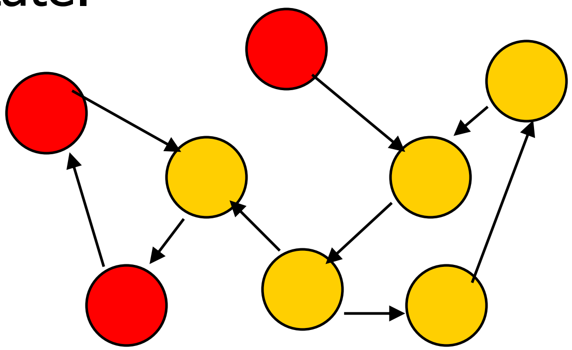
Finding accepting runs

If there is an accepting run, then at least one accepting state repeats on it forever.

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

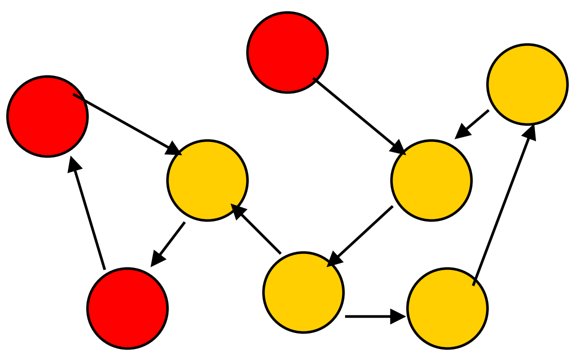
These states form a strongly connected component on the automaton graph, including an accepting state.

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



Equivalently…

- A strongly connected component: a set of nodes where each node is reachable by a path from each other node. Find a reachable strongly connected component with an accepting node.



How to complement?

- Complementation is hard!
- Can ask for the negated property (the sequences that should never occur).
- Can translate from LTL formula φ to automaton A, and complement A. But: can translate $\neg\varphi$ into an automaton directly!

Translating from logic to automata

(Book: Chapter 6)

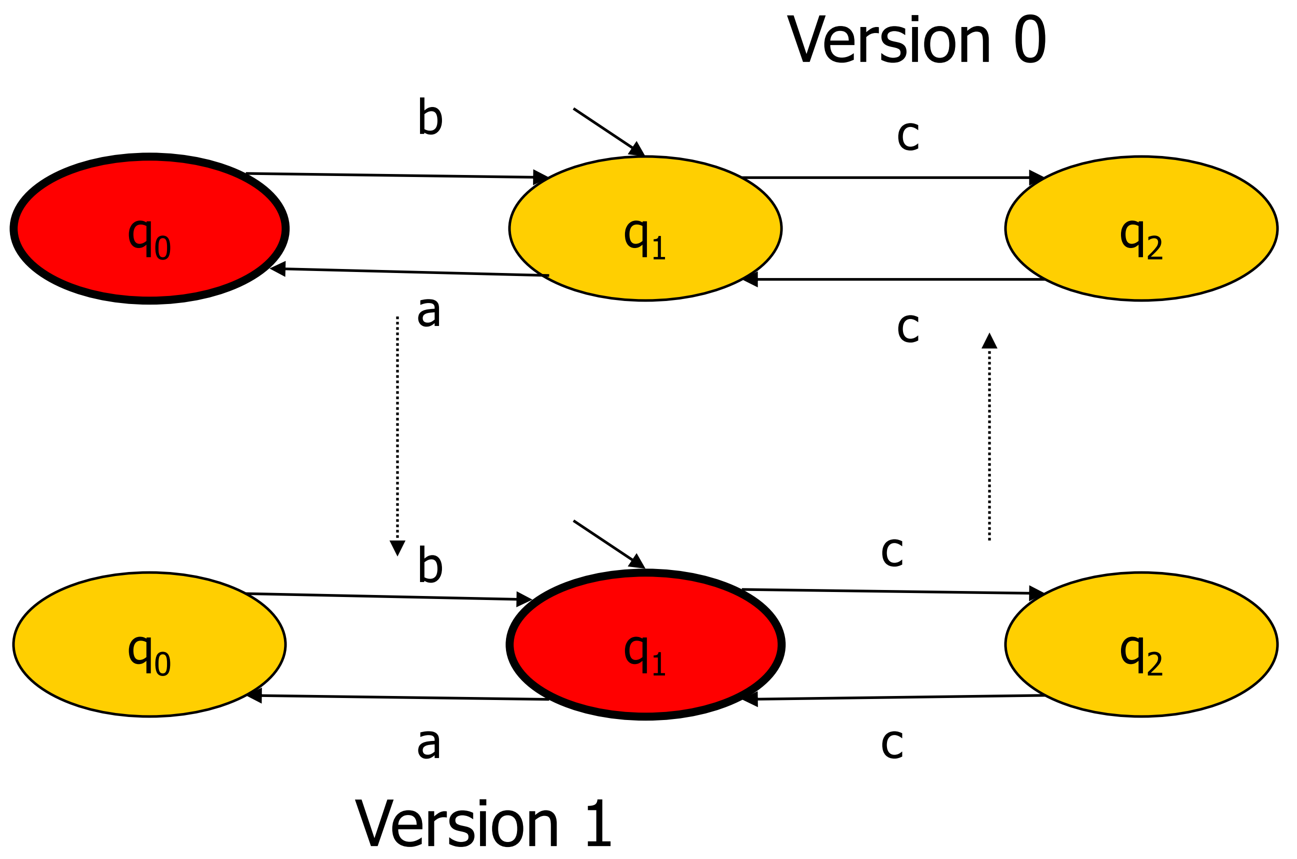
Why translating?

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

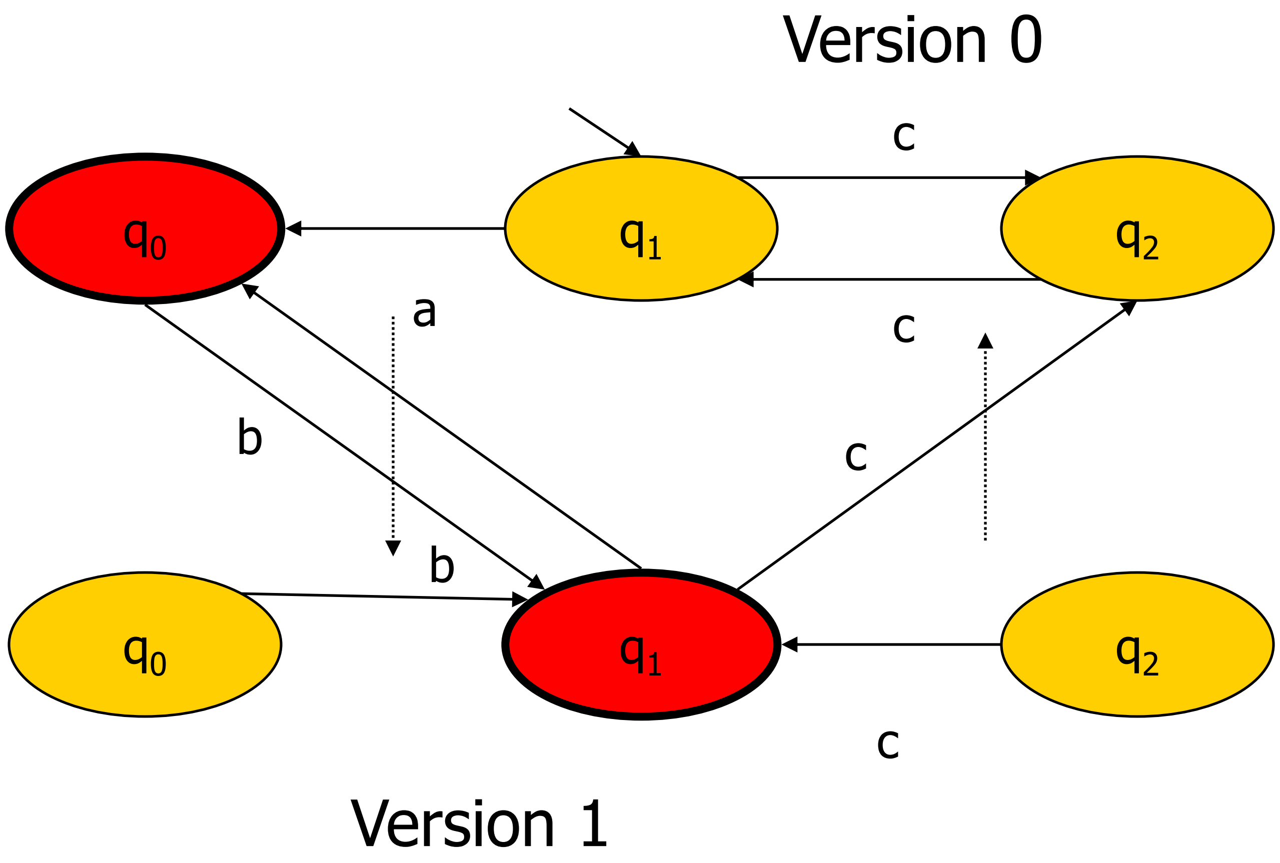
Generalized Büchi automata

- Acceptance condition F is a set $F=\{f_1, f_2, \ldots, f_n\}$ 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.

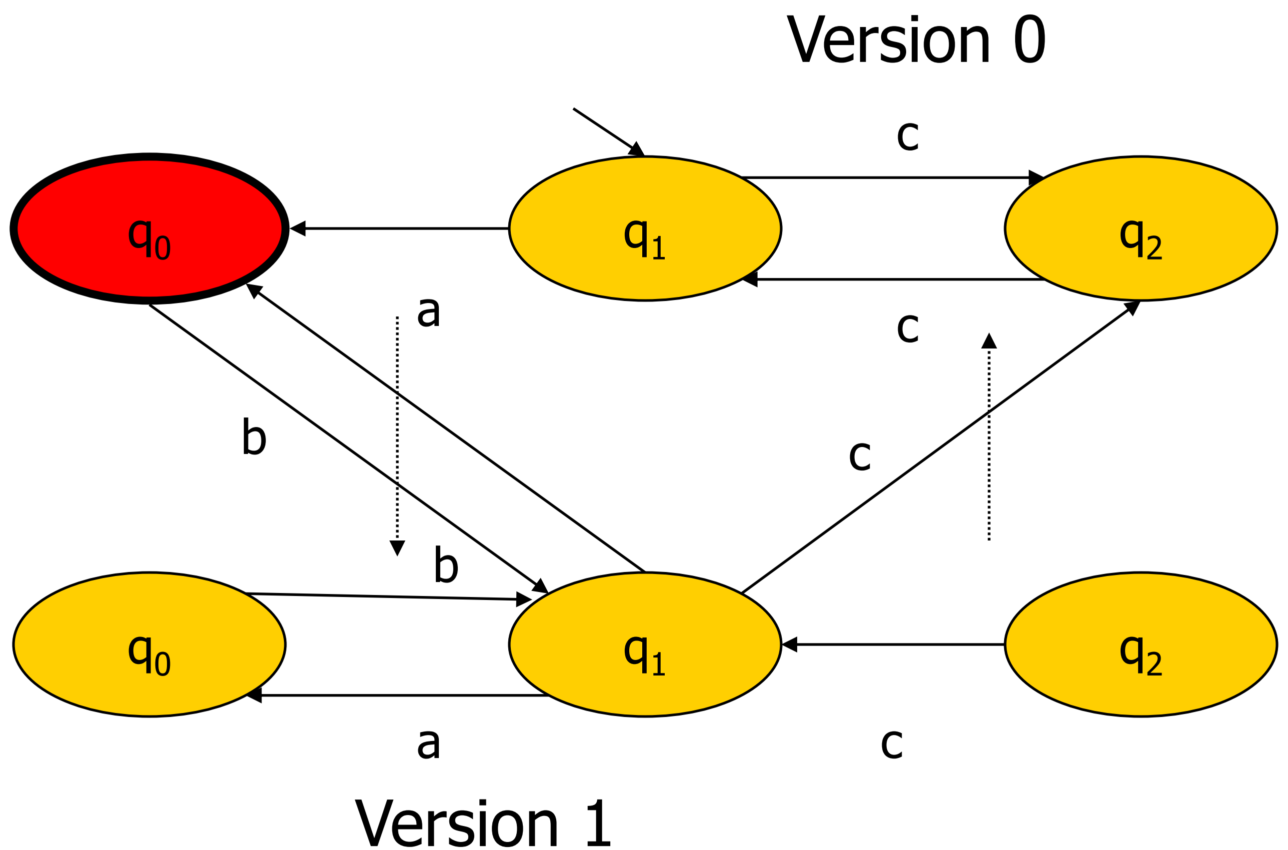
Translating into simple Büchi automaton



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 $<>\neg\varphi$.
- $\neg<>\varphi$ becomes $[]\neg\varphi$.
- What about $\neg(\varphi \mathrm{U} \psi)$?
- Define operator *R* such that
  $\neg(\varphi\, U\, \psi) = (\neg\varphi)\, R\, (\neg\psi)$,
  $\neg(\varphi\, R\, \psi) = (\neg\varphi)\, U\, (\neg\psi)$.

Semantics of p *R* q

¬p	¬p	¬p	¬p	¬p	¬p	¬p	¬p	¬p
q	q	q	q	q	q	q	q	q

¬p	¬p	¬p	¬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 $(\neg\varphi) \vee (\neg\psi)$

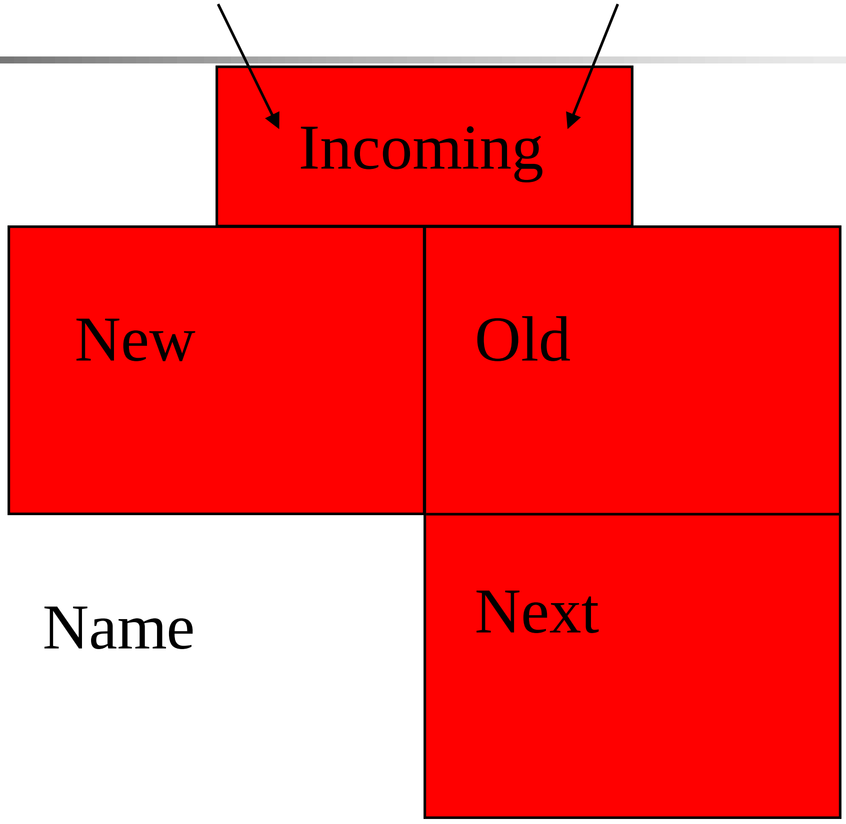
Eliminate implications, <>, []

- Replace φ -> ψ by (¬ φ) ⋁ ψ.
- Replace <>φ by (true *U* φ).
- Replace []φ by (false *R* φ).

Example

- Translate $([]<>P) \rightarrow ([]<>Q)$
- Eliminate implication $\neg([]<>P) \vee ([]<>Q)$
- Eliminate $[]$, $<>$:
  $\neg(\text{false } R \,(\text{true } U\, P)) \vee (\text{false } R\, (\text{true } U\, Q))$
- Push negation inwards:
  $(\text{true } U\, (\text{false } R\, \neg P)) \vee (\text{false } R\, (\text{true } U\, Q))$

The data structure



The main idea

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

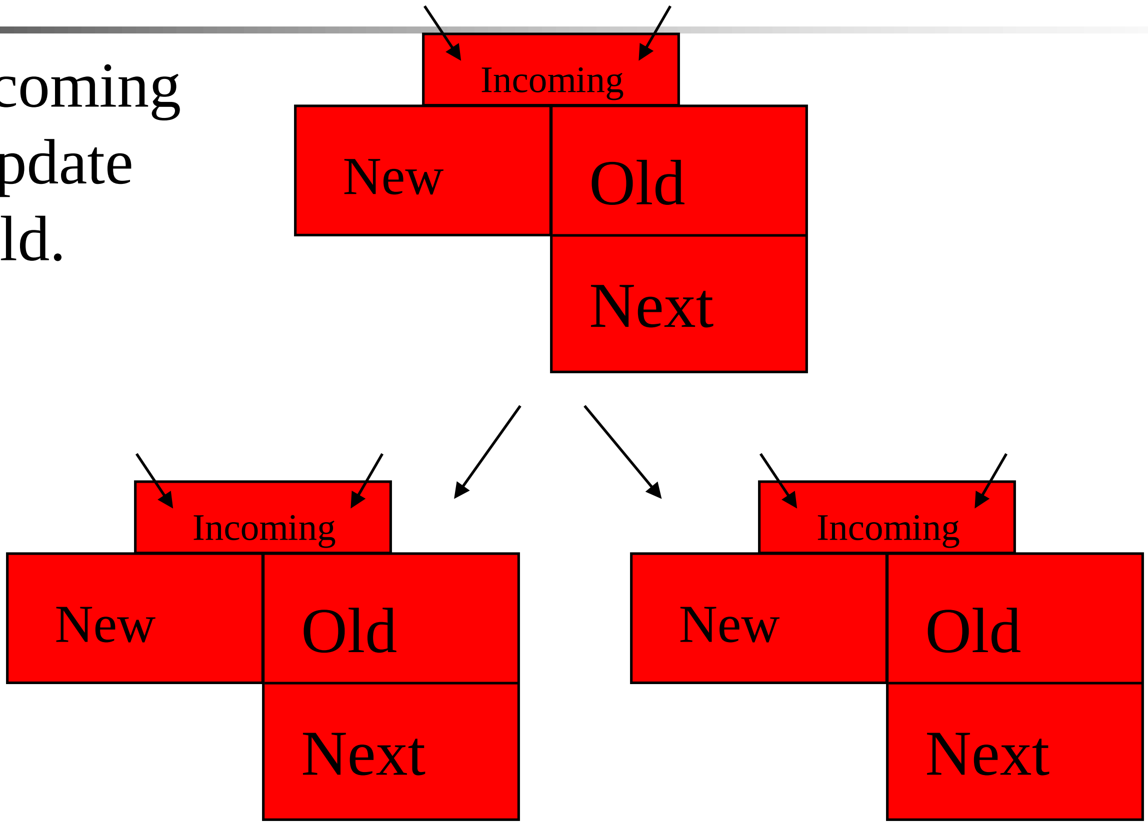
This separates the formulas into two parts: one holds in the current state, and the other in the next state.

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*** characteristics), or
  - Evolve the node into a new version (***and*** characteristics).

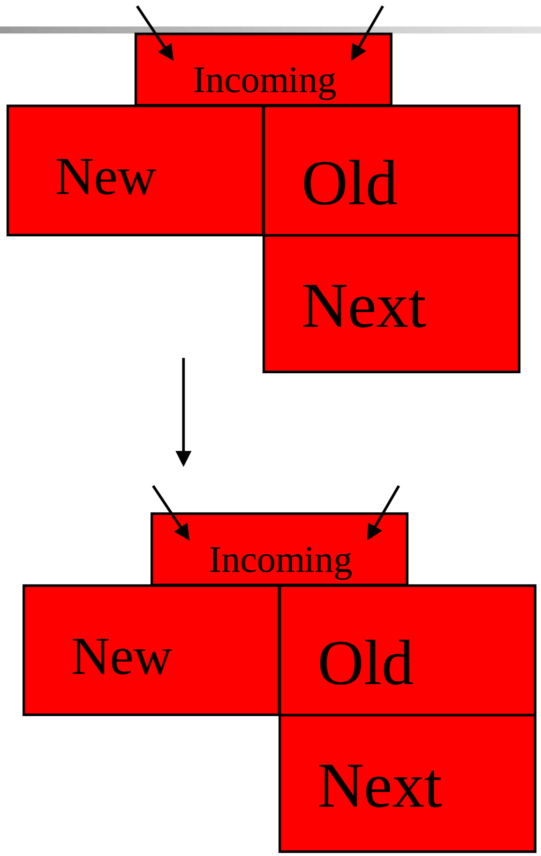
Splitting

Copy incoming edges, update other field.



Evolving

Copy incoming edges, update other field.



Possible cases:

- $\varphi \bigvee \psi$, split:
  - Add φ to New.
  - Add ψ to New.
- φ /\ ψ, evolve:
  - Add φ, ψ to New.
- O φ, evolve:
  - Add φ to Next.

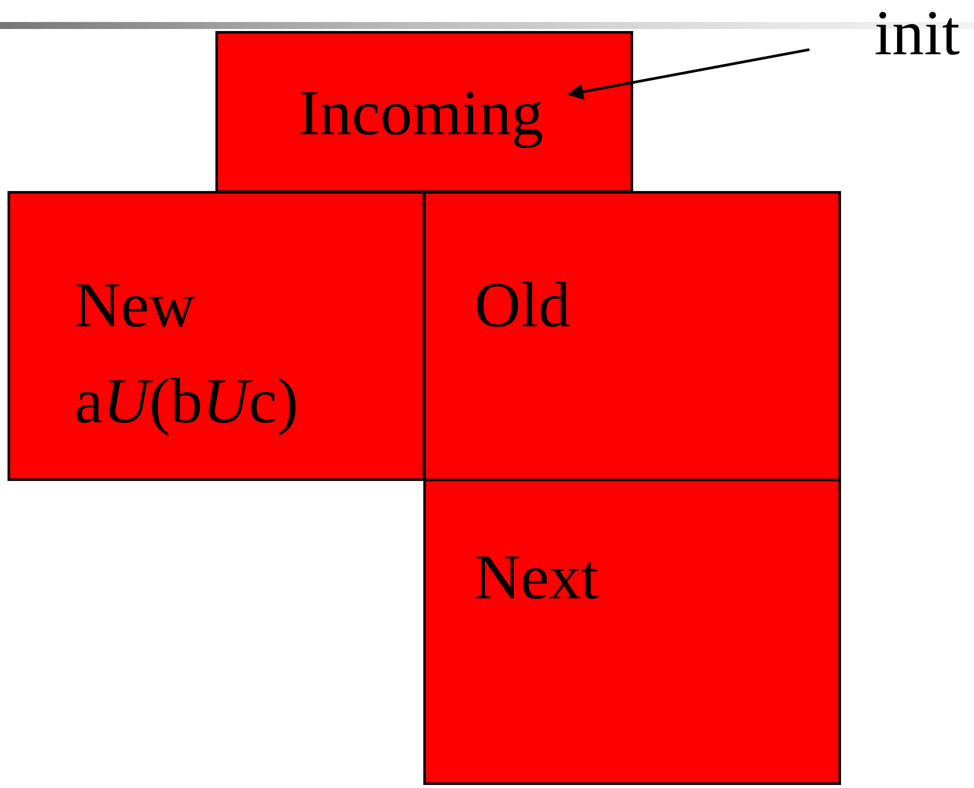
More cases:

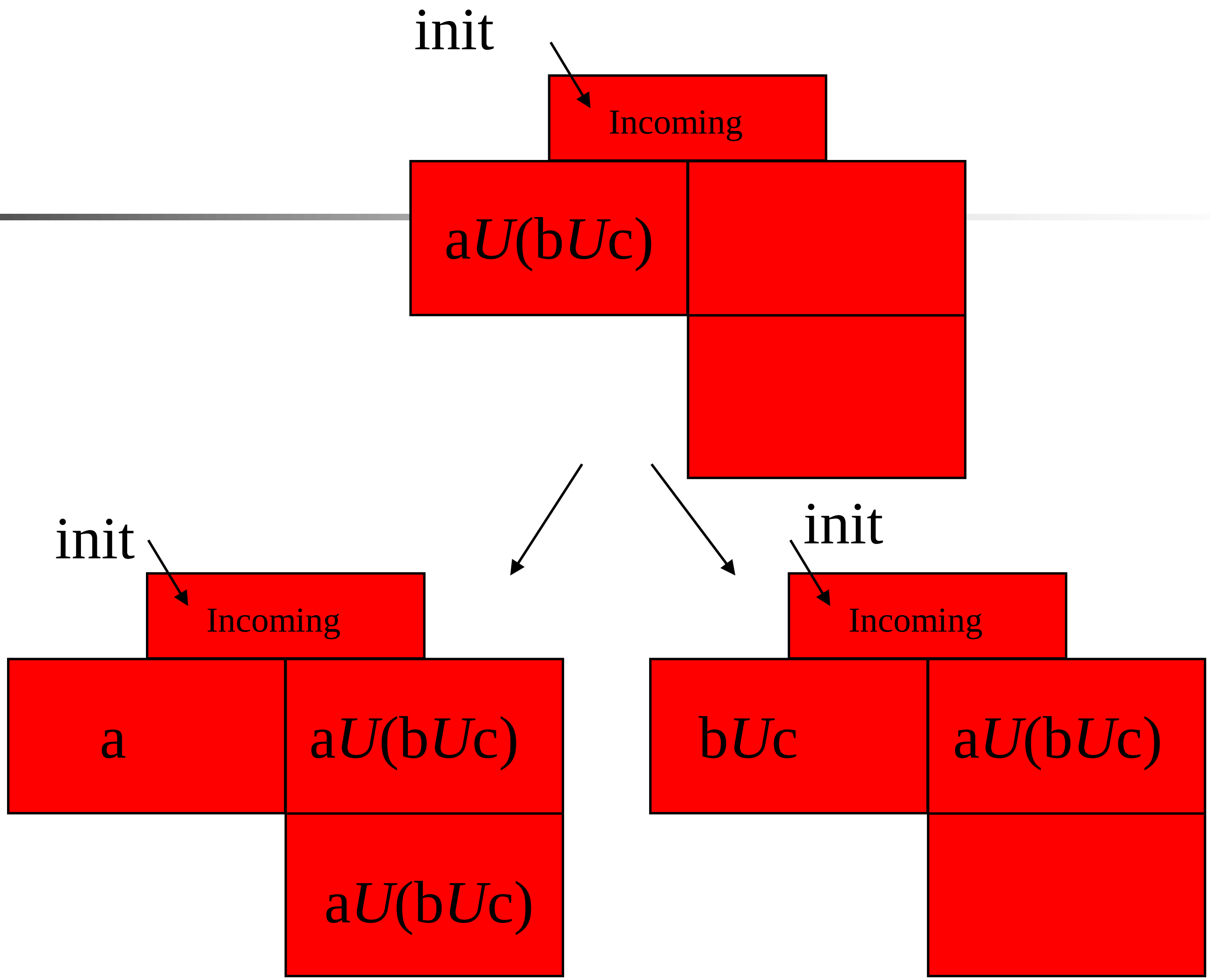
- $\varphi\ U\ \psi$, split:
  - Add φ to New, add $\varphi U \psi$ to Next.
  - Add ψ to New.

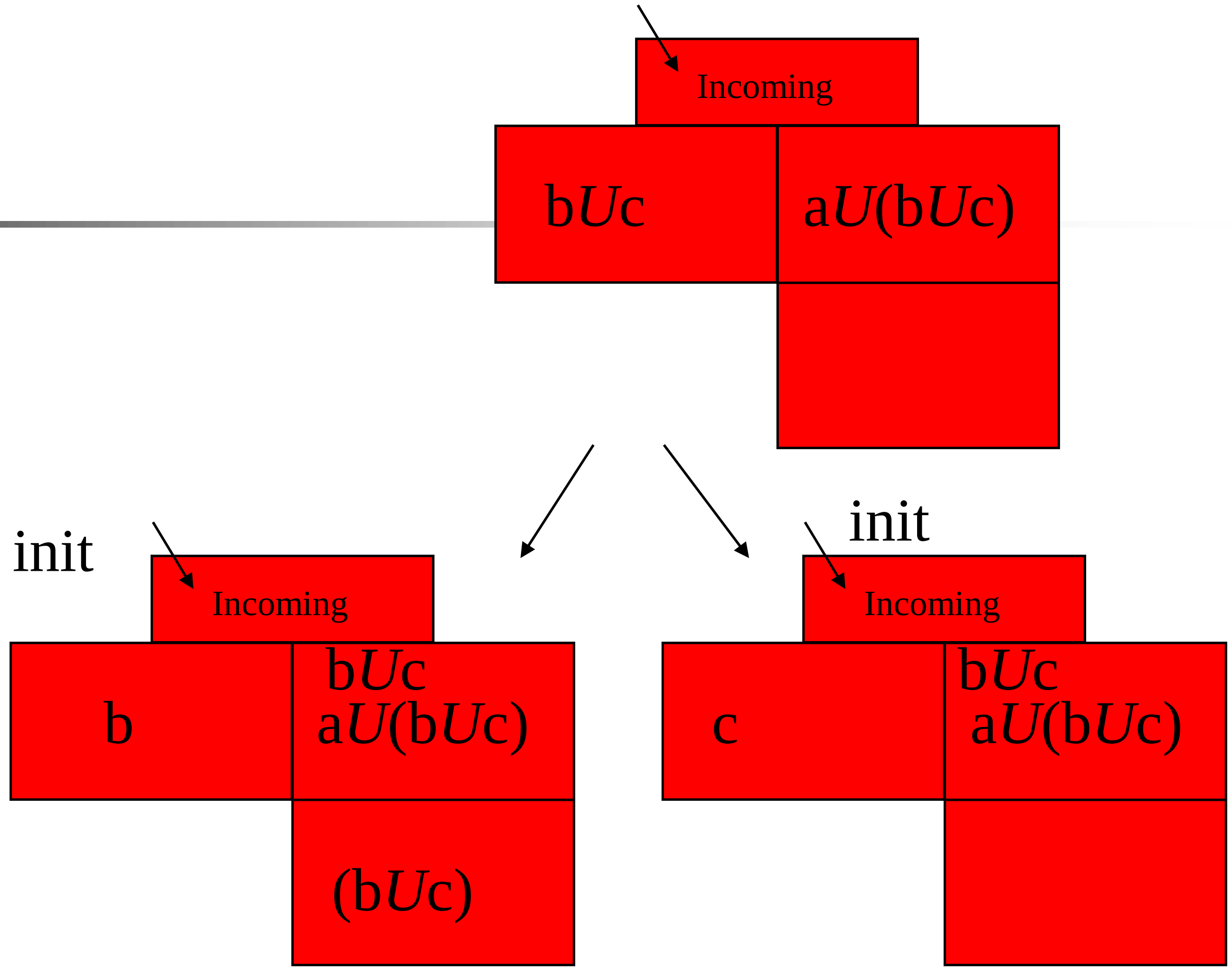
  [Because $\varphi U \psi = \psi \lor (\varphi \land O\ (\varphi U \psi))$.]
- $\varphi\ R\ \psi$, split:
  - Add φ, ψ to New.
  - Add ψ to New, $\varphi\ R\ \psi$ to Next.

  [Because $\varphi\ R\ \psi = \psi \land (\varphi \lor O\ (\varphi\ R\ \psi))= (\psi \land \varphi) \lor (\psi \land O\ (\varphi\ R\ \psi))$.]

How to start?




init
Incoming
aU(bUc)
init
Incoming
a
aU(bUc)
aU(bUc)
init
Incoming
bUc
aU(bUc)


Incoming
bUc
aU(bUc)
init
Incoming
b
bUc
aU(bUc)
(bUc)
init
Incoming
c
bUc
aU(bUc)

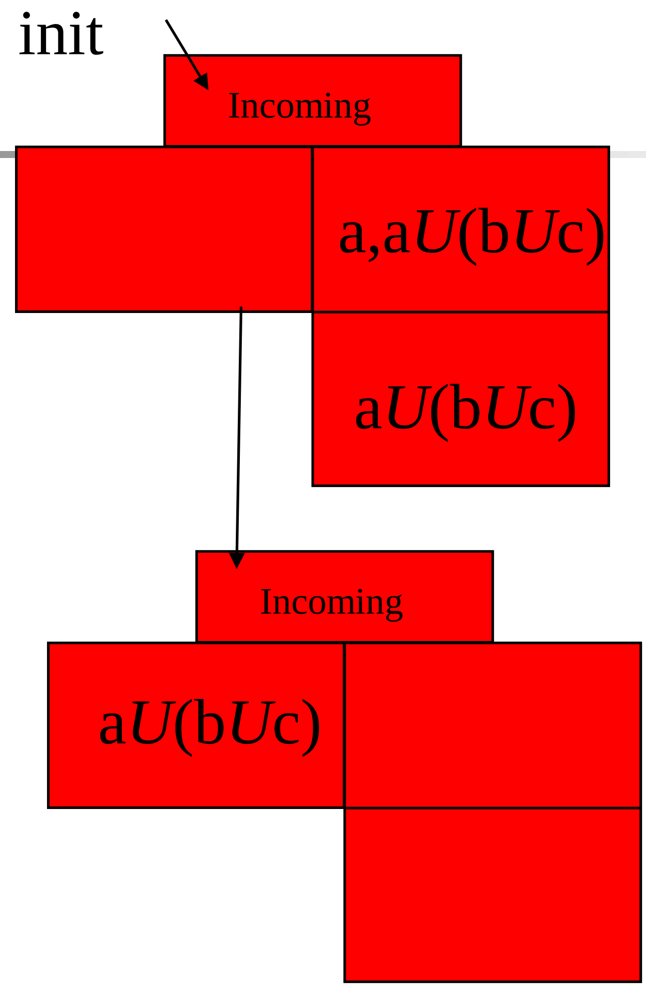
When to stop splitting?

- 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.

When a node is added to "Nodes"…

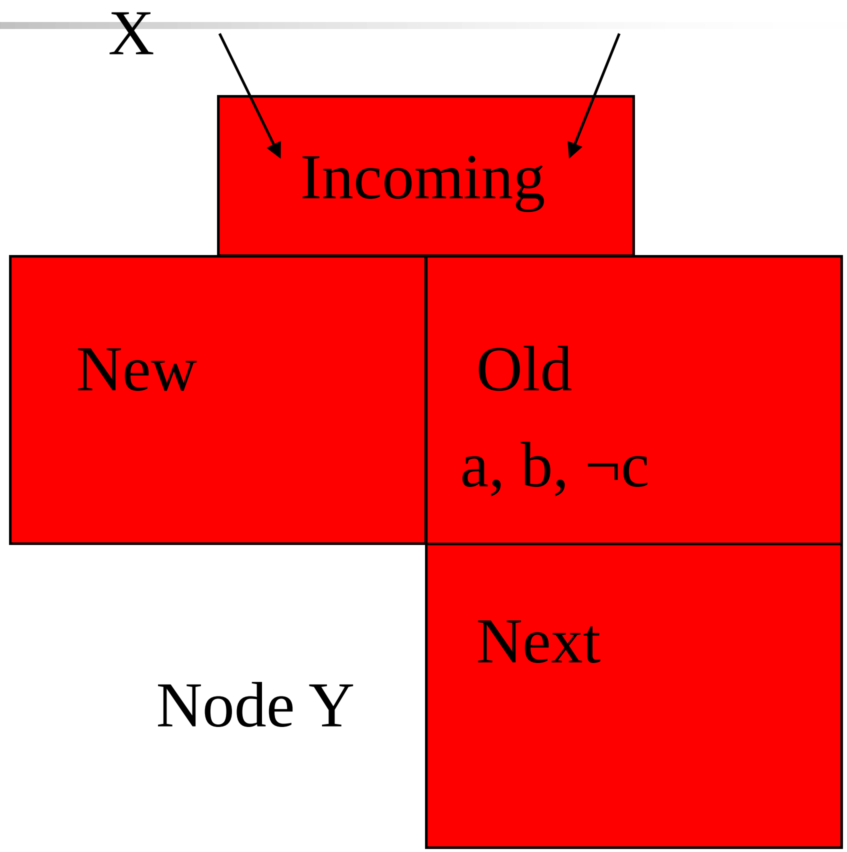
Copy Next field to New field of the successor, and making an edge to the new successor.

Start evolving/splitting successor

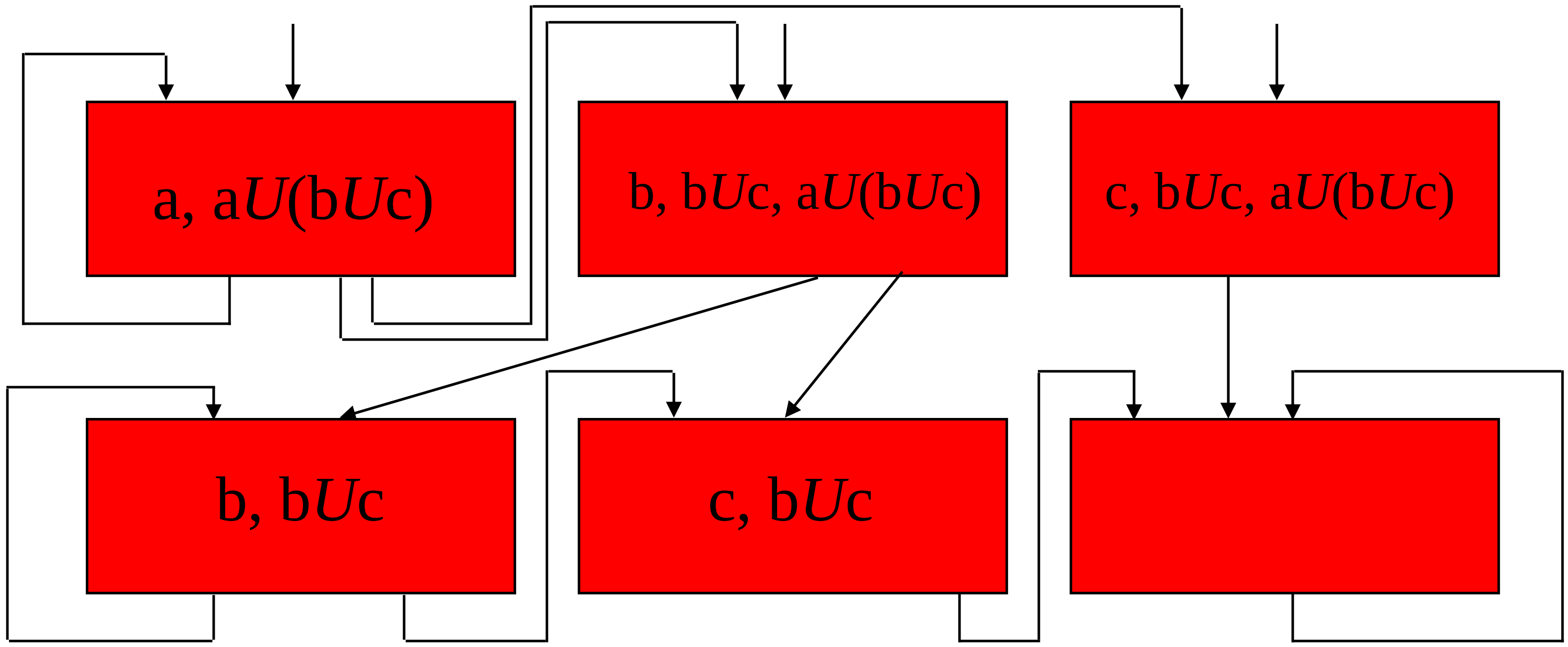


When there are no pending nodes/successors to process

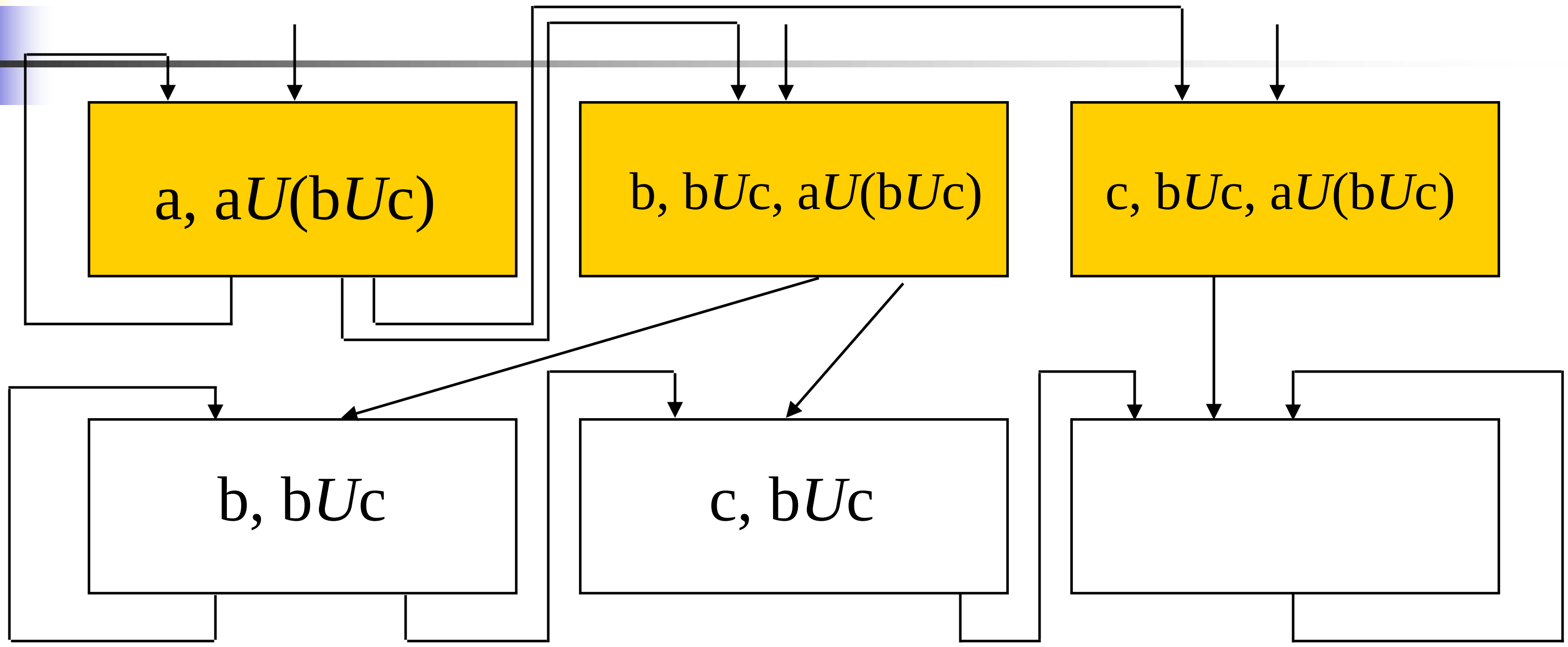
- Each node in "Nodes" become a state in the automaton. It is labeled by the propositions/negated propositions in the "old" field.
- Successor relationship according to the "incoming" field.



The resulted nodes.



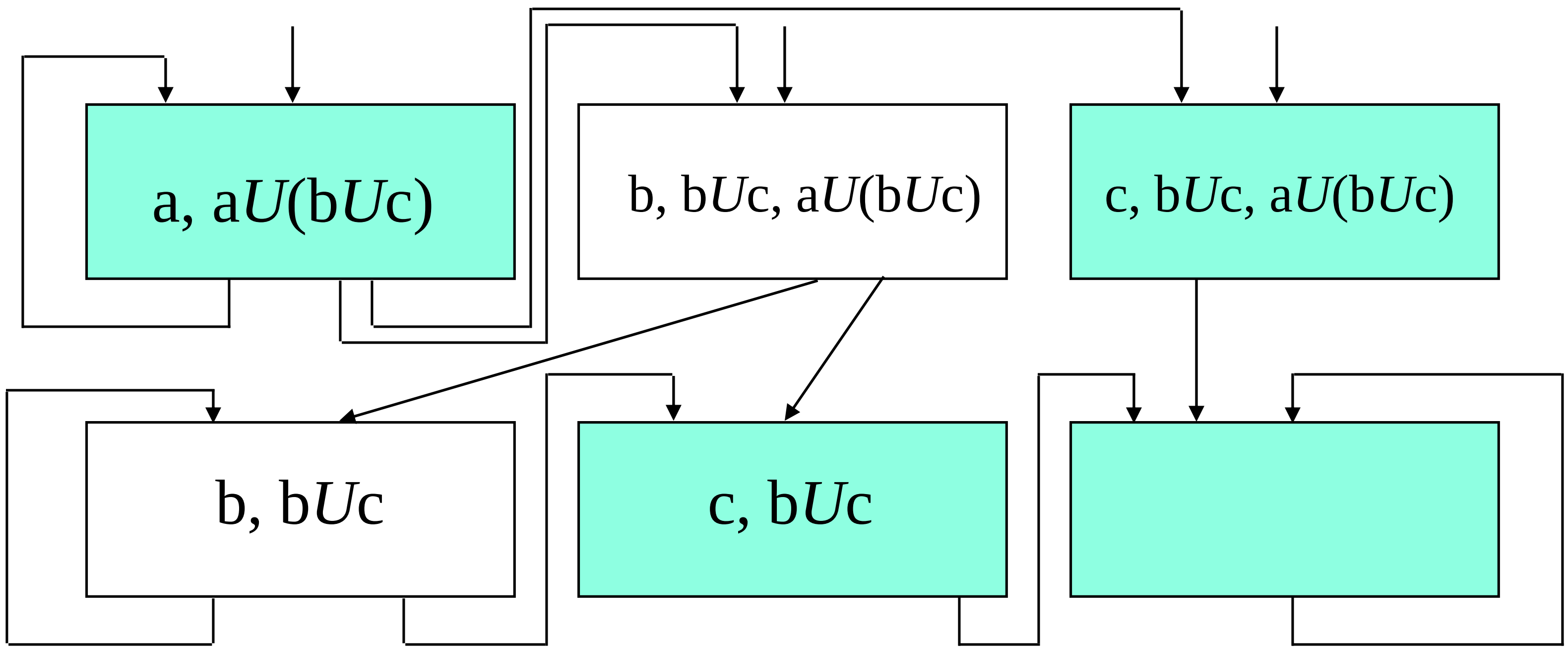
Initial nodes: those with “init” edge in “incoming”



Acceptance conditions: guaranteeing that for each subformula $\varphi U \psi$, ψ eventually holds

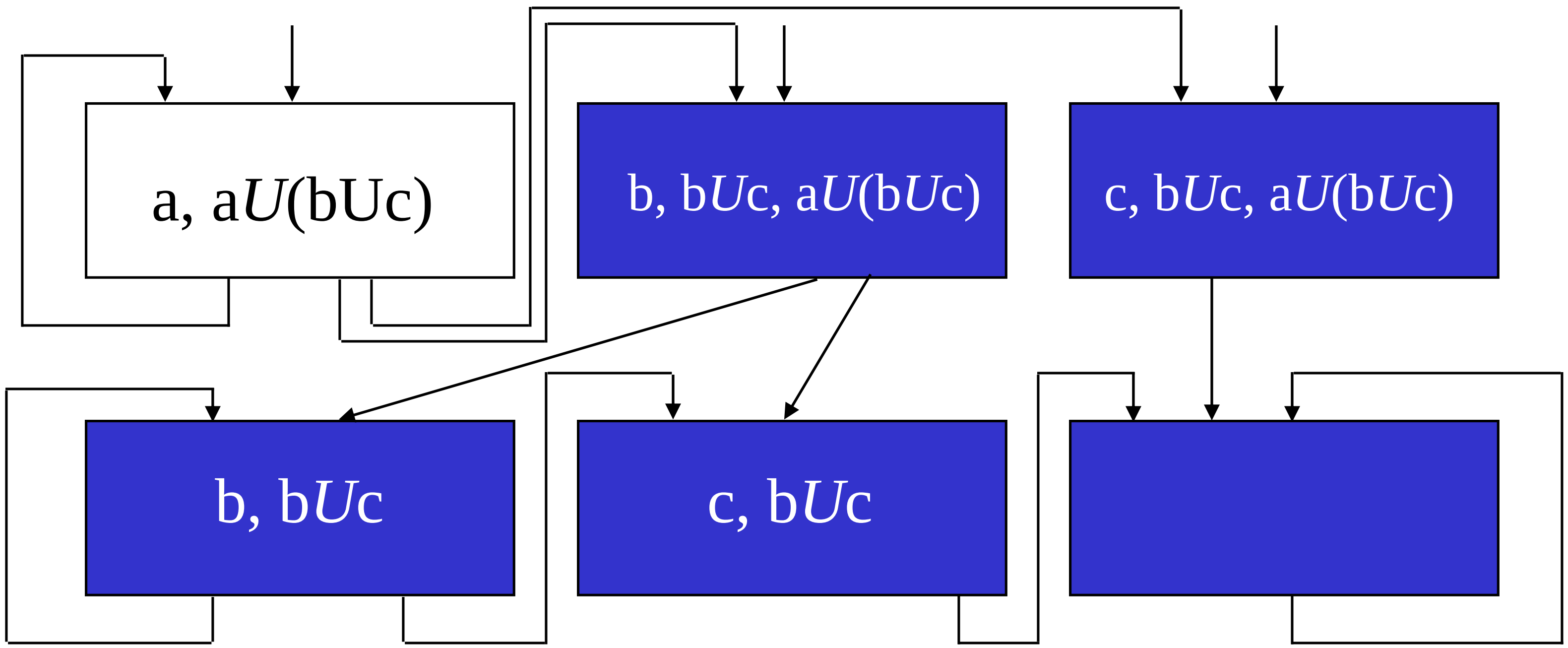
- The successor relation only guarantees that either ψ holds now, or is delayed.
- Use "generalized Buchi automata", where there are several acceptance sets $f_1, f_2, \ldots, f_n$, and each accepted infinite sequence must include at least one state from each set infinitely often.
- Each set corresponds to a subformula of form $\varphi U \psi$. Guarantees that it is never the case that ψ is delayed forever.

Accepting w.r.t. b*U* c



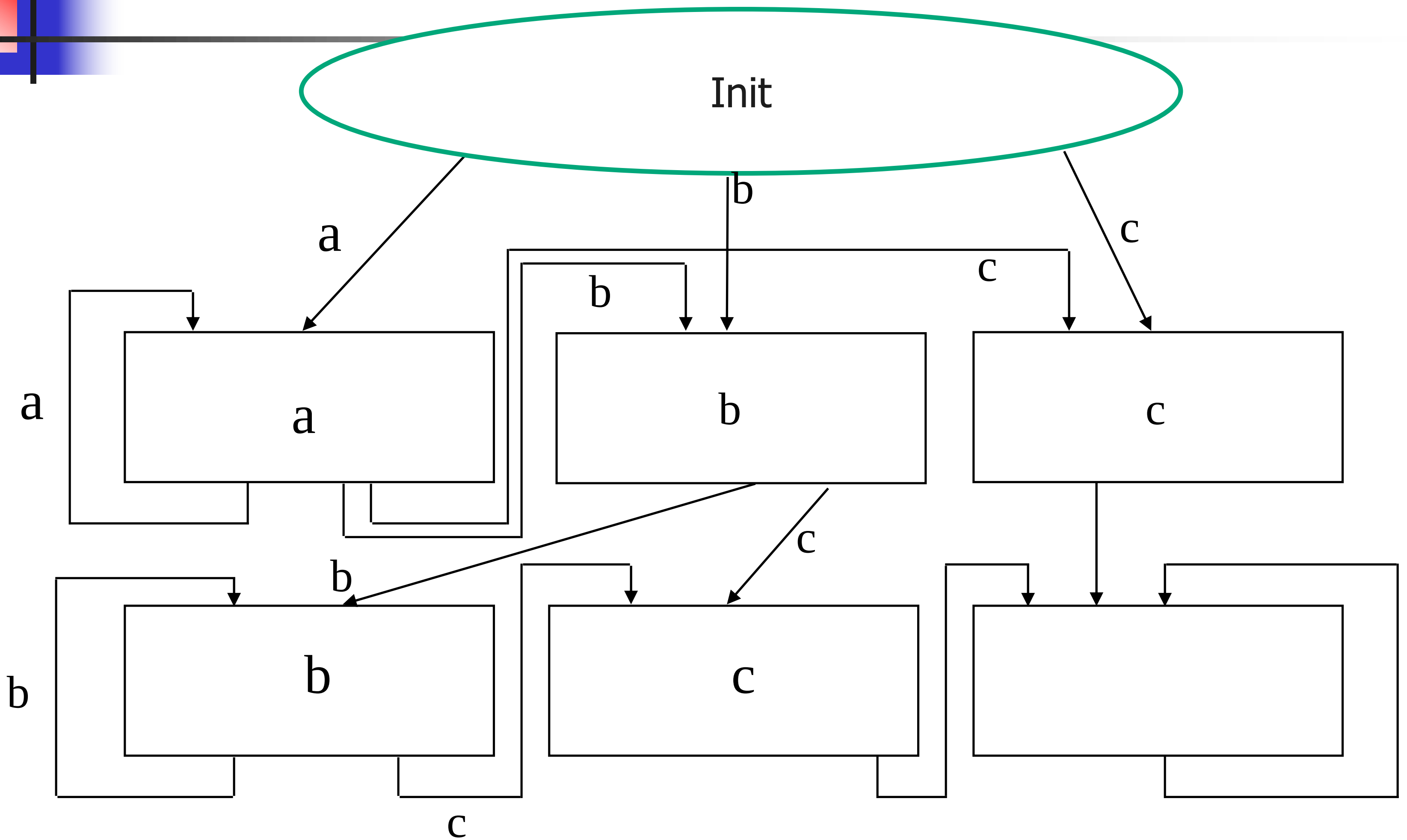
All nodes with c, or without bUc.

Acceptance w.r.t. a*U* (b*U* c)



All nodes with bUc or without aU(bUc).

The automaton (without the accepting conditions)



The SPIN System

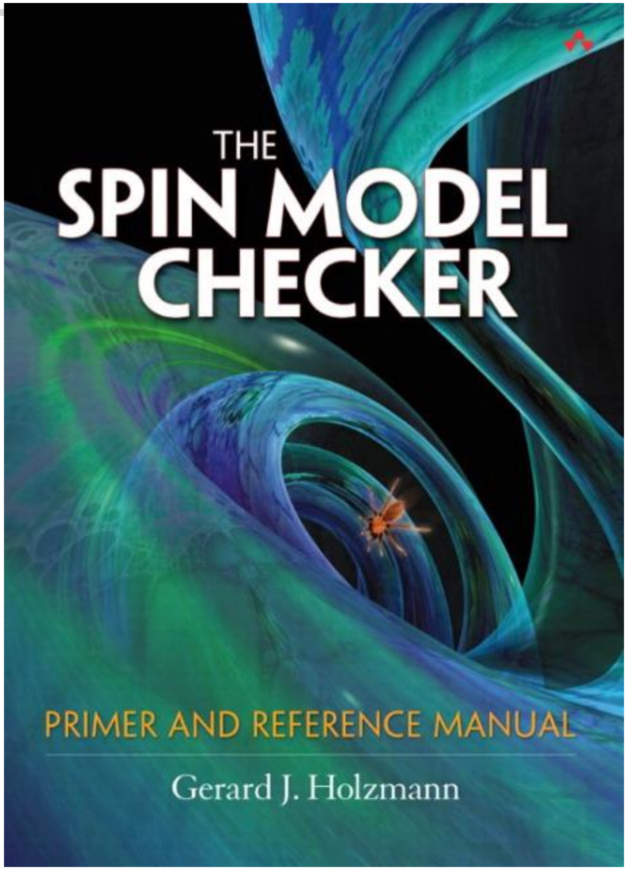
What is SPIN?

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

Documentation

Paper: The model checker SPIN, G.J. Holzmann, IEEE Transactions on Software Engineering, Vol 23, 279-295.

Web: http://www.spinroot.com



The language of SPIN

- The expressions are from C.
- The communication is from CSP.
- The constructs are from Dijkstra’s Guarded Command.

Expressions

- Arithmetic: +, -, *, /, %
- Comparison: >, >=, <, <=, ==, !=
- Boolean: &&, ||, !
- Assignment: =
- Increment/decrement: ++, --

Declaration

- byte name1, name2=4, name3;
- bit b1,b2,b3;
- short s1,s2;
- int arr1[5];

Message types and channels

- mtype = {OK, READY, ACK}
- mtype Mvar = ACK

- chan Ng=[2] of {byte, byte, mtype},
  Next=[0] of {byte}

  Ng has a buffer of 2, each message consists of two bytes and an enumerable type (mtype).
  Next is used with *handshake* message passing.

Sending and receiving a message

Channel declaration:

- chan qname=[3] of {mtype, byte, byte}

In sender:

- qname!tag3(expr1, expr2)
  or equivalently:
  qname!tag3, expr1, expr2

In Receiver:

- qname?tag3(var1,var2)

Defining an array of channels

Channel declaration:

- chan qname=[3] of {mtype, byte, byte}
  defines a channel with buffer size 3.
- chan comm[5]=[0] of {byte, byte}
  defines an array of channels (indexed 0 to 4. Communication is synchronous (handshaking), meaning that the sender waits for the receiver.

Condition

```
if
:: x%2==1 -> z=z*y; x--
:: x%2==0 -> y=y*y; x=x/2
fi
```

If more than one guard is enabled: a nondeterministic choice.

If no guard is enabled: the process waits (until a guard becomes enabled).

Looping

```
do
:: x>y -> x=x-y
:: y>x -> y=y-x
:: else break
od;
```

Normal way to terminate a loop: with *break*. (or goto).

As in condition, we may have a nondeterministic loop or have to wait.

Processes

Definition of a process:

```
proctype prname (byte Id; chan Comm)
{
    statements
}
```

Activation of a process:

```
run prname (7, Con[1]);
```

init process is the root of activating all others

```
init {   statements }
init {byte I=0;
        atomic{do
                ::I<10 -> run prname(I, chan[I]);
                    I=I+1
                ::I=10 -> break;
                od}}
```

atomic allows performing several actions as one atomic step.

Exmaples of Mutual exclusion

Reference:

A. Ben-Ari, Principles of Concurrent and Distributed Programs, Prentice-Hall 1990.

General structure of mutual exclusion algorithm\

loop

Non_Critical_Section
;
TR:Pre_Protocol;
CR:Critical_Section;
Post_protocol;
end loop;

Propositions:
inCRi, inTRi.

Properties

loop

    Non_Critical_Section;
    TR:Pre_Protocol;
    CR:Critical_Section;
    Post_protocol;

end loop;

Assumption:

~<>[]inCRi

Requirements:

[]~(inCR0/\inCR1)

[](inTRi→<>inCRi)

Not assuming:

[]<>inTRi

Turn:bit:=1;

```
task P0 is
begin
  loop
    Non_Critical_Sec;
    Wait Turn=0;
    Critical_Sec;
    Turn:=1;
  end loop
end P0.
```

```
task P1 is
begin
  loop
    Non_Critical_Sec;
    Wait Turn=1;
    Critical_Sec;
    Turn:=0;
  end loop
end P1.
```

Translating into SPIN

```
#define  critical  (incrit[0] ||incrit[1])
byte turn=0, incrit[2]=0;
proctype P (bool id)
{ do
   :: 1 ->
      do
      :: 1 -> skip
      :: 1 -> break
      od;
try:if
       ::turn==id -> skip
     fi;
  cr:incrit[id]=1;
     incrit[id]=0;
     turn=1-turn
   od}
init { atomic{
   run P(0);  run P(1) } }
```

Running SPIN

- Can download and implement (for free) using www.spinroot.com
- Available in our system.
- Graphical interface: *xspin*

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.
```

Project

- Model in Spin
- Specify properties
- Do model checking
- Can this work without fairness?
- What to do with fairness?

Modeling issues

Book: chapters 4.12, 5.4, 8.4, 10.1

Fairness

(Book: Chapter 4.12, 8.3, 8.4)

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

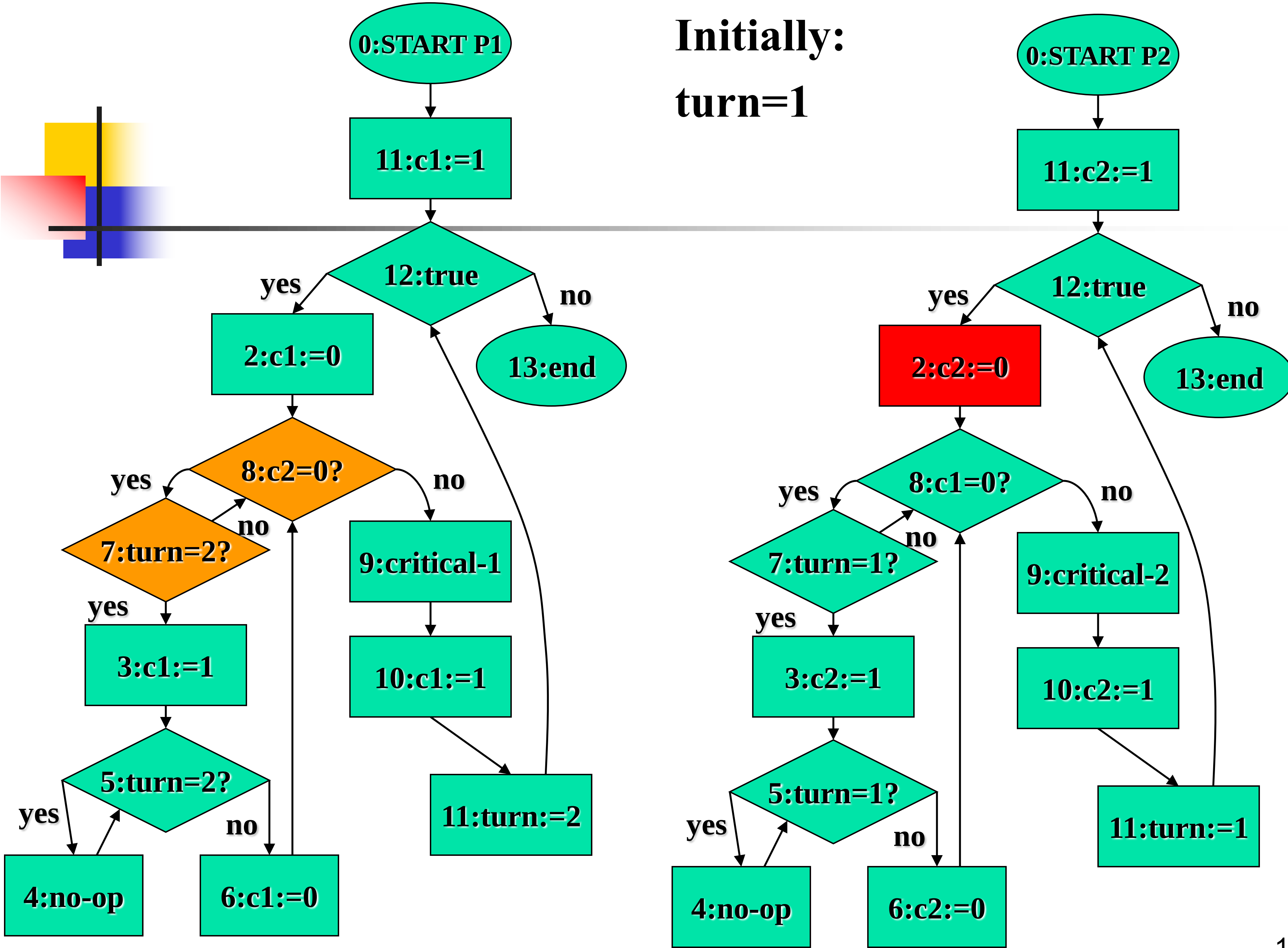
P1 waits for P2 to set c2 to 1 again. Since turn=1 (priority for P1), P2 is ready to do that. But never gets the chance, since P1 is constantly active checking c2 in its while loop.

```
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?

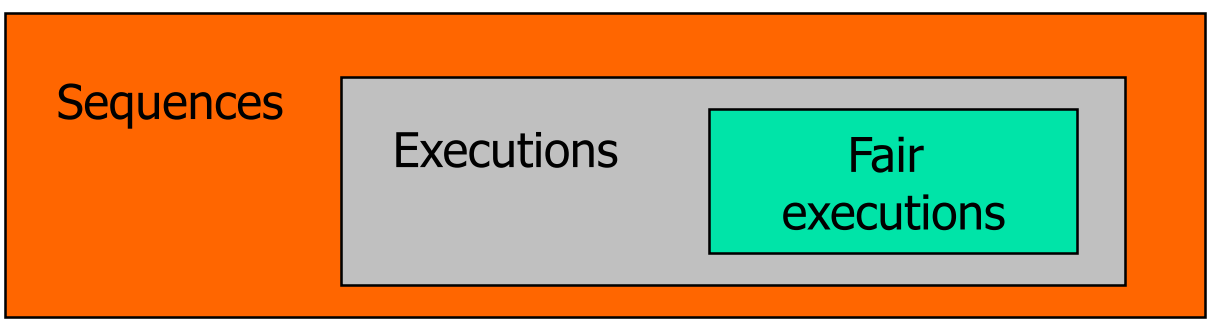
- The execution is *unfair* to P2. It is not allowed a chance to execute.
- Such an execution is due to the interleaving model (just picking an enabled transition to execute next).
- If it did, it would continue and set c2 to 0, which would allow P1 to progress.
- *Fairness* = excluding some of the executions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

Recall: The interleaving model

- An execution is a finite or infinite sequence of states s_0, s_1, s_2, …
- 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 non 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

Initially: x=0; y=0;

P1::x=1

P2: do
:: y==0 ->
if
:: *true*
:: x==1 -> y=1
fi
od

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.

pc1=l0➔(pc1,x):=(l1,1) /* x=1 */

pc2=r0/\y=0➔pc2=r1 /* y==0*/

pc2=r1➔pc2=r0 /* *true* */

pc2=r1/\x=1➔(pc2,y):=(r0,1)
/* x==1 ➔ y:=1 */

Weak transition fairness

Initially: x=0; y=0;

P1::x=1

```
P2: do
      :: y==0 ->
         if
         :: true
         :: x==1 -> y=1
         fi
      od
```

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 terminates, since the transition for checking x==1 is not continuously enabled (program counter not always there).

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

P2: do
:: y==0 ->
if
:: true
:: x==1 -> y=1
fi
od

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.

Specifying fairness conditions

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

  $s_0\ \alpha_1\ s_1\ \alpha_1\ s_2\ \ldots$
- Use transition predicates exec_{α}.

Some fairness definitions

$exec_\alpha$ α is executed.

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

en_α α is enabled.

en_{Pi} some transition of process Pi is enabled.

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

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

- *Weak transition fairness:*

  $\bigwedge_{\alpha \in T} (\text{<>[]}en_\alpha \rightarrow \text{[]<>}exec_\alpha)$.

  Equivalently: $\bigwedge_{\alpha \in T} \neg\text{<>[]}(en_\alpha \wedge \neg exec_\alpha)$
- *Weak **process** fairness:*

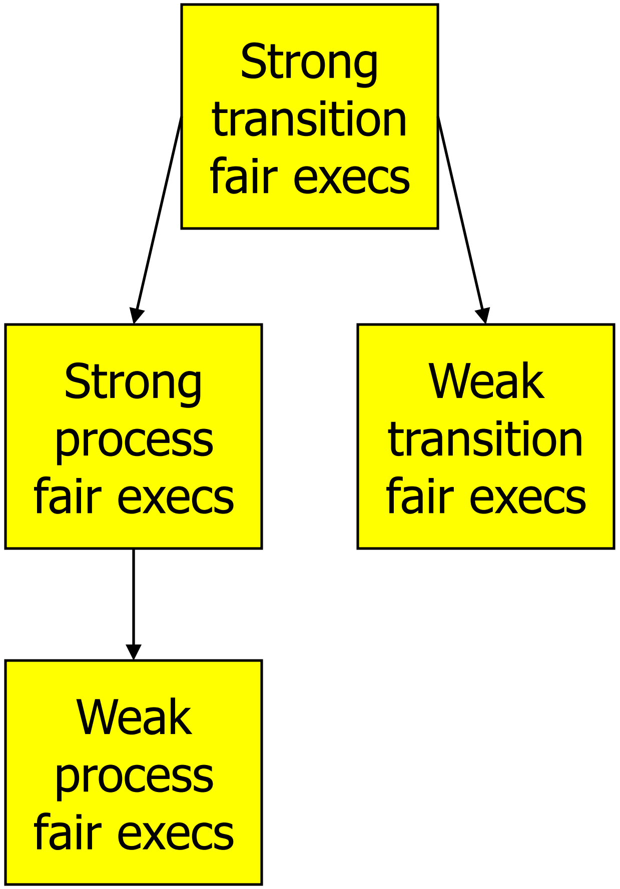
  $\bigwedge_{Pi} (\text{<>[]}en_{Pi} \rightarrow \text{[]<>}exec_{Pi})$
- ***Strong** transition fairness:*

  $\bigwedge_{\alpha \in T} (\text{[]<>}en_\alpha \rightarrow \text{[]<>}exec_\alpha)$
- ***Strong process** fairness:*

  $\bigwedge_{Pi} (\text{[]<>}en_{Pi} \rightarrow \text{[]<>}exec_{Pi})$

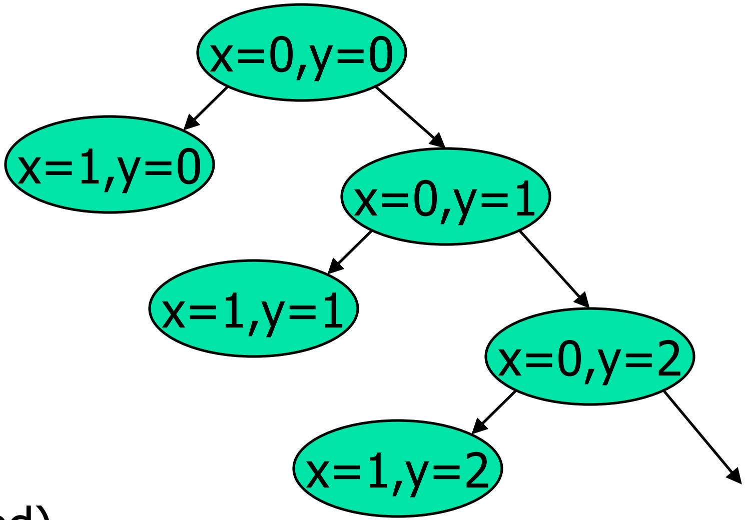
“Weaker fairness condition”

- *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 -> y=y+1
          :: x==1 -> 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 φ, 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 on occurrence of a transition from *P*, or
- it contains a state where *P* is disabled.

- Weak transition fairness: similar.
- Strong fairness: much more difficult algorithm.

Abstractions

(Book: Chapter 10.1)

Problems with software analysis

- Many possible outcomes and interactions.
- Not manageable by an algorithm (undecideable, complex).
- Requires a lot of practice and ingenuity (e.g., finding invariants).

More problems

- Testing methods fail to cover potential errors.
- Deductive verification techniques require
  - too much time,
  - mathematical expertise,
  - ingenuity.
- Model checking requires a lot of time/space and may introduce modeling errors.

How to alleviate the complexity?

- Abstraction
- Compositionality
- Partial Order Reduction
- Symmetry

Abstraction

- Represent the program using a smaller model.
- Pay attention to preserving the checked properties.
- Do not affect the flow of control.
- [Abstract interpretation (using Galois connection) is equivalent to simulation!]

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:
  $$\mathrm{sgn}(x) = \begin{cases} 1 & \text{if } x>0, \\ 0 & \text{if } x=0, \text{ and} \\ -1 & \text{if } x<0. \end{cases}$$
  $\mathrm{sgn}(x)*\mathrm{sgn}(y)=\mathrm{sgn}(x*y)$.

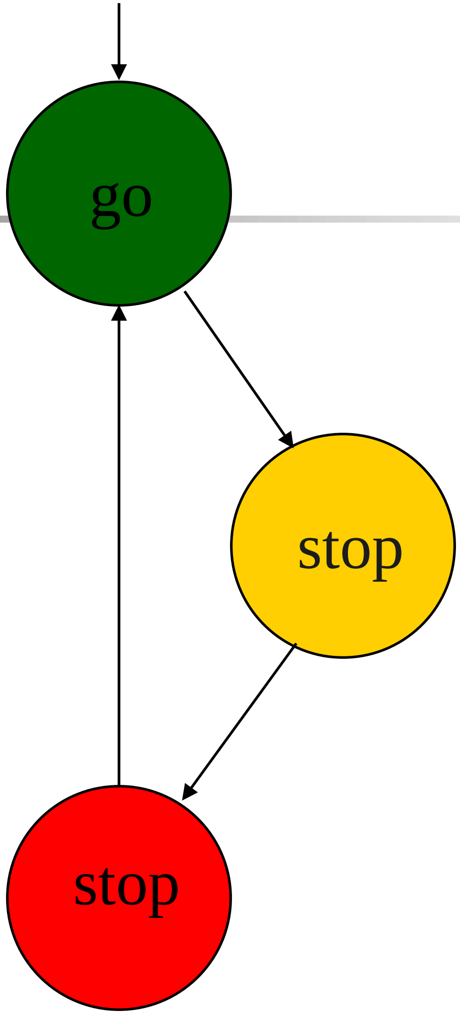
Abstraction mapping

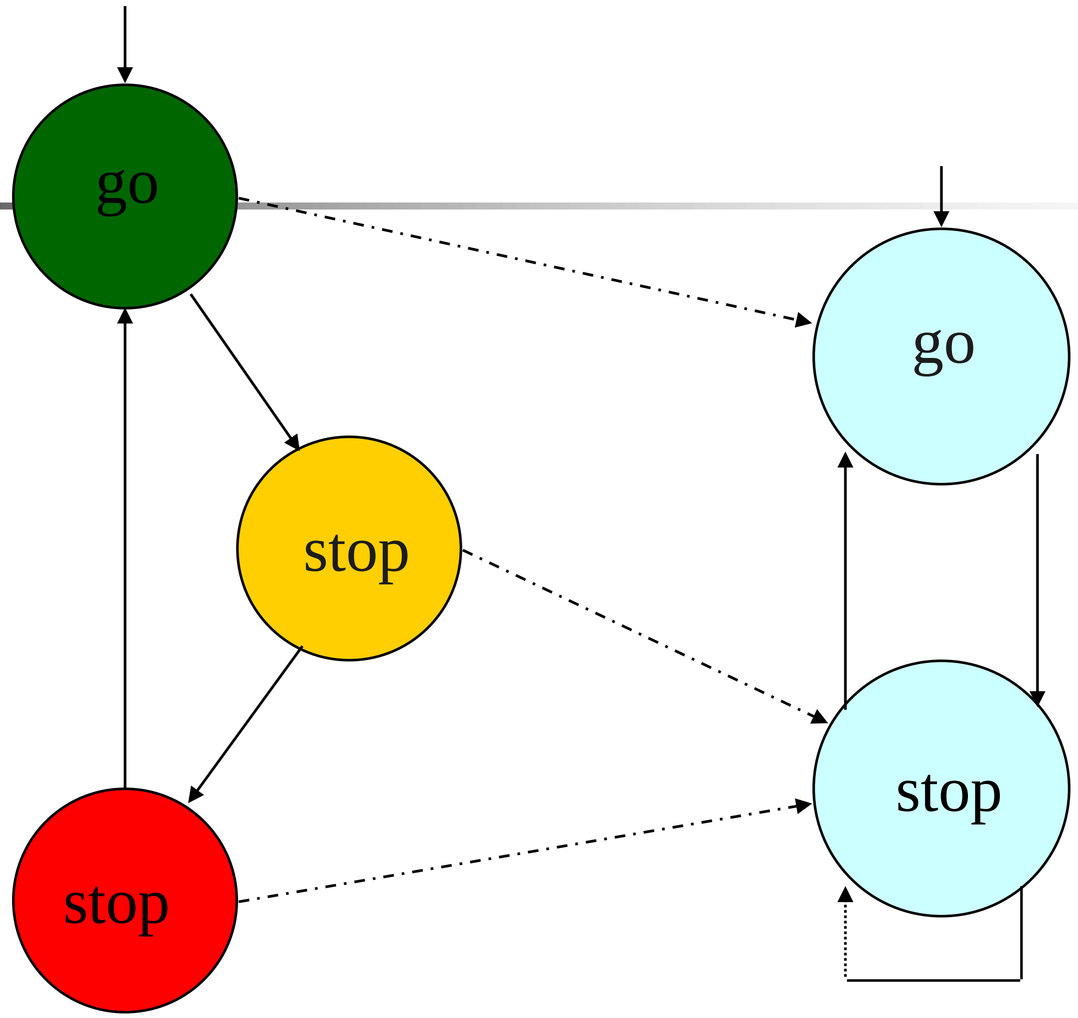
- S - states, I - initial states. L(s) - labeling.
- R(S,S) - transition relation.
- h(s) maps s into its abstract image.

Full model	-h→	Abstract model
I(s)	→	I(h(s))
R(s, t)	→	R(h(s),h(t))

L(h(s))=L(s)

Traffic light example



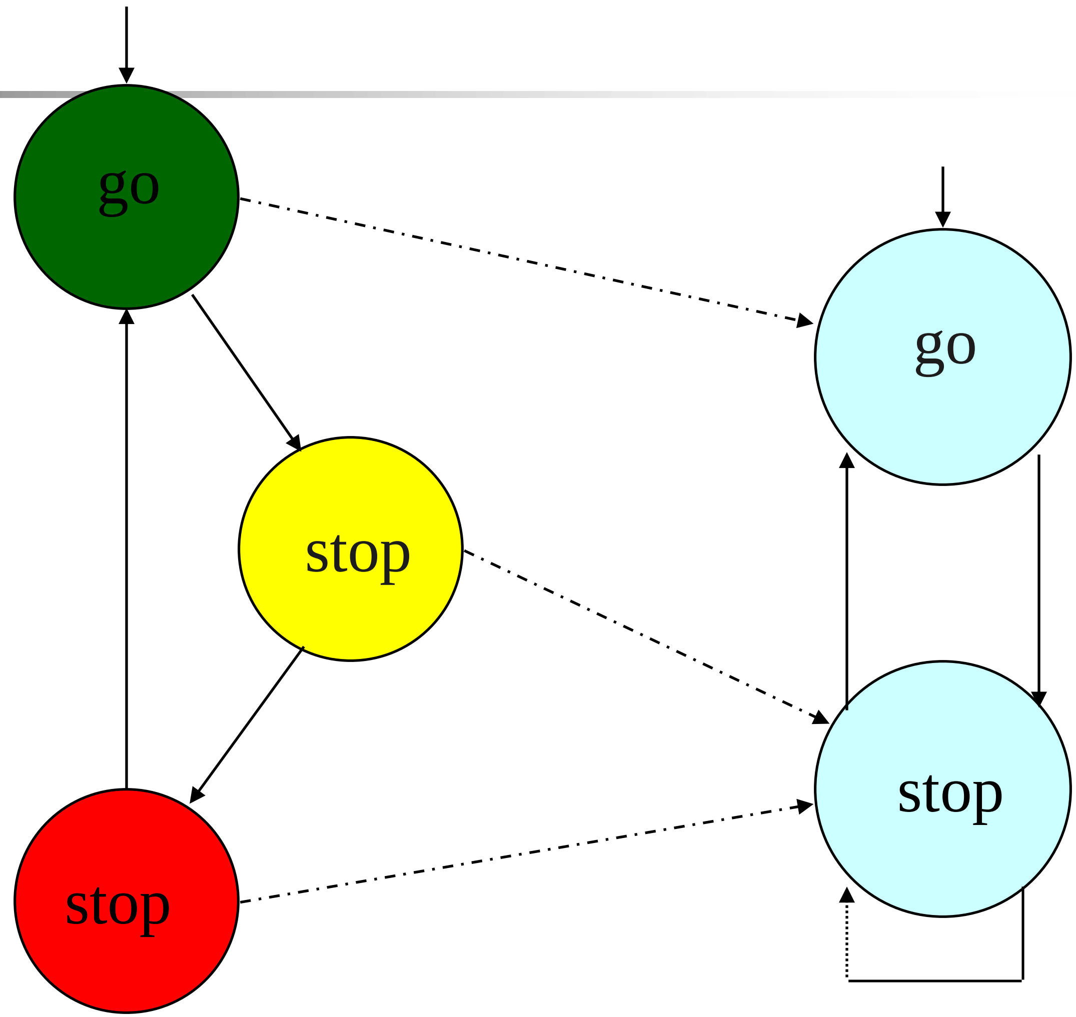

go
stop
stop
go
stop

What do we preserve?

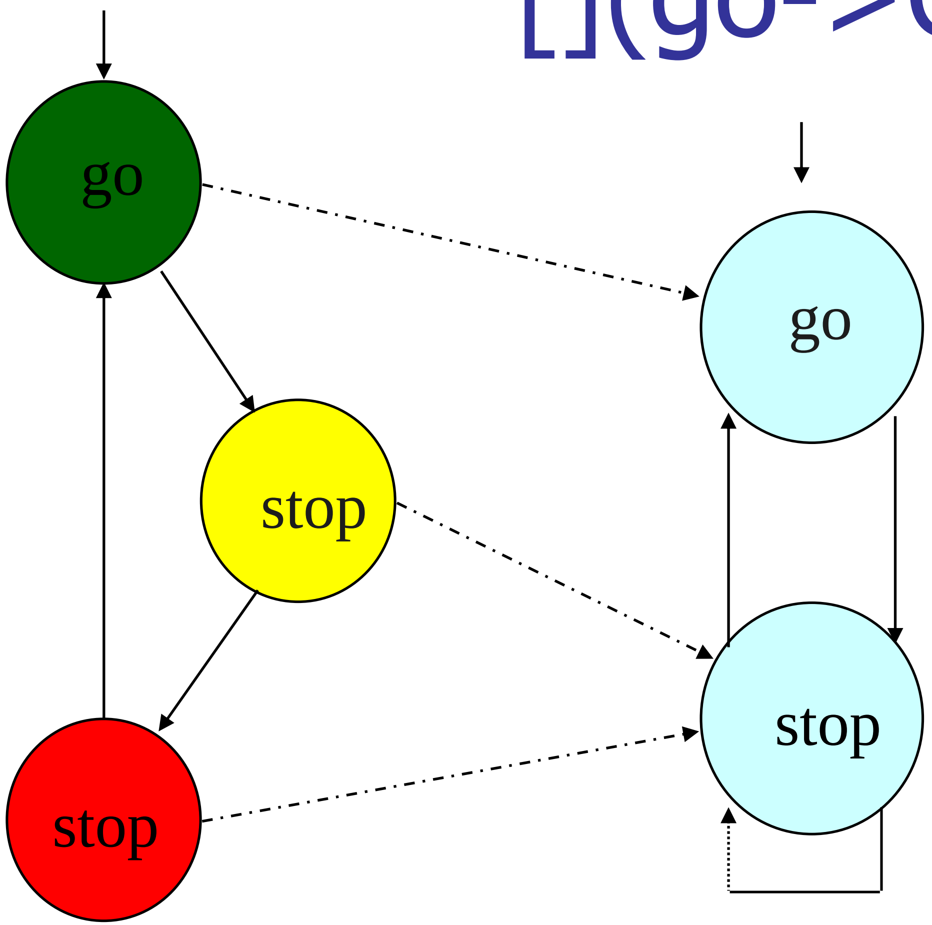
Every execution of the full model can be simulated by an execution of the reduced one.

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 [false negatives].



Preserved: [](go->O stop)



Not preserved:

[]<>go

Counterexamples need to be checked.

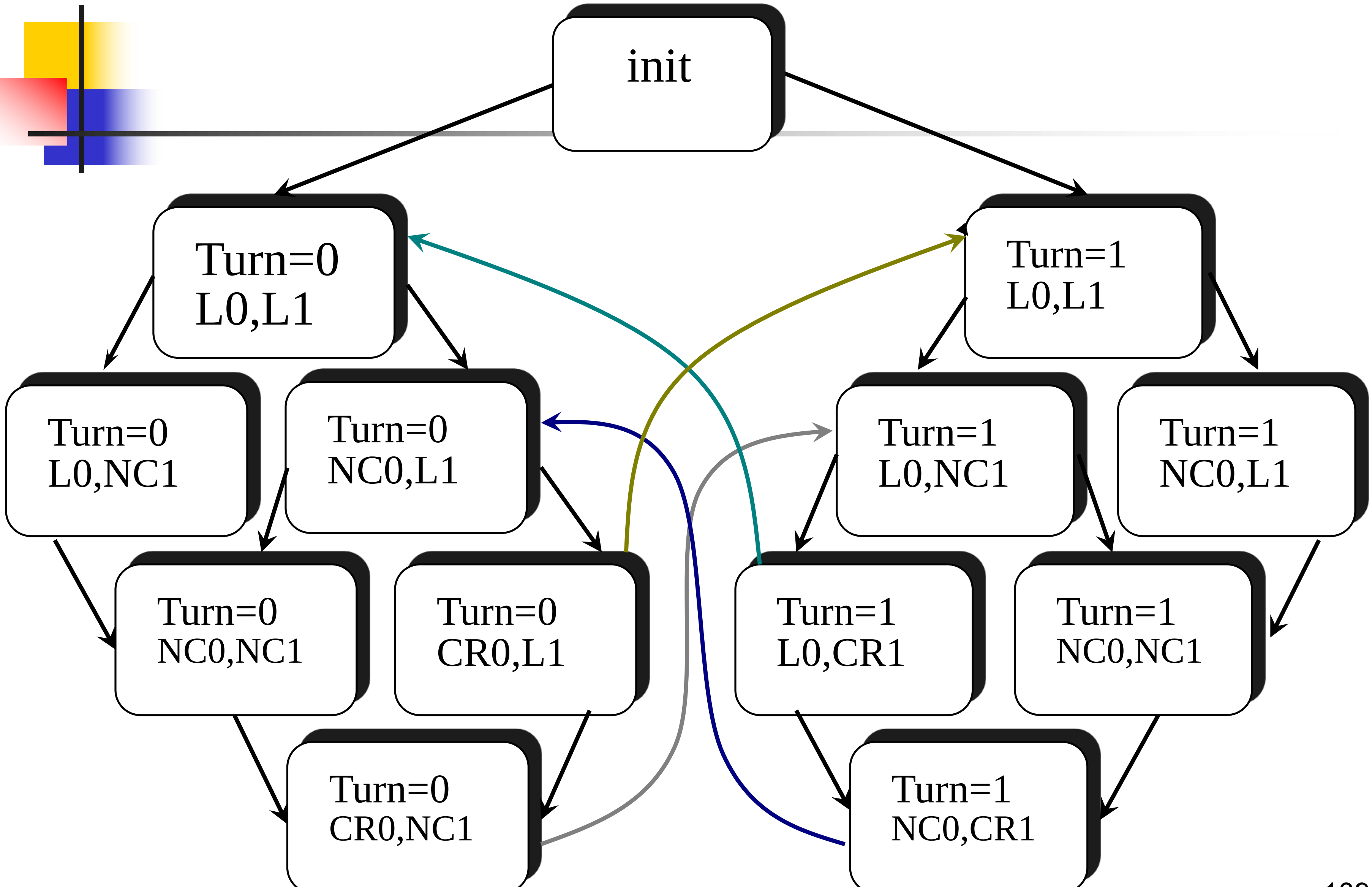
Symmetry

- A permutation is a one-one and onto function p:A➔A.
  For example, 1➔3, 2➔4, 3➔1, 4➔5, 5➔2.
- One can combine permutations, e.g.,
  p1: 1➔3, 2➔1, 3➔2
  p2: 1➔2, 2➔1, 3➔3
  p1@p2: 1➔3, 2➔2, 3➔1
- A set of permutations with @ is called a symmetry group.

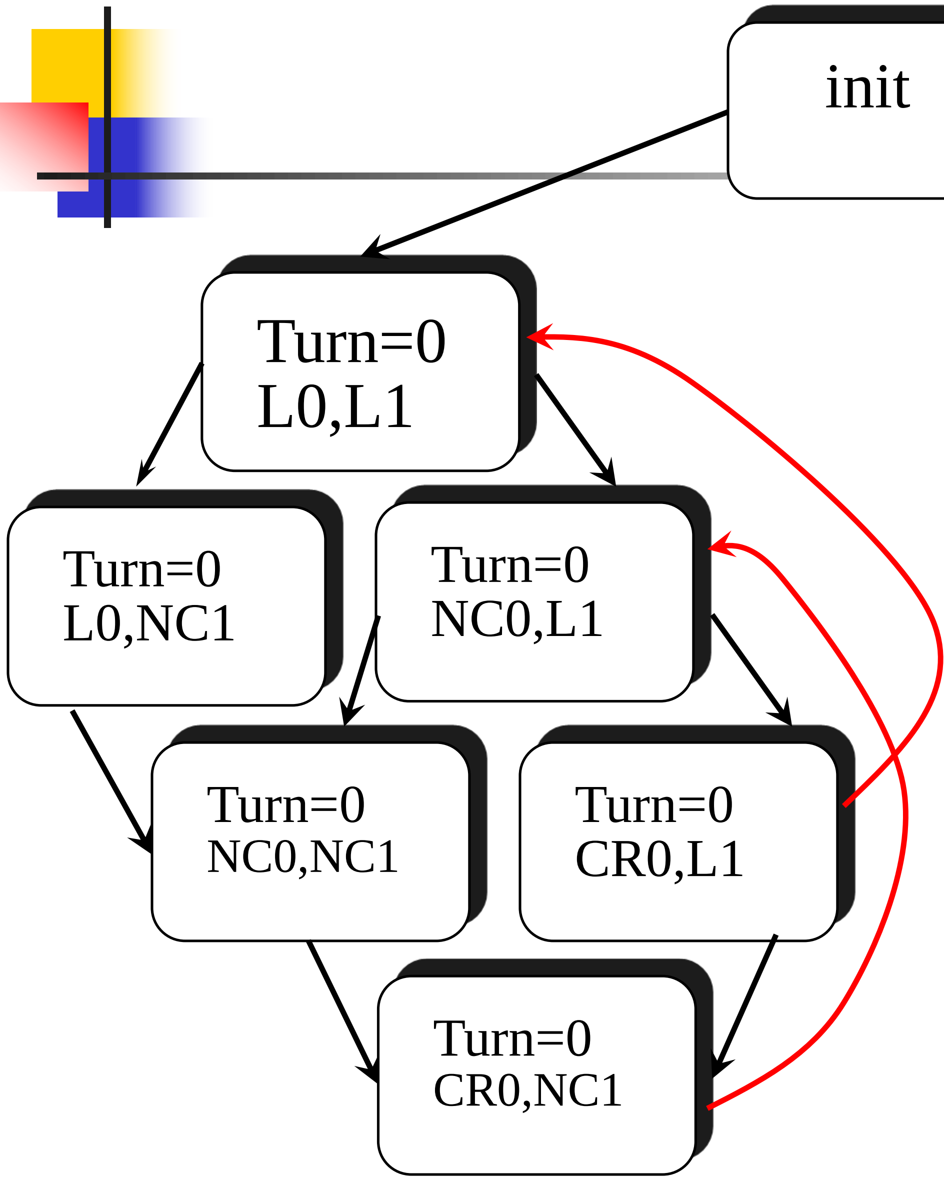
Using symmetry in analysis

- Want to find some symmetry group such that for each permutation p in it, R(s,t) if and only if R(p(s), p(t)) and L(p(s))=L(s).

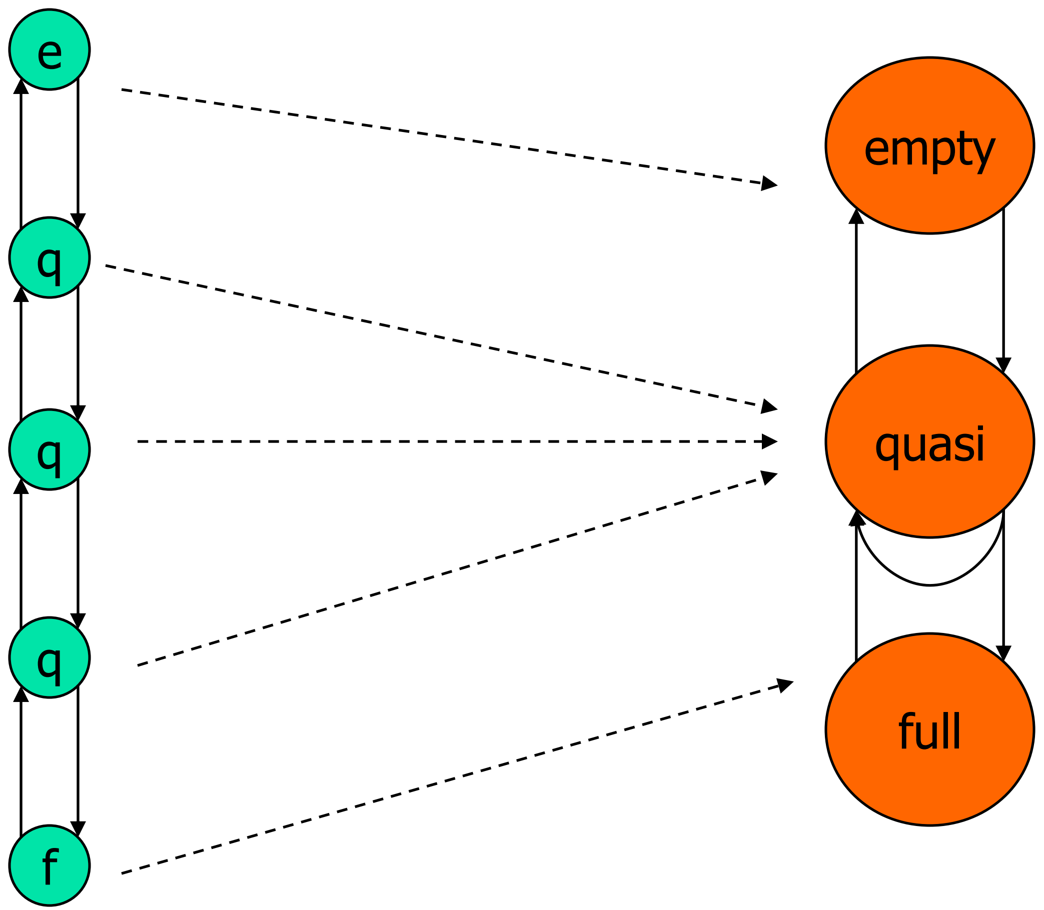
- Let K(s) be all the states that can be permuted to s. This is a set of states such that each one can be permuted to the other.


init
Turn=0
L0,L1
Turn=1
L0,L1
Turn=0
L0,NC1
Turn=0
NC0,L1
Turn=1
L0,NC1
Turn=1
NC0,L1
Turn=0
NC0,NC1
Turn=0
CR0,L1
Turn=1
L0,CR1
Turn=1
NC0,NC1
Turn=0
CR0,NC1
Turn=1
NC0,CR1

The quotient model



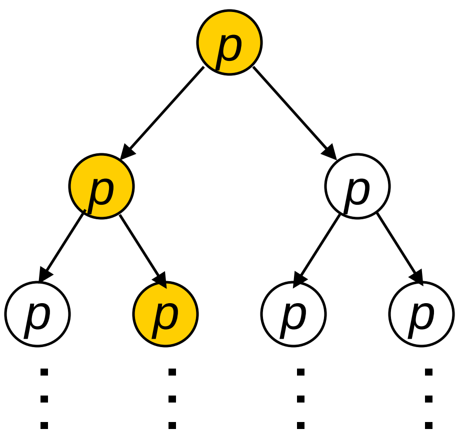
Homework: what is preserved in the following buffer abstraction? What is not preserved?



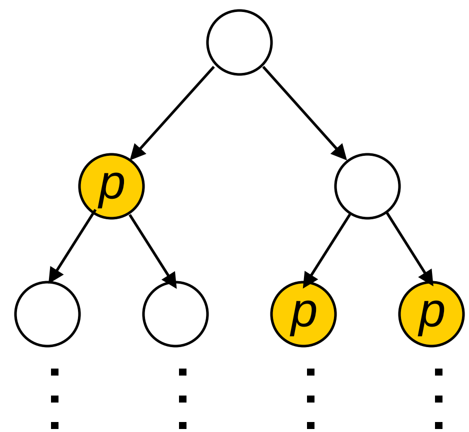
BDD representation

Computation Tree Logic

EG p

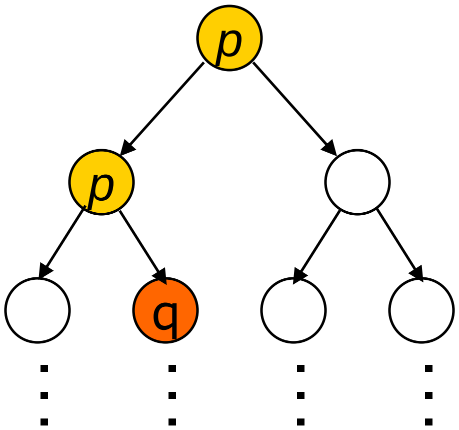


AF p

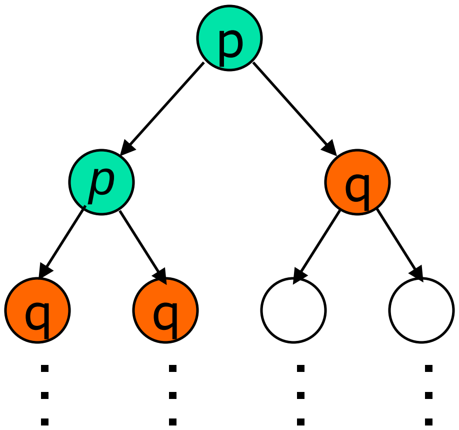


Computation Tree Logic

E pUq



A pUq



Example formulas

CTL formulas:

- **mutual exclusion:** **AG** $\neg(cs_1 \wedge cs_2)$
- **non starvation:** **AG** (request $\Rightarrow$ **AF** grant)
- **"sanity" check:** **EF** request

Model Checking M |= f

[Clarke, Emerson, Sistla 83]

- The **Model Checking** algorithm works **iteratively**
  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|×|f|)**

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 := emptyset; Q' := S_g ;

while Q $\neq$ Q' do

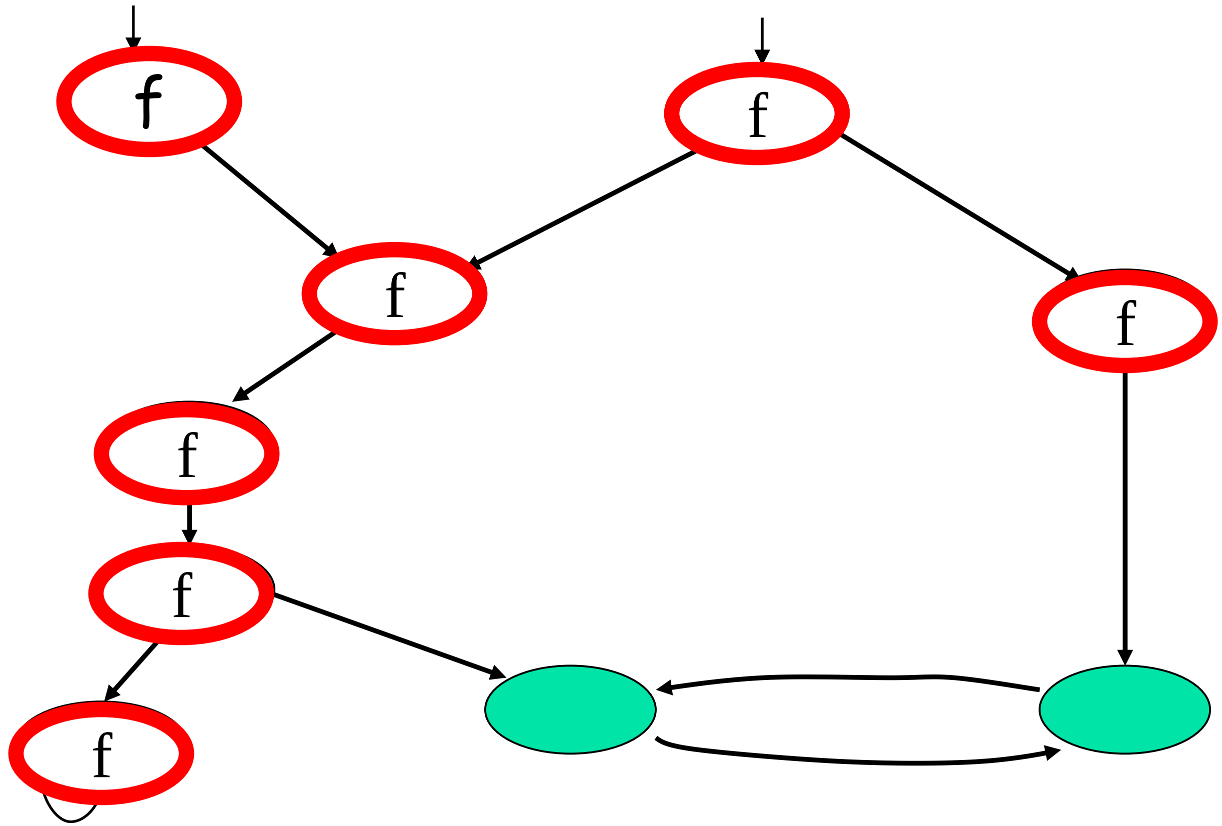
Q := Q';

Q' := Q $\cup$ { s | $\exists$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 $M = < S, I, R, L >$ and S_g and returns S_f

procedure CheckEG (S_g)

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

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

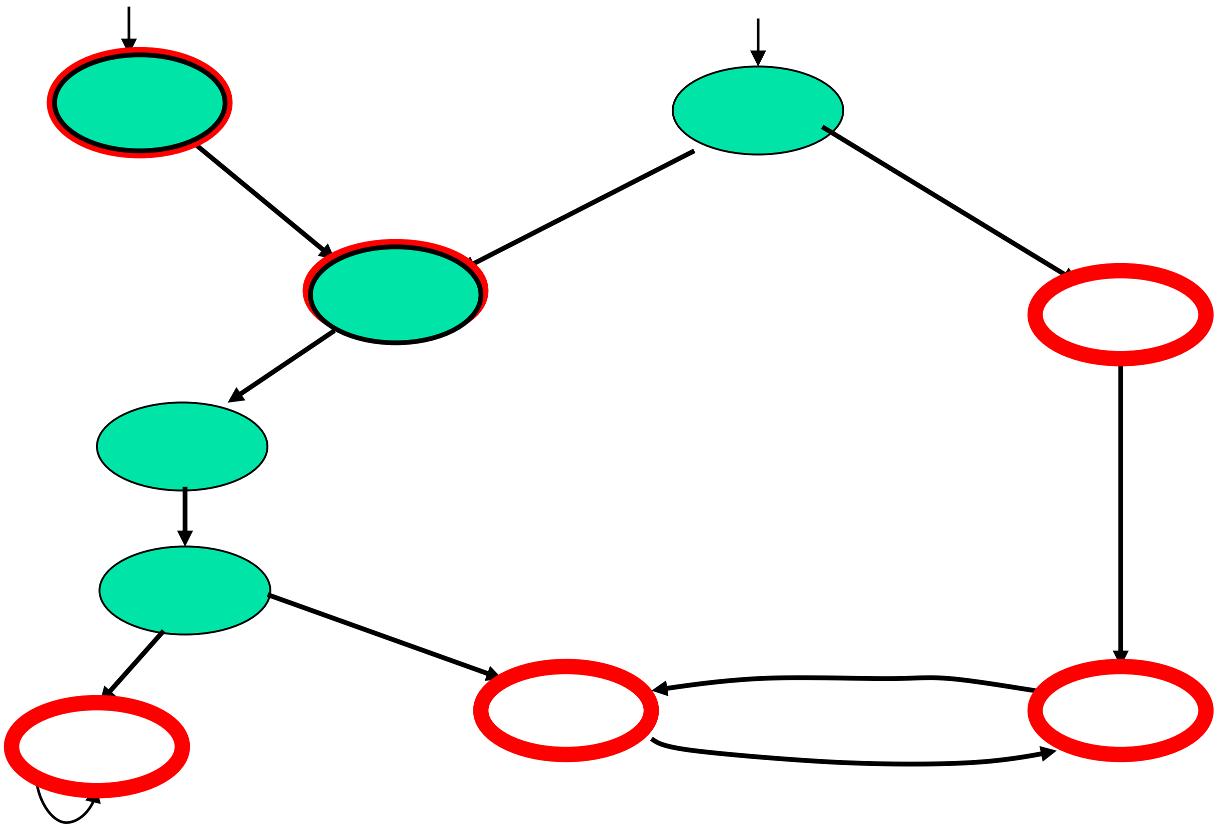
$\quad Q := Q';$

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

end while

$S_f := Q\ ;\quad \mathrm{return}(S_f\ )$

Example: **f = EG g**



Symbolic model checking [Burch, Clarke, McMillan, Dill 1990]

If the model is given **explicitly** (e.g. by **adjacent matrix**) then only systems with about **ten** Boolean variables (~1000 states) can be handled

Symbolic model checking uses

Binary Decision Diagrams (BDDs)

to represent the **model** and **sets of states**. It can handle systems with **hundreds** of Boolean variables.

Binary decision diagrams (BDDs) [Bryant 86]

- Data structure for representing Boolean functions
- Often **concise** in memory
- **Canonical** representation
- **Boolean operations** on BDDs can be done in **polynomial time** in the BDD size

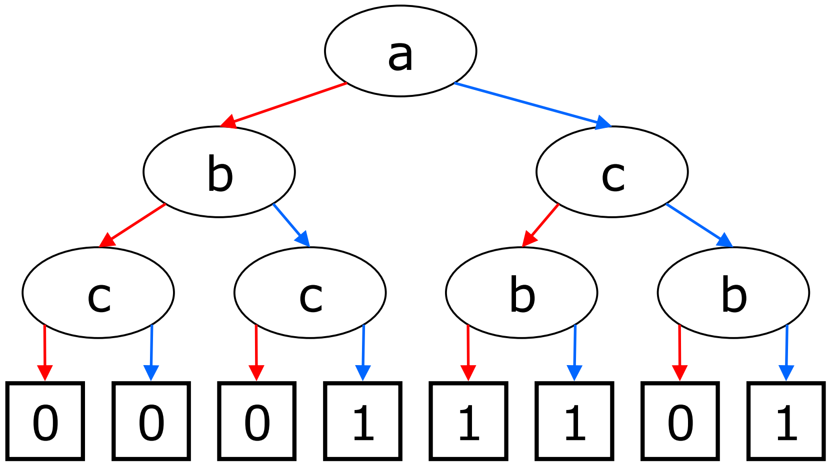
BDDs in model checking

- Assume that **states** in model M are **encoded by $\{0,1\}^n$** and described by Boolean variables $\mathbf{v_1 \ldots v_n}$
- $\mathbf{S_f}$ can be represented by a BDD over $\mathbf{v_1 \ldots v_n}$
- **R** (a set of pairs of states **(s,s')**) can be represented by a BDD over $\mathbf{v_1 \ldots v_n\ v_1' \ldots v_n'}$

BDD definition

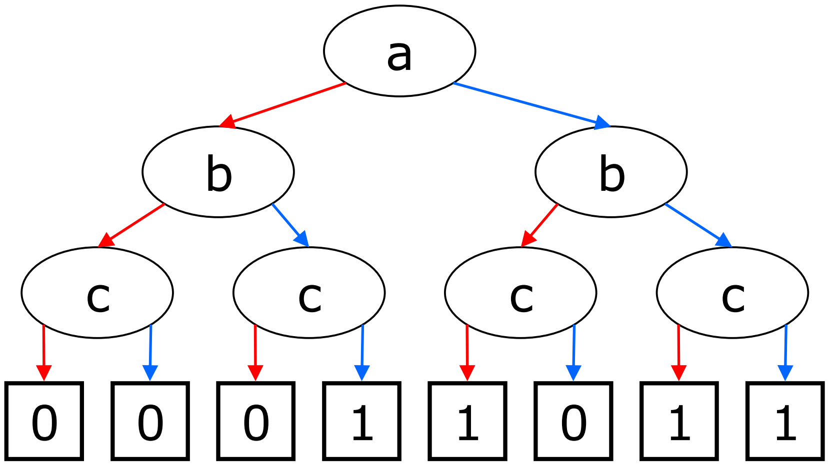
- A tree representation of a Boolean formula.
- Each leaf represents 0 (false) or 1 (true).
- Each internal leaf represents a node.
- If we follow a path in the tree and go from a node left (low) on 0 and right (high) on 1, we obtain a leaf that corresponds to the value of the formula under this truth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

Example



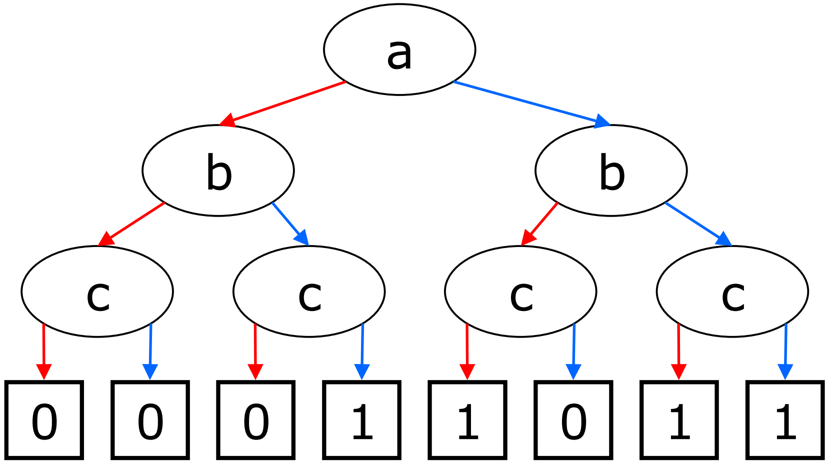
$(a \wedge (b \vee \neg c)) \vee (\neg a \wedge (b \wedge c))$

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



(a/\(b\/¬c))\/(¬a/\(b/\c))

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

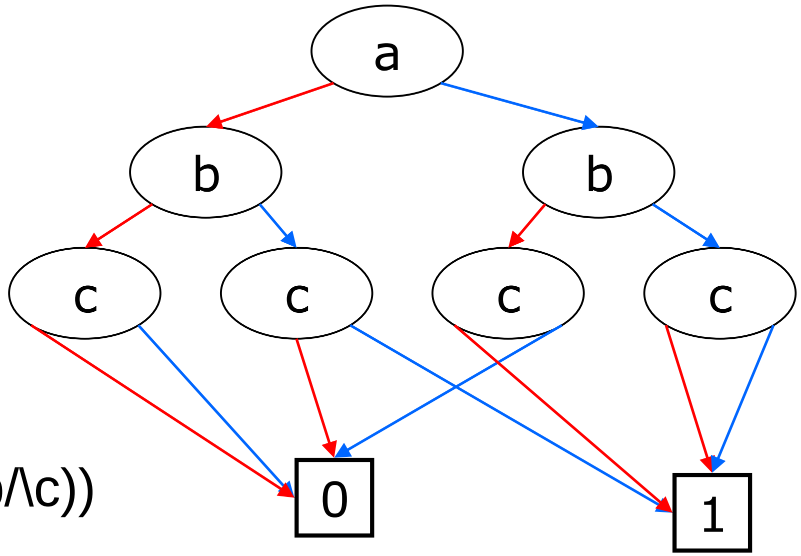


(a/\(b\/¬c))\/(¬a/\(b/\c))

In addition, remove nodes with identical children (low(x)=high(x)).

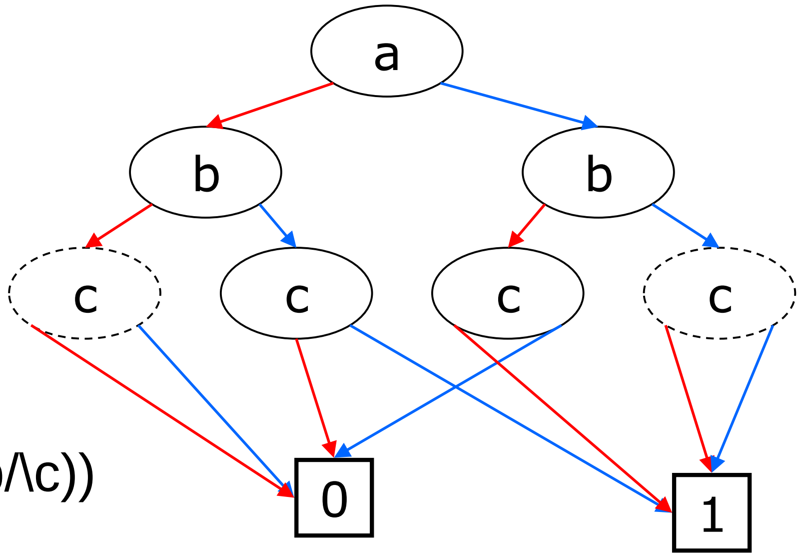
In reduced form: combine all leafs with same values, all isomorphic subgraph.

(a/\(b\/¬c))\/(¬a/\(b/\c))



Unify isomorphic subtrees. Shortcut nodes with identical children (low(x)=high(x)).Apply bottom up until not possible.

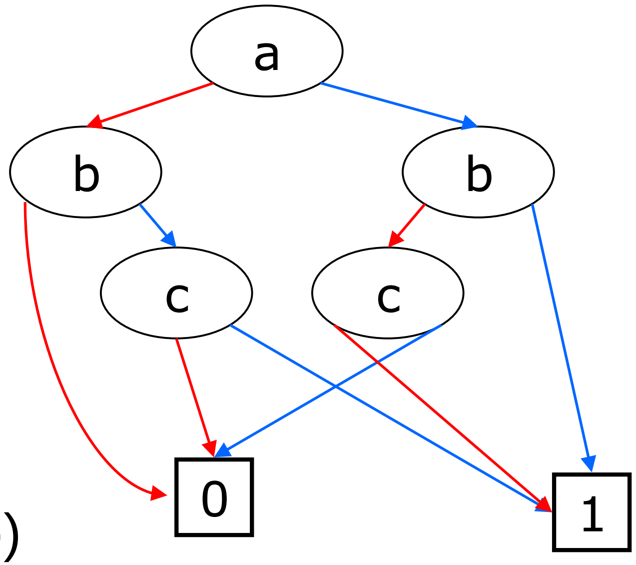
In reduced form: combine all leafs with same values, all isomorphic subgraph.



(a/\(b\/¬c))\/(¬a/\(b/\c))

Unify isomorphic subtrees. Shortcut nodes with identical children (low(x)=high(x)).Apply bottom up until not possible.

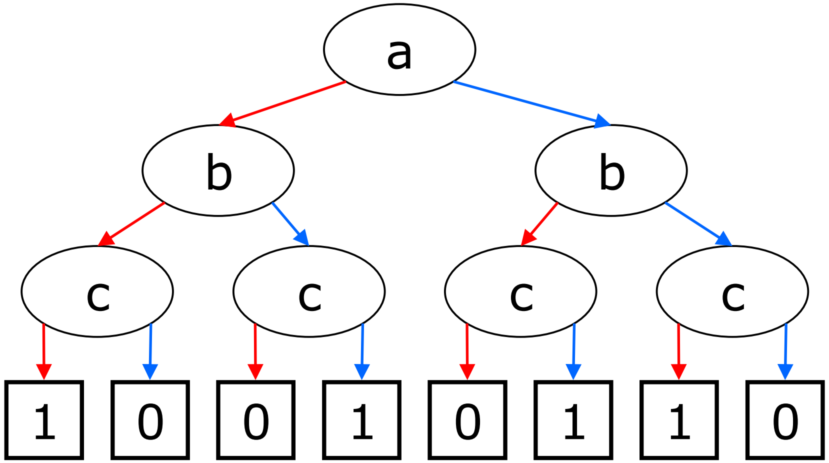
In reduced form: combine all leafs with same values, all isomorphic subgraph.



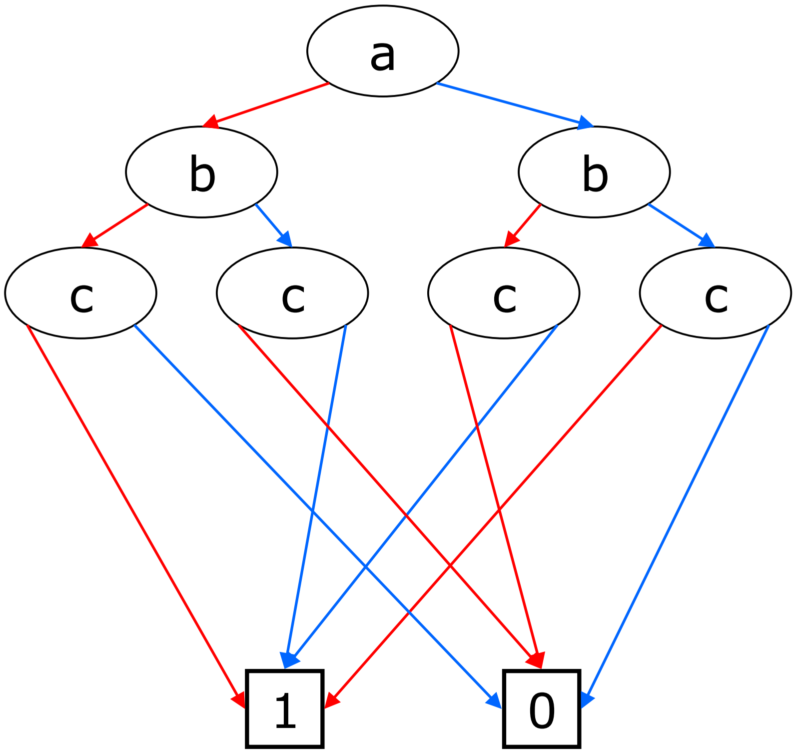
(a/\(b\/¬c))\/(¬a/\(b/\c))

Unify isomorphic subtrees. Shortcut nodes with identical children (low(x)=high(x)).Apply bottom up until not possible.

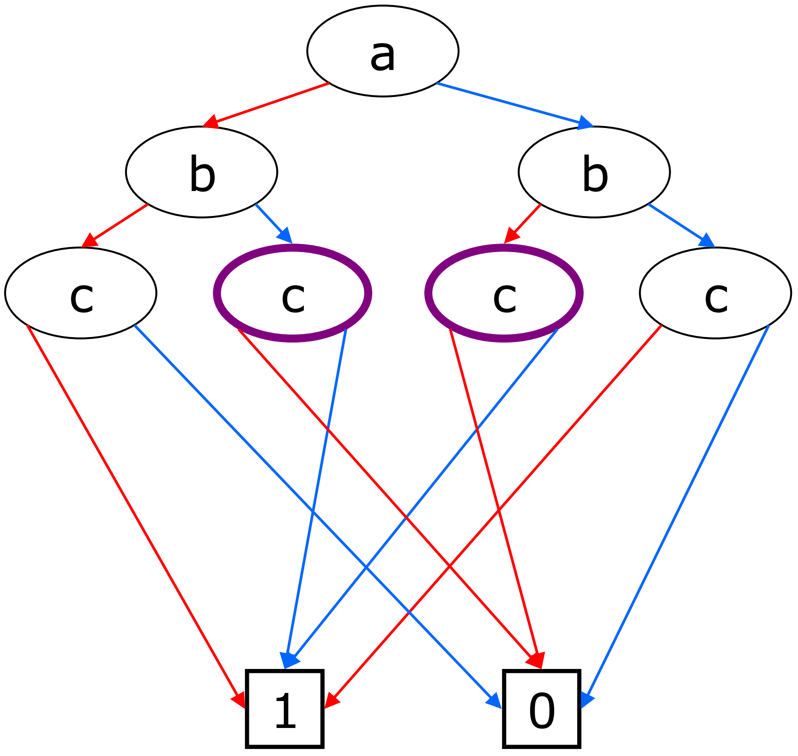
Example, even parity, 3 bits



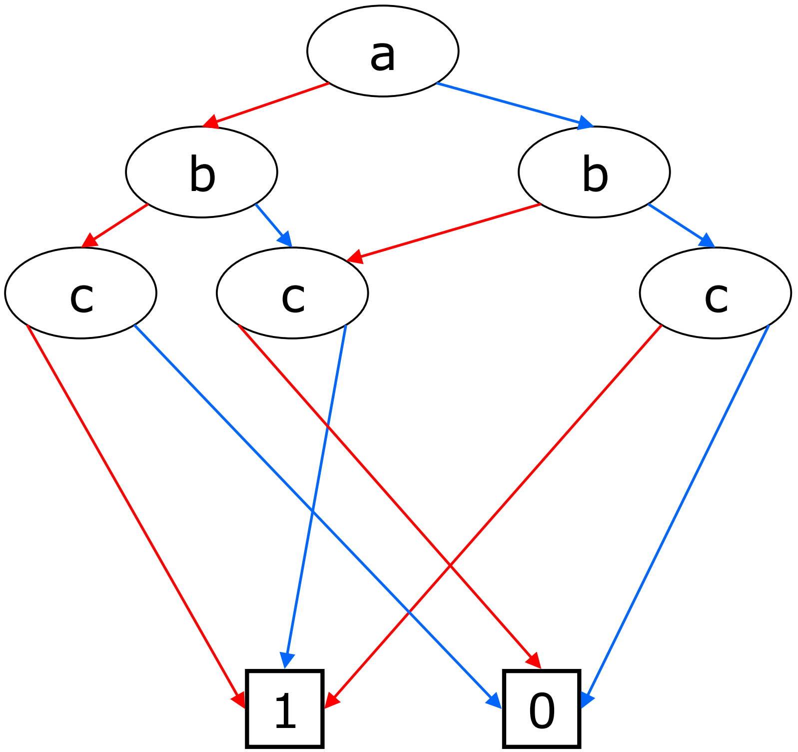
Apply reduce



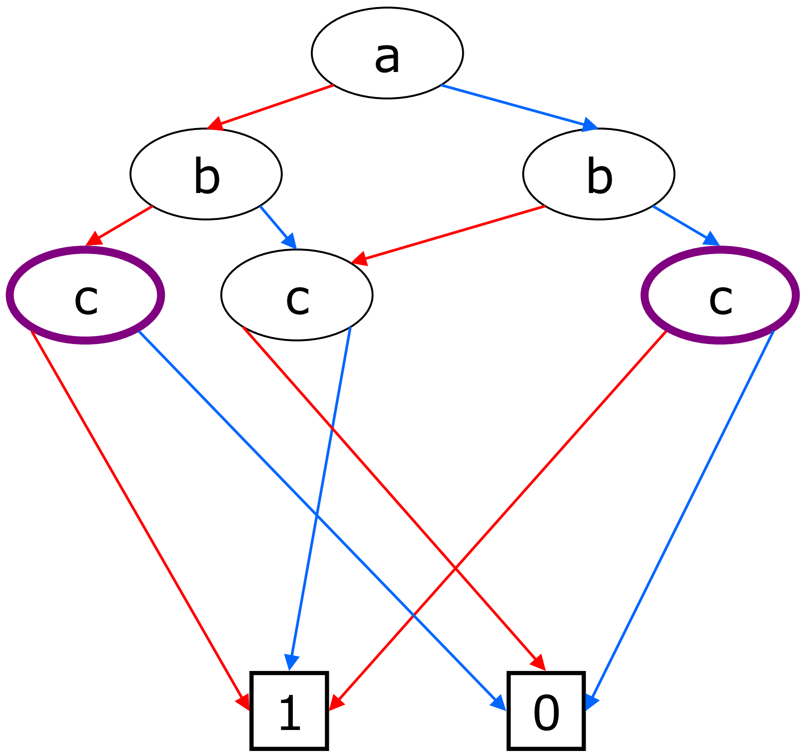
Apply reduce



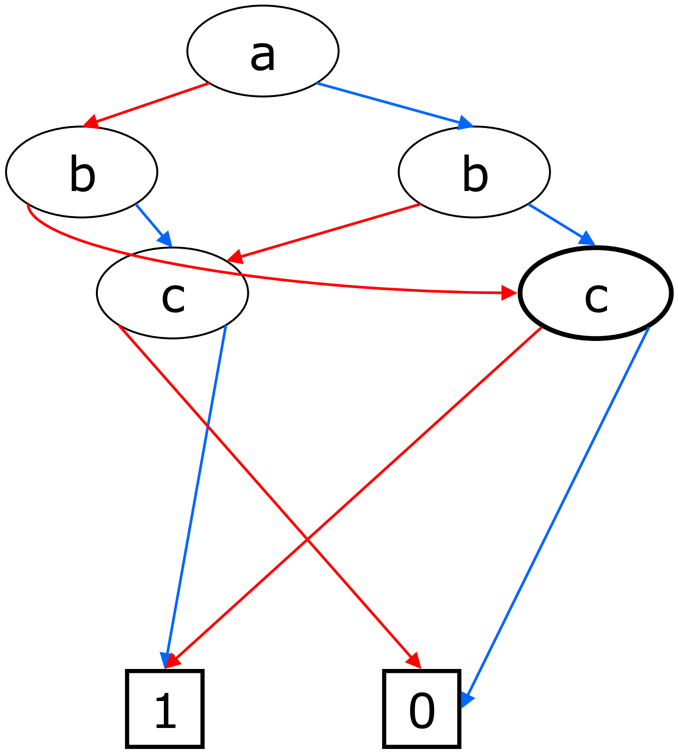
Apply reduce



Apply reduce



Apply reduce



f[0/x], f[1/x] (“restrict” algorithm)

- 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 φ and then "apply" the disjunction.

Shannon expansion of Boolean expression f.

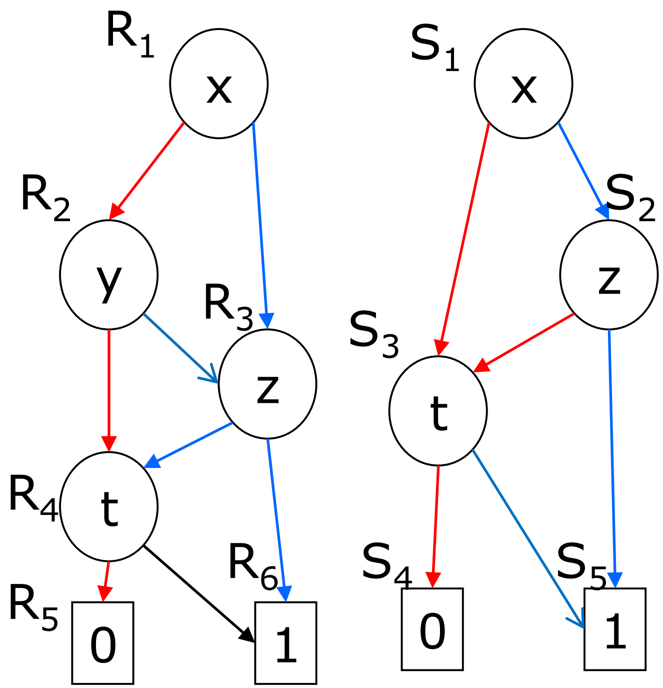
- $f=(\neg x \wedge f[0/x]) \vee (x \wedge f[1/x])$
- Thus, $f\#g$, for some logical operator # is
  $f\#g=(\neg x \wedge f\#g\ [0/x]) \vee (x \wedge f\#g\ [1/x])=$
  $(\neg x \wedge f\ [0/x]\#g\ [0/x]) \vee (x \wedge f\ [1/x]\#g[1/x])$

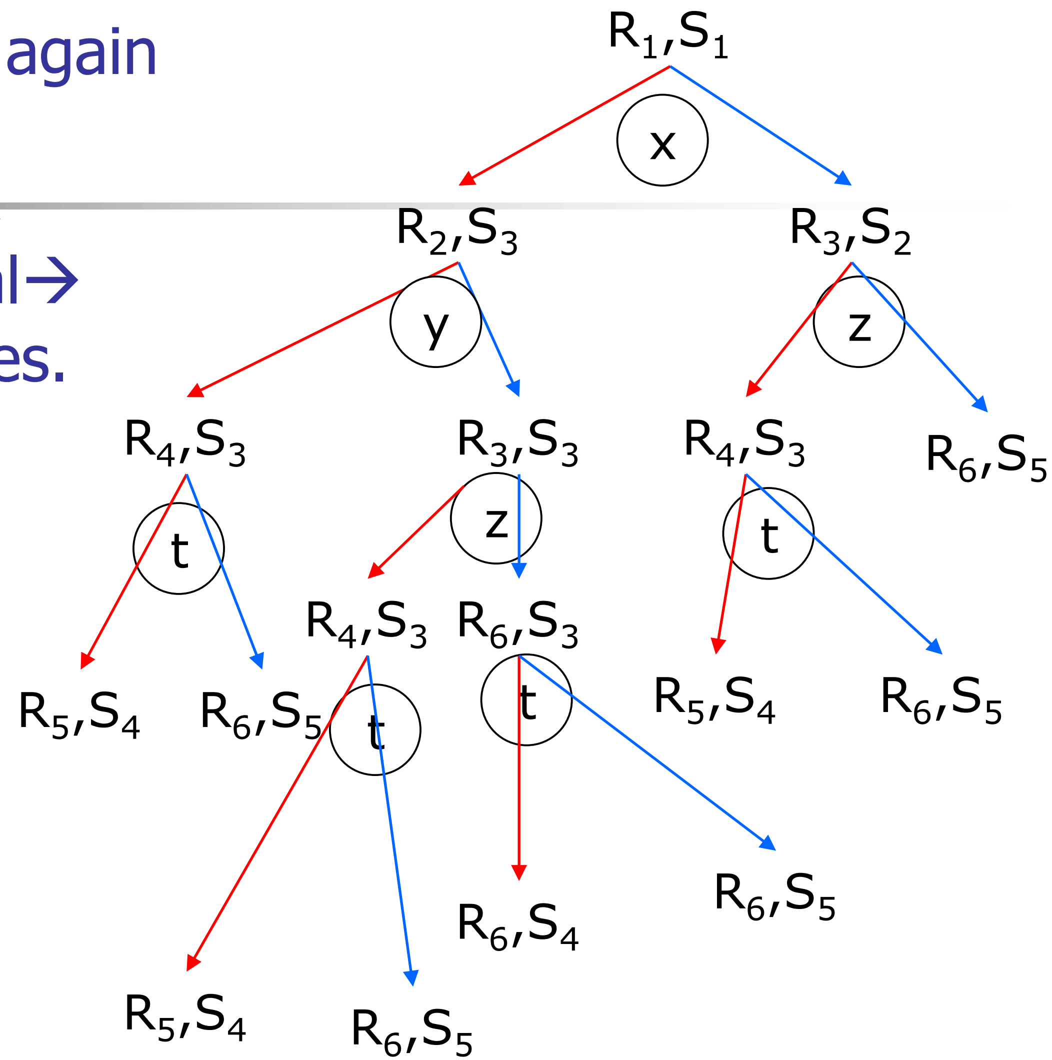
Now compute f#g recursively:

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 $r_f\#r_g$ and put the result.
- If both roots are same node (say x), then create a low edge to low(r_f)#low(r_g), and a high edge to high(r_f)#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 low(r_f)#g and a high edge to high(r_f)#g. The symmetric case is handled similarly.
- We reduce.

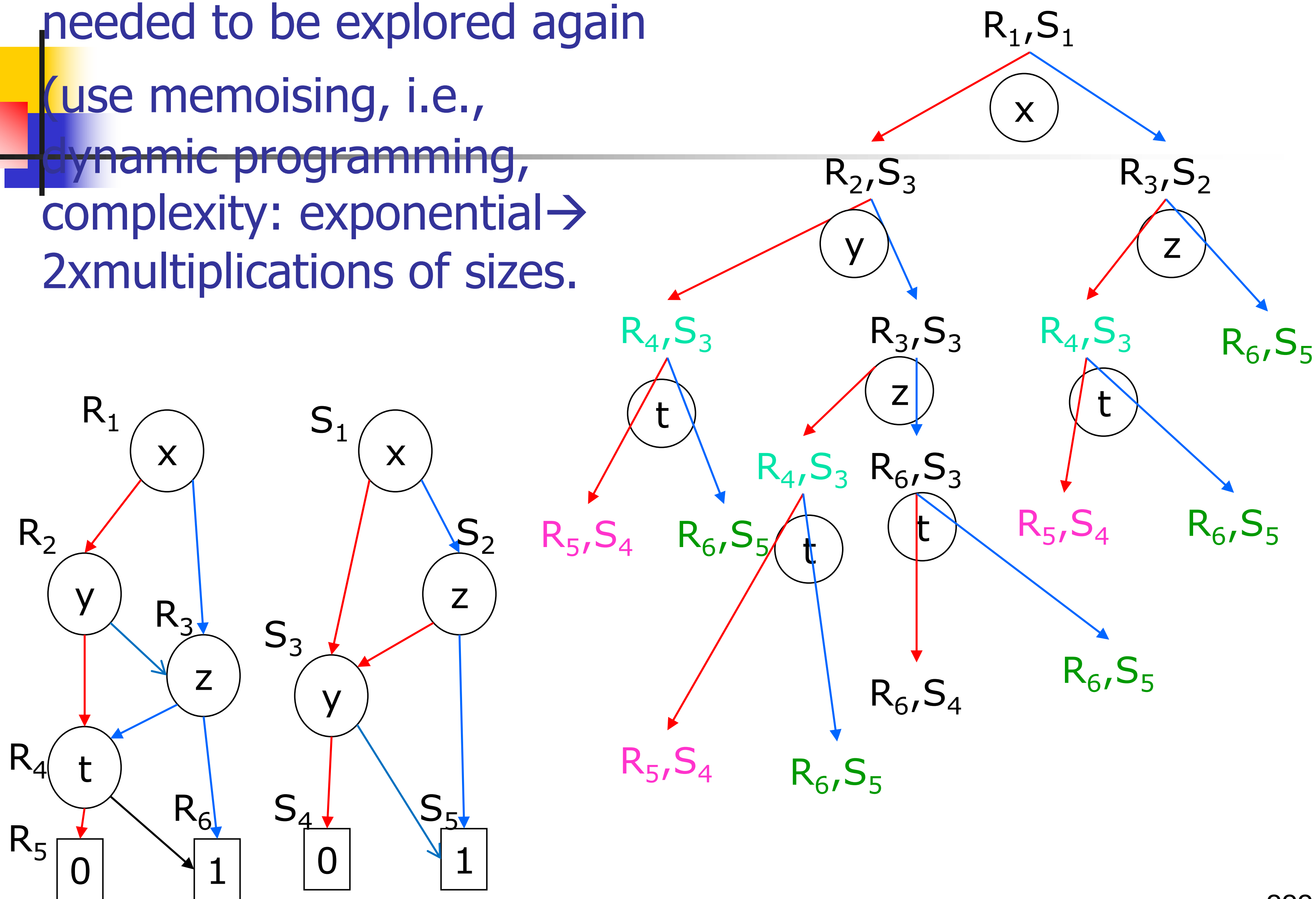
Same subgraphs are not needed to be explored again (use memoising, i.e., dynamic programming, complexity: exponential→ 2xmultiplications of sizes.





Same subgraphs are not needed to be explored again

(use memoising, i.e., dynamic programming, complexity: exponential→ 2xmultiplications of sizes.



Symbolic Model Checking

- Characterize CTL formulas using fixpoints.
- $AF\varphi = \varphi \bigvee AX\ AF\varphi$
- **$EF\varphi = \varphi \vee EX\ EF\varphi \Rightarrow \mu Z.\varphi \vee EX\ Z$**
- $AG\varphi = \varphi /\backslash AX\ AG\varphi$
- **$EG\varphi = \varphi \wedge EX\ EG\varphi \Rightarrow \nu Z.\varphi \wedge EX\ Z$**
- $A\varphi U\psi = \psi \bigvee (\varphi /\backslash AX\varphi U\psi)$
- **$E\varphi U\psi = \psi \vee (\varphi \wedge EX\varphi U\psi) \Rightarrow \mu Z.\psi \vee (\varphi \wedge EXZ)$**
- $A\varphi R\psi = \psi /\backslash (\varphi \bigvee AX\varphi R\psi)$
- **$E\varphi R\psi = \psi \wedge (\varphi \vee EX\varphi R\psi) \Rightarrow \nu Z.\psi \wedge (\varphi \vee EXZ)$**

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 $\tau(Z)=\varphi \vee$ EX Z

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

Model checking $\mu Z\ \tau$ (least fixed point)
For example, $\tau = \varphi \vee EX\ Z$
For formulas with main operator $\vee$.

procedure Check LFP (τ)

Q :=False; Q' := τ(Q) ;

while Q $\neq$ Q' do

Q := Q';

Q' := τ(Q) ;

end while

return(Q)

Model checking $\mu Z\ \tau$ (Greatest fixed point)
For example, $\tau = \psi \wedge (\varphi \vee EX\ Z)$
For formulas with main operator $\wedge$.

procedure Check GFP (τ)

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

while $Q \neq Q'$ do

$Q := Q'$;

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

end while

return(Q)