LTL Model Checking
(Ch. 4 LN)

Wishnu Prasetya

wishnu@cs.uu.nl
www.cs.uu.nl/docs/vakken/pv

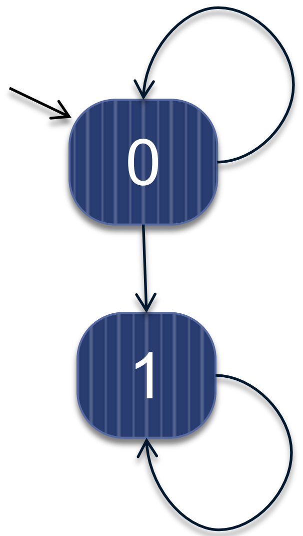
Overview

- This pack :
  - Abstract model of programs
  - Temporal properties
  - Verification (via model checking) algorithm
  - Concurrency

Run-time properties

- Hoare triple : express what should hold when the program terminates.
- Many programs are supposed to work continuously
  - They should be “safe”
  - They should not dead lock
  - No process should starve
- Linear Temporal Logic (LTL)
  - Originally designed by philosophers to study the way that time is used in arguments
    Based on a number of operators to express relation over time: “next”, “always”, “eventually”
  - Belong to the class of modal logics
  - Brought to Computer Science by Pnueli, 1977.

Finite State Automaton/Machine



- Abstraction of a real program
- Choices
  - What information do we want to put in the states?
  - In the arrows?
- How to model execution? → a path through the FSA, staring at an initial state.
  - Does it have to be finite?
  - Do we need a concept of “acceptance” ?
- These choices influence what you can express, and how you can verify properties over executions.

FSA

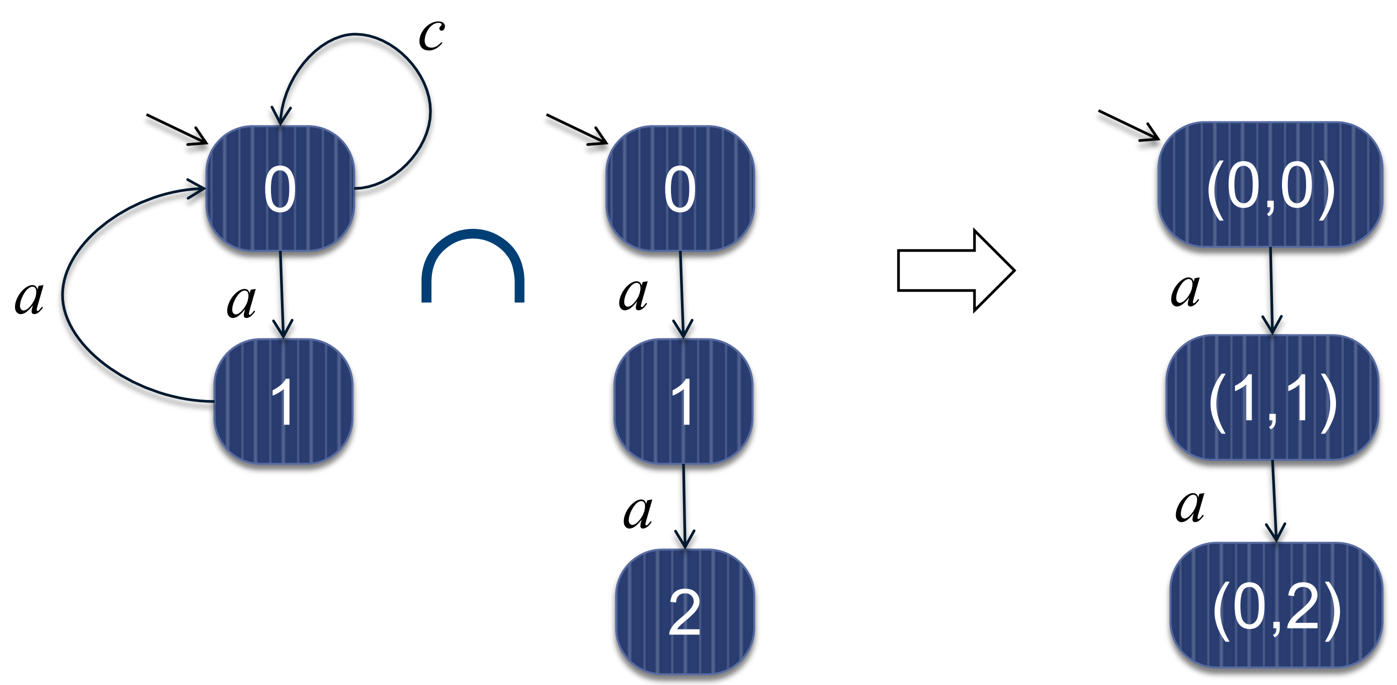
- Described by (S, I, Σ, R)

  - S : the set of possible states
  - $I \subseteq S$: set of possible initial states
  - Σ : set of labels, to decorate the arrows.
    - They model possible actions.
  - R : $S \rightarrow \Sigma \rightarrow$ **pow**(S), the arrows
    - $R(s,a)$ is the set of possible next-states of a if executed on s
    - non-deterministic

Program compositions can be modeled by operations over FSA

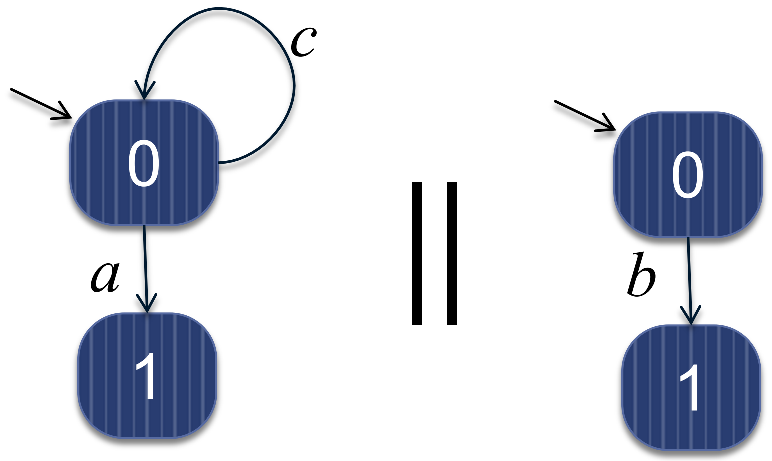
- We assume actions to be **atomic**.
- M_1 ; M_2
  - connect "terminal states" of M_1 to M_2's initial states.
- $M_1 \cap M_2$
  - only do executions that are possible in both
- $M_1 \,||\, M_2$
  - model parallel execution of M_1 and M_2

Intersection



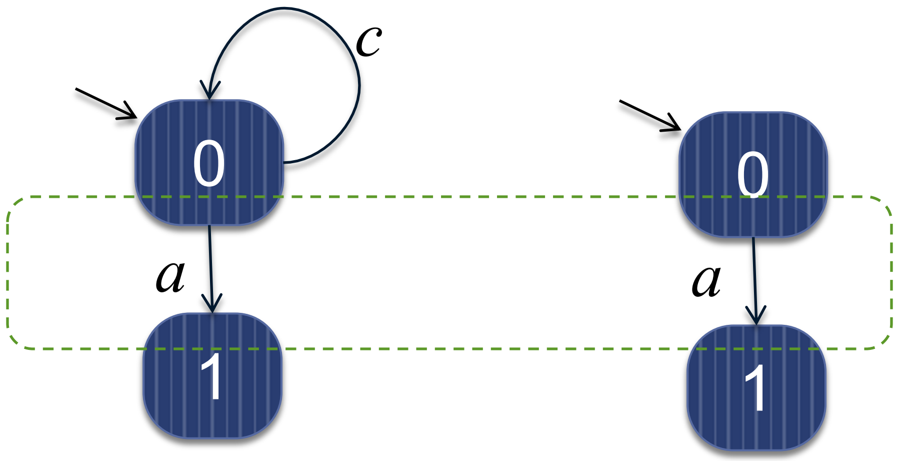
- Not something you typically do in real programs
- A useful concept for verification → later.

Parallel composition



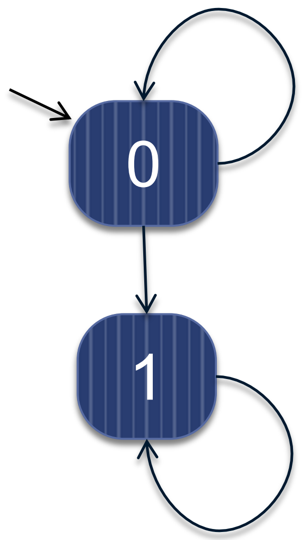
- Suppose M_1 and M_2 has **no common action**.
- Their parallel composition is basically the “full product” of M_1 and M_2 .
- So,if each component has n number of states, constructing || over k components produces an automaton with n^k states.

Parallel composition with synchronized actions



- Suppose we require that any action a ∈ $\Sigma_1 \cap \Sigma_2$ has to be executed **together** (synchronizedly) by both automata.

Let’s add labels



Consider these set of “labels”, *Prop* = { isOdd *x*, *x*>0 }. The labeling is describe by this function V:

$\boldsymbol{V}(0) = \{ \text{isOdd } x \}$
$\boldsymbol{V}(1) = \{ \text{isOdd } x,\ x>0 \}$

So far, the only things we know about the states is that they differ from each other. Let’s extend the available information with propositions.

Kripke Structure

- A finite automaton (S, s_0, R, $Prop$, V)

  - S : the set of possible states, with s_0 the initial state.
  - R : $S \rightarrow$ **pow**(S), the arrows
    - $R(s)$ is the set of possible next-states from s
    - non-deterministic
  - $Prop$: set of *atomic* propositions
    - abstractly modeling state properties.
  - V : $S \rightarrow$ **pow**($Prop$), labeling function
    - $a \in V(s)$ means a holds in s, else it does *not* hold.
  - No concept of accepting states.

Prop

- It consists of *atomic* propositions.
- We'll require them to be non-contradictive. That is, for any subset Q of Prop :

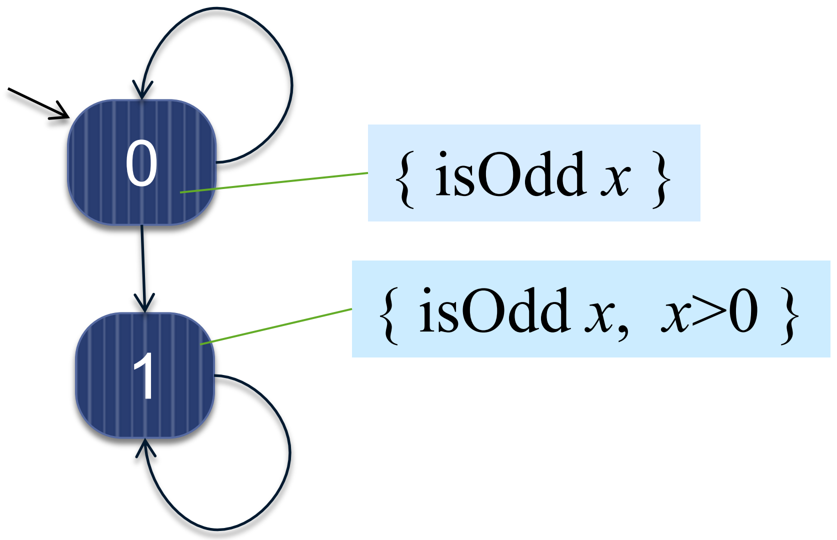
$$\bigwedge Q \;\wedge\; \bigwedge\{\neg p \mid p \in \mathrm{Prop} \wedge p \notin Q\}$$

  is satisfiable. Else you may get inconsistent labeling.
- This is the case if they are *independent* of each other.
- Example:
  - *Prop* = { $x>0$, $y>0$ } is ok.
  - *Prop* = { $x>0$, $x>1$ } is not ok. E.g. the subset { $x>1$ } is inconsistent.

Execution

- An *execution* is a path through your automaton, starting from an initial state.
- Let’s focus on properties of infinite executions
  - All executions are assumed **infinite**
  - Extend each “terminal” state (states with no successor) in the original Krikpe with a stuttering loop.
- This induces an ‘*abstract*’ execution: Nat$\rightarrow$**pow**(*Prop*)
  - infinite *sequence of* the *set of propositions* that hold along that path.
  - We’ll often use the term execution and abstract execution interchangbly.

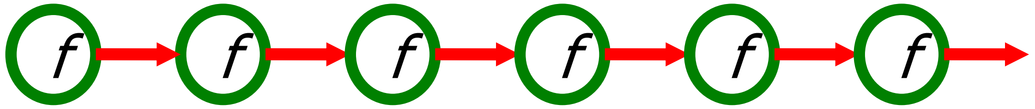
Example



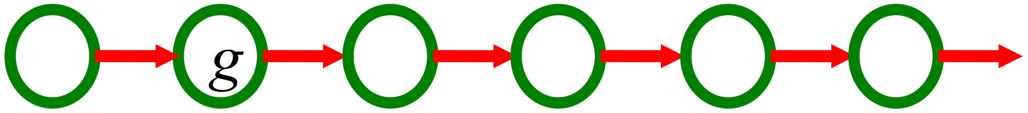
Exec. :	0	0	1	1	…
Abs-exec:	{isOdd x} ,	{isOdd x},	{isOdd x, x>0},	{isOdd x, x>0}	, …

LTL, informal meaning

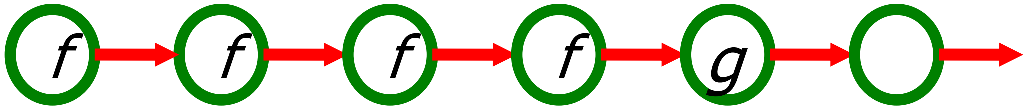
$\Box f$ // always *f*



X *g* // next *g*



f U *g* // *f* holds until *g*



You can combine operators

- $\Box(p \rightarrow (\text{true } \mathsf{U}\ q))$ // whenever p holds, eventually q will hold
- $p\ \mathsf{U}\ (q\ \mathsf{U}\ r)$
- $\text{true}\ \mathsf{U}\ \Box p$ // eventually stabilizing to p
- $\text{true}\ \mathsf{U}\ \Box\neg\ \Box\neg p$ // eventually p will hold infinitely many often

Syntax

- φ ::= *p* // atomic proposition from *Prop*

  | $\neg\varphi$ | $\varphi \wedge \psi$ | X φ | φ U ψ

- Derived operators:
  - $\varphi \vee \psi = \neg(\neg\varphi \wedge \neg\psi)$
  - $\varphi \rightarrow \psi = \neg\varphi \vee \psi$
  - □ , ◊ , W , …

- Interpreted over abstract executions.

Defining the meaning of temporal formulas

- First we'll define the meaning wrt to a single abstract execution. Let Π be such an execution:
  - $\Pi, i \;\; |== \;\; \varphi$
  - $\Pi \;\; |== \;\; \varphi \quad = \quad \Pi, 0 \;\; |== \;\; \varphi$
- If *P* is a Kripke structure,

  $P \; |== \; \varphi$ means that φ holds on all abstract executions of *P* that start from P's initial state

Meaning

- Let Π be an (abstract) execution.
  - Π, i |== p = $p \in \Pi(i)$ // $p \in Prop$
  - Π, i |== $\neg\varphi$ = not (Π, i |== φ)
  - Π, i |== $\varphi \land \psi$ = Π, i |== φ
    and
    Π, i |== ψ

Meaning

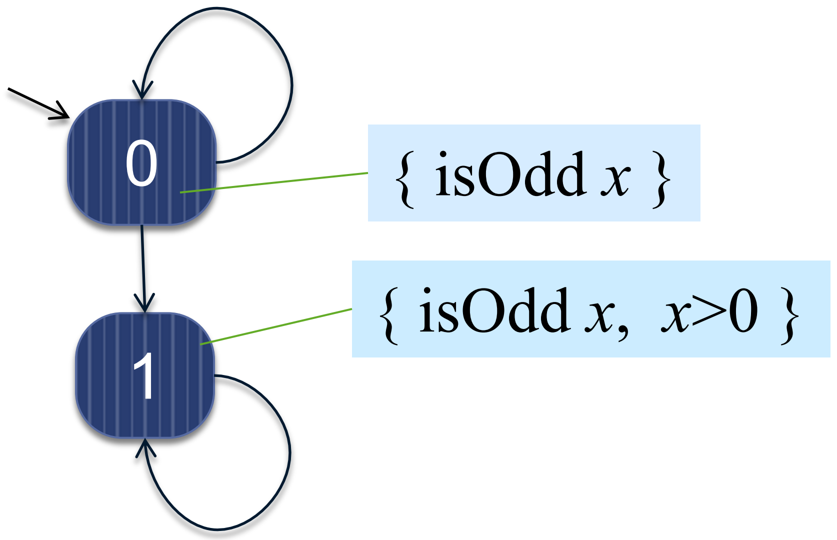
- $\Pi, i \models X\varphi \quad = \quad \Pi, i+1 \models \varphi$

- $\Pi, i \models \varphi\ U\ \psi \quad = \quad$ there is a $j \geq i$ such that $\Pi, j \models \psi$

  and

  for all h, $i \leq h < j$, we have $\Pi, h \models \varphi$.

Example



Consider Π : {isOdd x} , {isOdd x}, {isOdd x, x>0}, {isOdd x, x>0} , ...

Π |== isOdd x U x>0

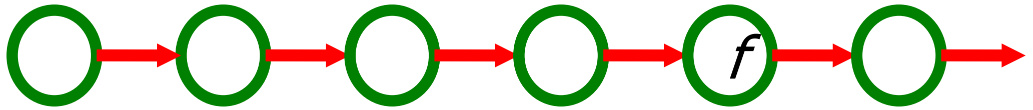
Is this a valid property of the FSA?

Derived operators

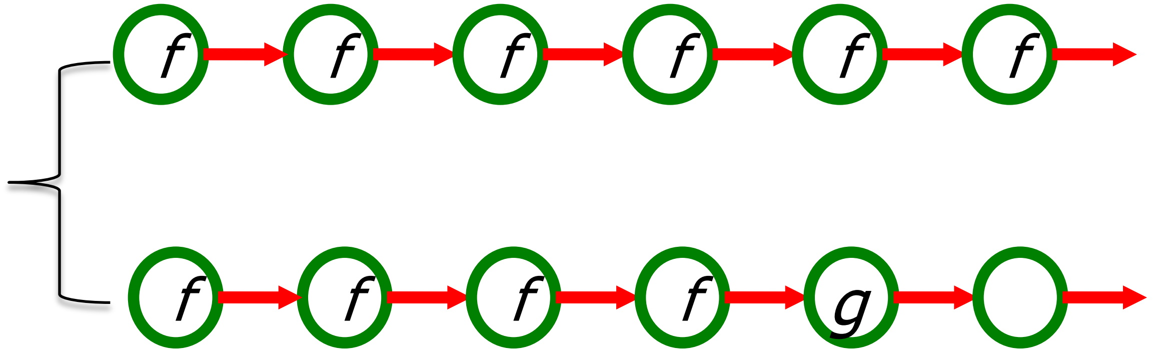
Eventualy	$\Diamond\varphi$	=	$\text{true}\ \mathrm{U}\ \varphi$
Always	$\Box\varphi$	=	$\neg\Diamond\neg\varphi$
Weak until	$\varphi\ \mathrm{W}\ \psi$	=	$\Box\varphi \vee (\varphi\ \mathrm{U}\ \psi)$
Release	$\varphi\ \mathrm{R}\ \psi$	=	$\psi\ \mathrm{W}\ (\varphi \wedge \psi)$

Some derived operators

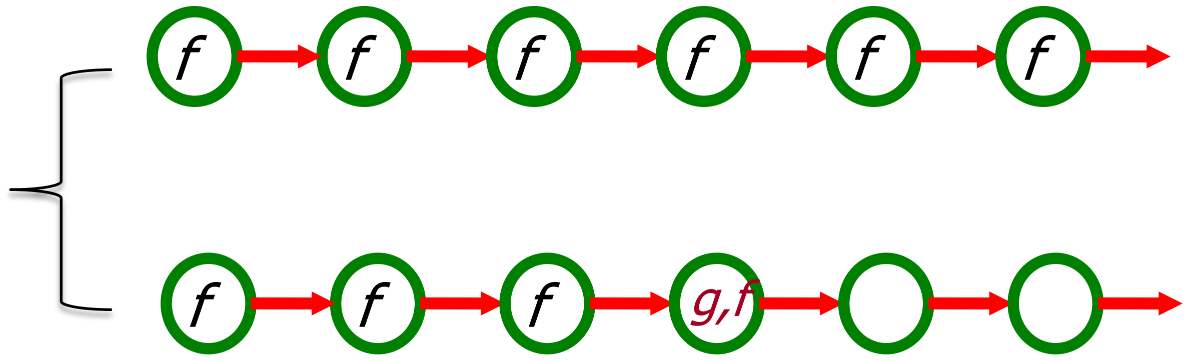
<> *f*   // eventually *f*



f W *g*   // weak until



g R *f*   // releases



Past operators

- Useful, e.g. to say: if P is doing something with *x*, then it must have asked a permission to do so.

- "previous"
  $\Pi, i \models\!= Y\varphi$ = φ holds in the previous state

- "since"
  $\Pi, i \models\!= \varphi\ S\ \psi$ = $\Pi, j \models\!= \psi$ for some $j \leq i$, and for all $j < k \leq i$ we have $\Pi, j \models\!= \varphi$

- Unfortunately, not supported by SPIN.

Ok, so how can I verify $M \models\!= \varphi$?

- We can't directly check all executions → infinite (in two dimensions).
- Try to prove it directly using the definitions?
- We'll take a look another strategy…
- First, let's view abstract executions as *sentences*.

  View *M* as a sentence-generator. Define:

$$\boxed{L(M) \quad = \quad \{ \underset{\uparrow}{\Pi} \quad | \quad \Pi \text{ is an abs-exec of } M \}}$$

these are sentences over ***pow****(Prop)*

Representing φ as an automaton …

- Let φ be the temporal formula we want to verify.
- Suppose we can construct automaton B_φ that ‘accepts’ exactly those infinite sentences over **pow**(*Prop*) for which φ holds.
- So B_φ is such that :

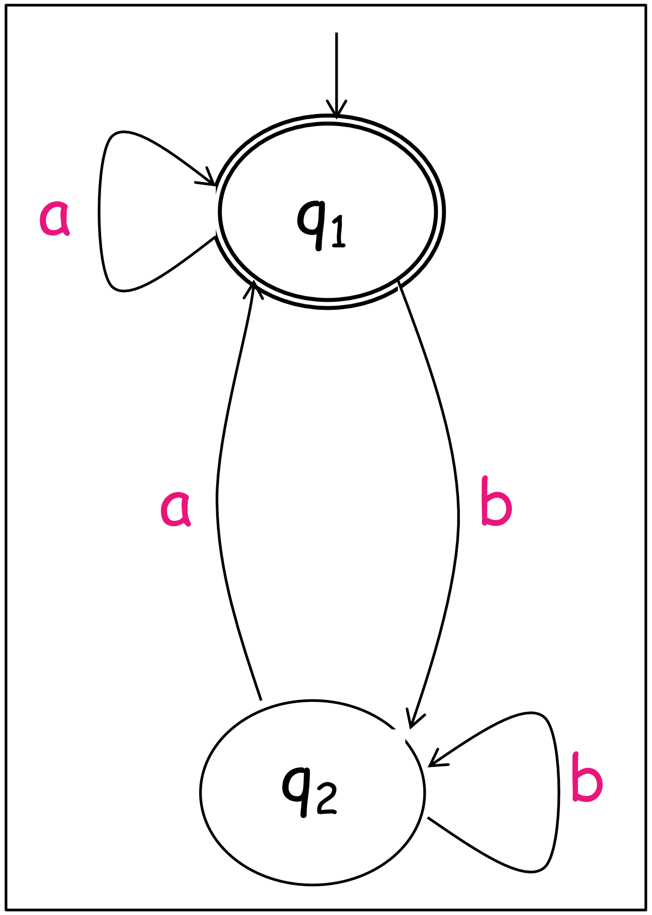
$$\boxed{L(B_\varphi) \;=\; \{ \Pi \;\mid\; \Pi \models\!= \varphi \}}$$

Re-express as a language problem

- Well, $M \mid\!\!== \varphi$ iff
  - There is no $\Pi \in L(M)$ where φ does not hold.
  - In other words, there is no $\Pi \in L(M)$ that will be accepted by $L(B_{\neg\varphi})$.
- So:

$$\boxed{M \mid\!\!== \varphi \quad \text{iff} \quad L(M) \cap L(B_{\neg\varphi}) = \emptyset}$$

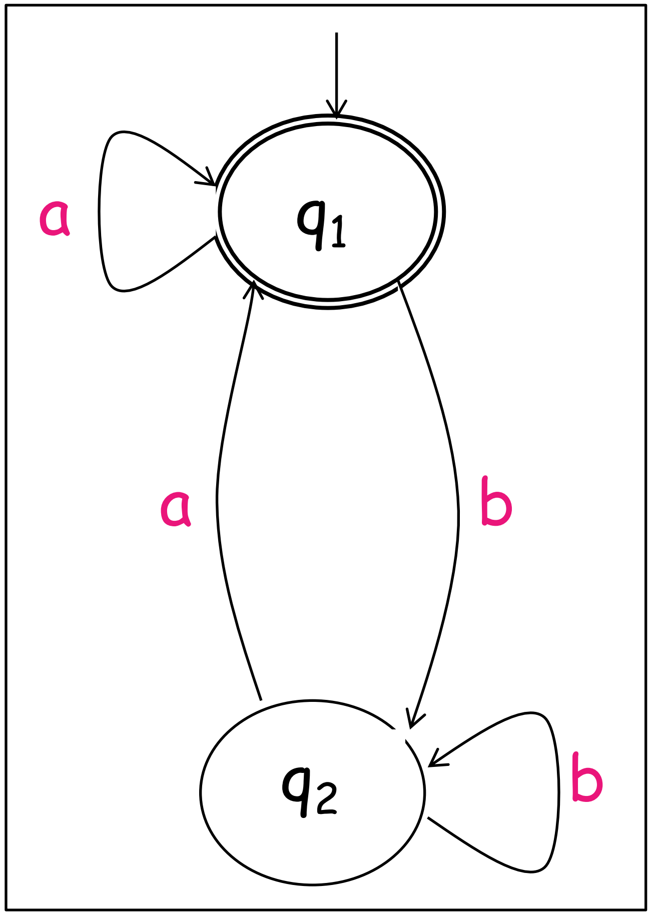
Automaton with "acceptance"



- So far, all paths are accepted. What if we only want to accept some of them?
- Add *acceptance states*.
- Accepted sentence:
  "*aba*" and "*aa*" is accepted
  "*bb*" is not accepted.

- But this is for finite sentences. For infinite ...?

Buchi Automaton



- Pass an acceptance state infinitely many times.
- Examples

"*ababa*" → not an infinite

"*ababab*…" → accepted

"*abbbb*…" → not accepted!

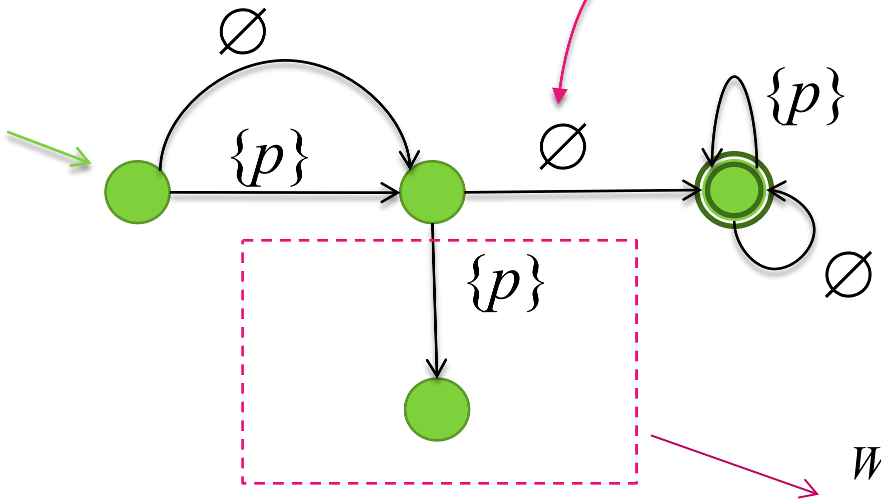
Expressing temporal formulas as Buchi

Use **pow**(*Prop*) as the alphabet Σ of arrow-labels.

Example: $\neg \mathrm{X} p \quad (= \mathrm{X}\neg p)$

We'll take *Prop* = { *p* }

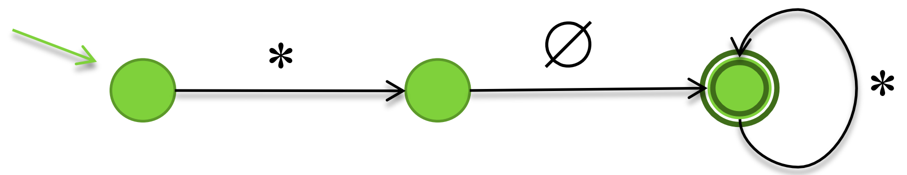
Indirectly saying that p is false.



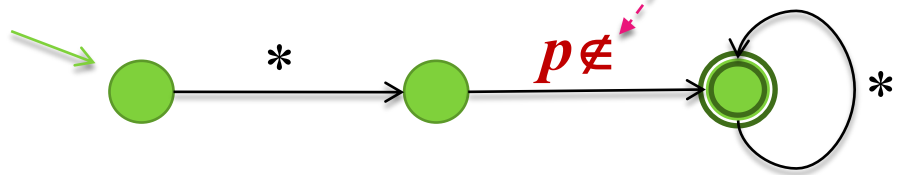
We can drop this, since we only need to (fully) cover accepted sentences.

To make the drawing less verbose...

$\neg \mathbf{X} p$, using $Prop = \{p\}$



$\neg \mathbf{X} p$, using $Prop = \{p,q\}$



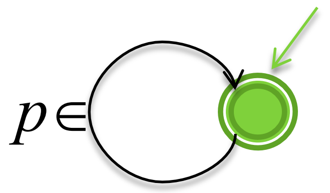
So we have 4 subsets.

$p\notin$ *Stands for all subsets of Prop that do not contain p; thus implying "p does not hold".*

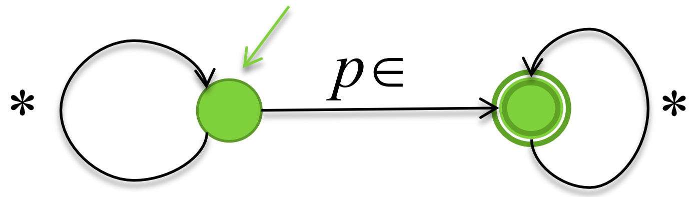
$p\in$ *Stands for all subsets of Prop that contain p; thus implying "p holds".*

Always and Eventually

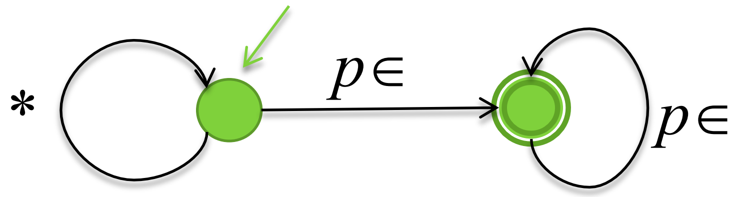
$[]p$



$<>p$

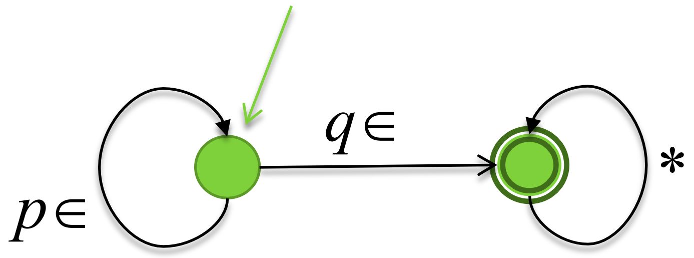


$<>[]p$

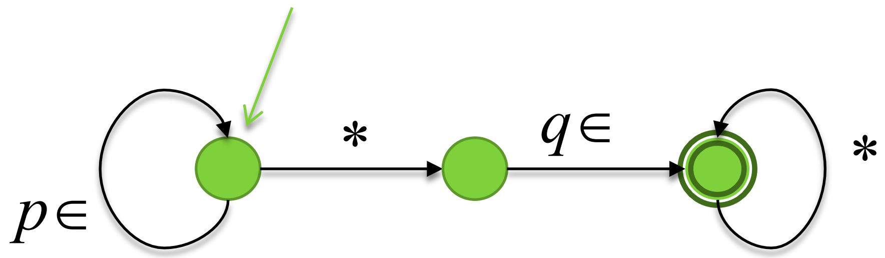


Until

$p \; \mathrm{U} \; q$:



$p \; \mathrm{U} \; \mathrm{X}q$:



Not Until

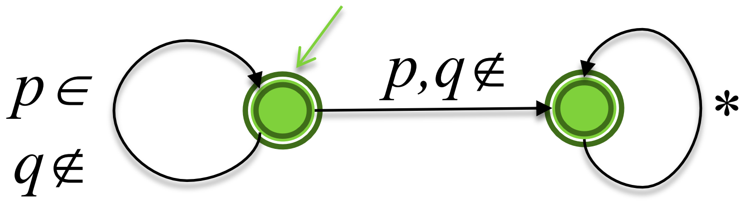
Formula: $\neg\,(\,p\ \mathbf{U}\ q\,)$

Note first these properties:

$\neg(p\ \mathbf{U}\ q) \quad = \quad p \wedge \neg q\ \ \mathbf{W}\ \ \neg p \wedge \neg q \quad \xrightarrow{} \quad = \ \neg q\ \ \mathbf{W}\ \ \neg p \wedge \neg q$

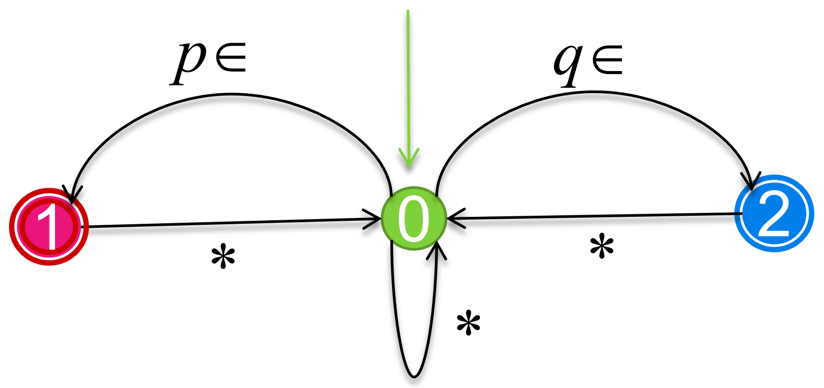
$\neg(p\ \ \mathbf{W}\ \ q) \quad = \quad p \wedge \neg q\ \ \mathbf{U}\ \ \neg p \wedge \neg q \quad \xrightarrow{} \quad = \ \neg q\ \ \mathbf{U}\ \ \neg p \wedge \neg q$

(also generally when *p*,*q* are LTL formulas)



Generalized Buchi Automaton

[]<>p ∧ []<>q



Sets of accepting states: **F** = { {1} , {2} }

which is different than just F = { 1, 2 } in an ordinary Buchi.

Every GBA can be converted to BA.

Difficult cases

- How about nested formulas like:

  $(\mathbf{X}p)\ \mathbf{U}\ q$
  $(p\ \mathbf{U}\ q)\ \mathbf{U}\ r$

  Their Buchi is not trivial to construct.

- Still, any LTL formula can be converted to a Buchi. SPIN implements an automated conversion algorithm; unfortunately it is quite complicated.

Check list

$$M \models \varphi \quad \text{iff} \quad L(M) \cap L(B_{\neg\varphi}) = \varnothing$$

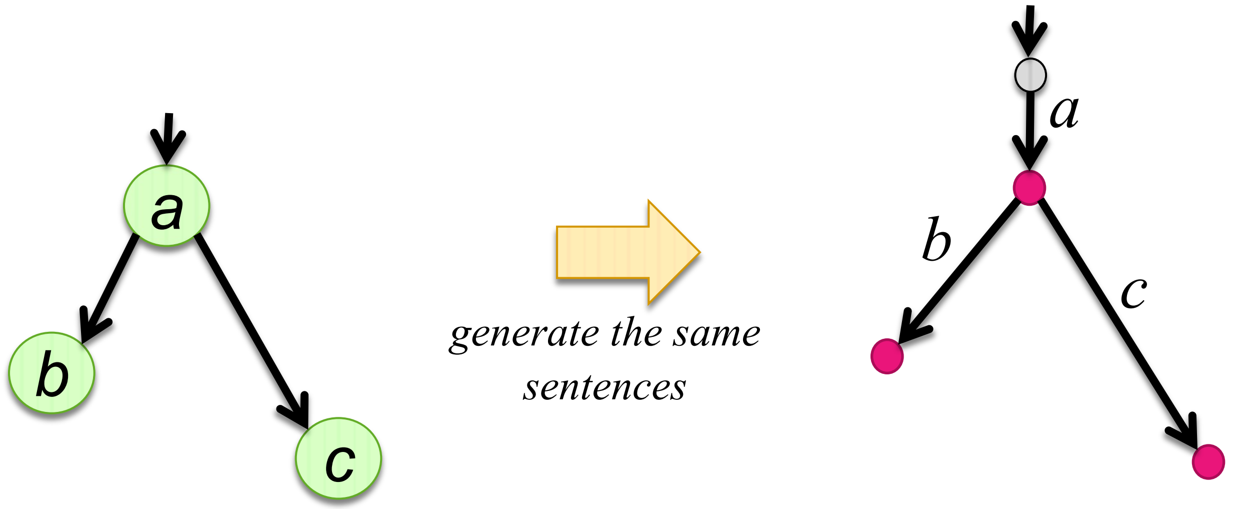
1. How to construct $B_{\neg\varphi}$? → Buchi ✓

2. We still have a mismatch, because *M* is a Kripke structure!
   - Fortunately, we can easily convert it to a Buchi.

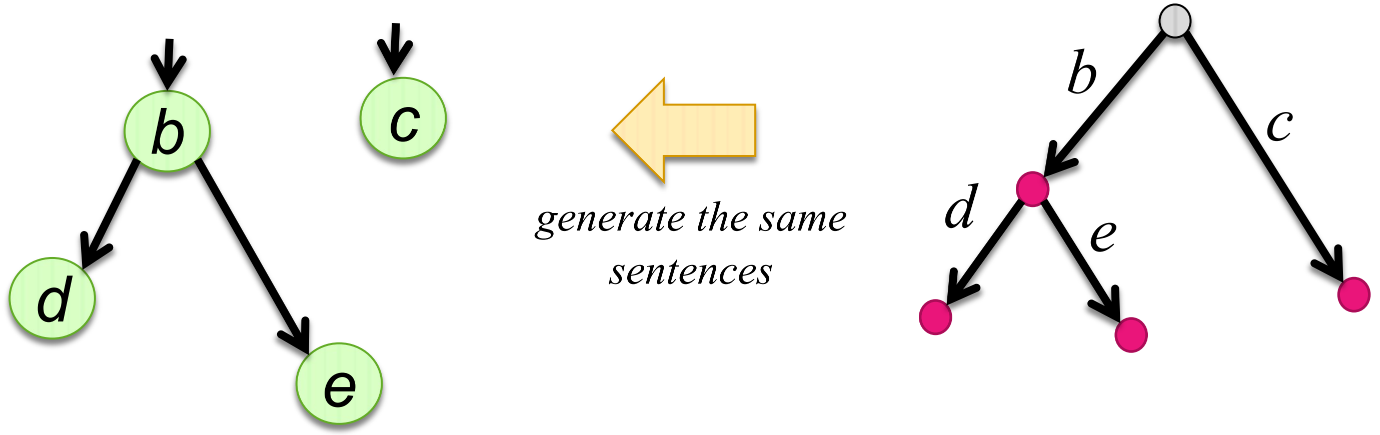
3. We still have to construct the intersection.

4. We still to figure out a way to check emptiness.

Label on state or label on arrow...

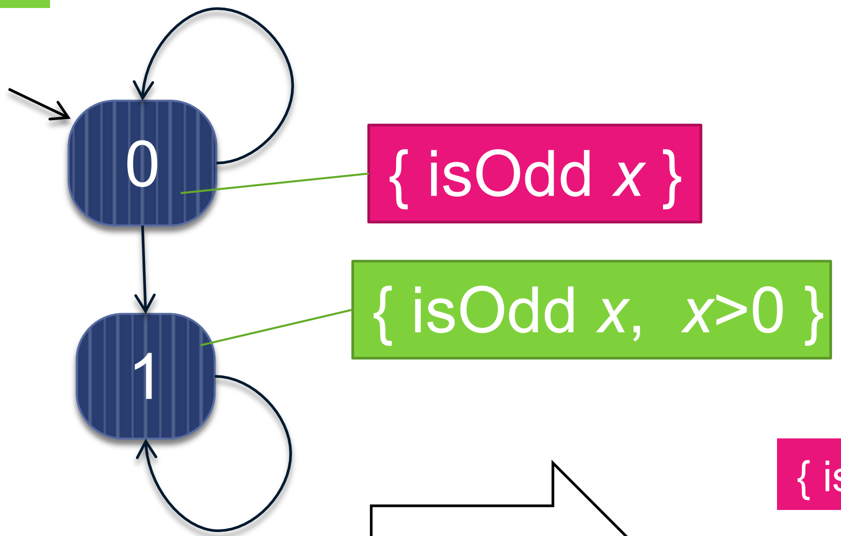


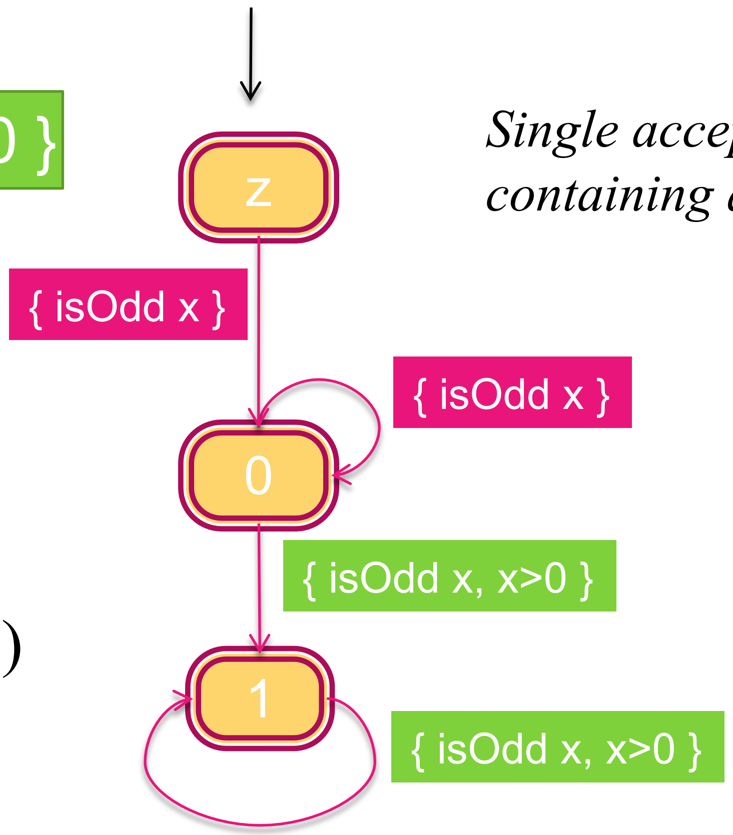
generate the same sentences



generate the same sentences

Converting Kripke to Buchi





Single accepting set F, containing all states.

Generate the same (infinite) sentences!

Computing intersection

- Rather than directly checking:

The Buchi version of Kripke M ☺

$$L(B_M) \cap L(B_{\neg\varphi}) = \emptyset$$

We check:

$$L(B_M \cap B_{\neg\varphi}) = \emptyset$$

We already discuss this! Execution over such an intersection is also called a "lock-step" execution.

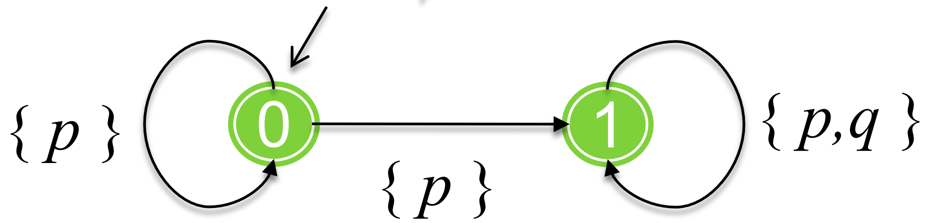
Intersection

- Two buchi automata *A* and *B over the same alphabet*
  - The set of states are respectively S_A and S_B.
  - starting at respectively $s0_A$ and $s0_B$.
  - Single accepting set, respectively F_A and F_B.
  - F_A is assumed to be S_A
- A $\cap$ B can be thought as defining lock-step execution of both:
  - The states : $S_A \times S_B$, starting at $(s0_A, s0_B)$
  - Can make a transition only if A and B can *both* make the corresponding transition.
  - A single acceptance set F; (s,t) is accepting if $t \in F_B$.

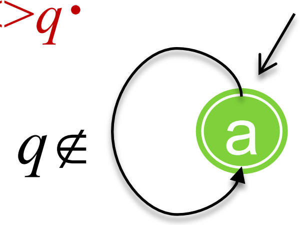
Constructing Intersection, example

p : isOdd x
q : x>0

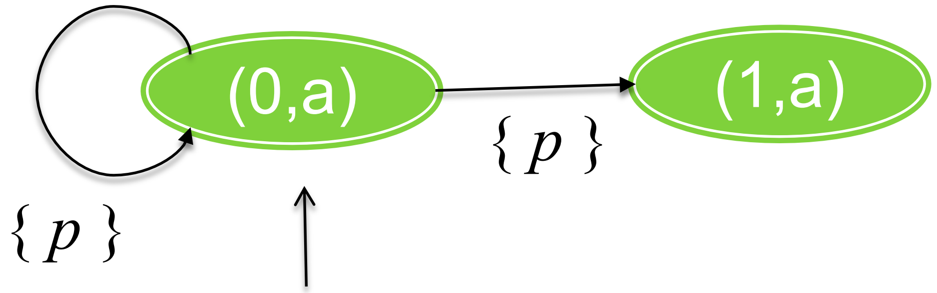
B_M:



$B_{\neg<>q}$:



$B_M \cap B_{\neg<>q}$:



Verification

- Sufficient to have an algorithm to check if L(*C*) = $\emptyset$, for the intersection-automaton *C*.

L(*C*) $\neq \emptyset$ iff there is a finite path from *C*'s initial state to an accepting state *f* , followed by a cycle back to *f* .

- So, it comes down to a cycle finding in a finite graph! Solvable.

- The path leading to such a cycle also acts as your counter example!

Approaches

- View $C = B_M \cap B_{\neg\varphi}$ as a directed graph.
  Approach 1 :
  1. Calculate all strongly connected component (SCCs) of *C* (e.g. with Tarjan) .
  2. Check if there is an SCC containing an accepting state, reachable from *C*'s initial state.

- Approach 2, based on Depth First Search (DFS); can be made *lazy* :
  - the full graph is constructed as-we-go, as you search for a cycle.
  - Importantly, if *M* represents a parallel composition $P_1 \mid\mid P_2 \mid\mid \ldots$, this means that we can lazily construct B_M.

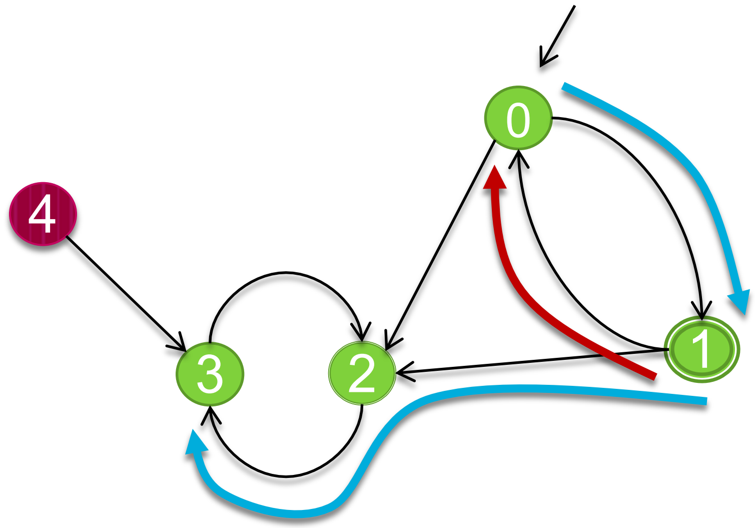
DFS-approach (SPIN)

- DFS is a way to traverse a graph :

```
DFS($u$)  {
    if ($u \in visited$)  return ;
    $visited.add(u)$ ;
    for ($s \in next(u)$)  DFS($s$) ;
}
```

- This will visit all reachable nodes. You can already use this to check “state assertions”.

Example



Adding cycle detection

```
DFS($u$)  {
    if ($u \in visited$) return ;
    $visited.add(u)$ ;
    for each ($s \in next(u)$)  {
        if ($u \in accept$)  {
            $visited2 = \emptyset$ ;
            $checkCycle\ (u,s)$ ;
        }
        DFS($s$ ) ;
    }
}
```

checkCycle is another DFS

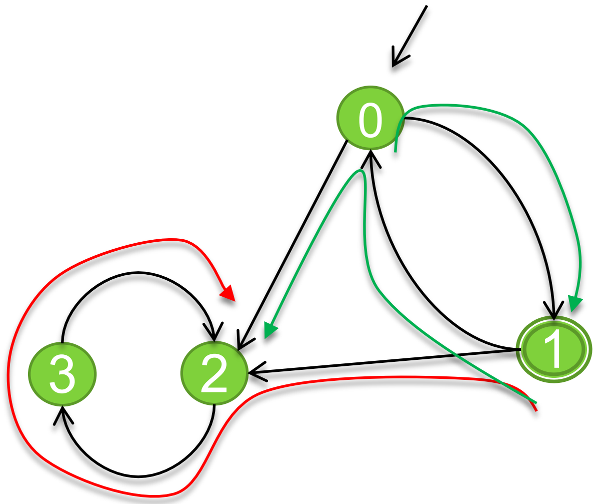
```
checkCycle(root,s) {

    if (s = root) throw CycleFound ;

    if ( s ∈ visited2 ) return ;
    visited2.add(s) ;
    for each (t ∈ next(s))
        checkCycle(root, t) ;
}
```

Can be extended to keep track of the path leading to the cycle → counter example.
See Lecture Notes.

Example



checkCycle(1,2)

root

the node currently being processed

Tweak: lazy model checking

- Remember that automaton to explore is $C = B_M \cap B_{\neg\varphi}$
- In particular, B_M can be huge if $M = P_1 \,||\, P_2 \,||\, \ldots$
- Can we construct C lazily?
- Benefit : if a cycle is found (verification fails), effort is not wasted to first construct the full *C.*
- Of course if φ turns out to be valid, then C will in the end fully constructed.
- How to deal with concrete states (rather than abstract states a la Kripke) ?

Lazily constructing the intersection

- Assume first that P is just a single process.
- Only need to change this in the DFS :

```
for each ($s \in$ next($u$))  ….
    if ($u \in$ accept)
```

"**next**" of the intersection automaton $C = B_M \cap B_{\neg\varphi}$

- Each state of C is of type $S_M \times S_{\neg\varphi}$.
- To check $(s_1,s_2) \in next_C(u_1,u_2)$, we check if there is a label L, such that: $s_1 \in next_M(u_1,L) \wedge s_2 \in next_{\neg\varphi}(u_2,L))$
- $(u_1,u_2) \in accept_C \equiv u_2 \in accept_{\neg\varphi}$

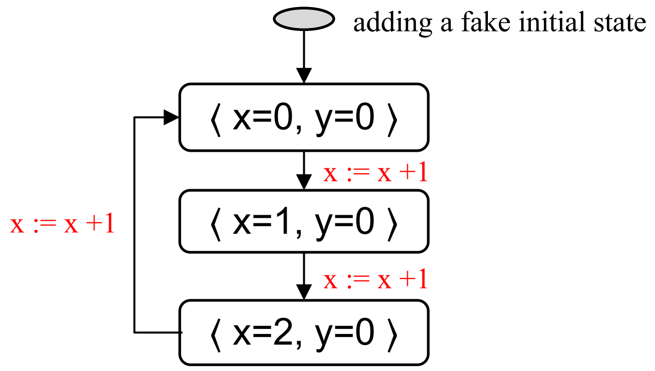
Dealing with concrete states

Consider a concrete program *Prg* :

var x = 0 , y = 0 ;

repeat (x := x+1 **mod** 3)

- A concrete state of Prg is a vector ⟨x,y⟩
- FSM *Prg* representing the program, in terms of concrete states:



- Given a concrete state s and a predicate p, let **eval**(p,s) denote the value of p when evaluated on s (so it is either true or false).
- So, to check if there is a successor of *s* such that a label $L \subseteq Prop$ holds, we check instead, if there is an (atomic) transition α in *Prg* such that for all $p \in L$, **eval**(p,α(s)) is true, and for all $q \notin L$, **eval**(q,α(s)) is false.

Dealing with concrete states

- So, if M is a program with concrete states, e.g. checking this in the DFS:

  $$(s_1,s_2) \in next_C(u_1,u_2)$$

  comes down to checking if there is an atomic transition α of M and a label L of $B_{\neg\varphi}$ such that $s_2 \in next_{\neg\varphi}(u_2,L)$), and :
  - for all propositions p∈L, **eval**(p, $\alpha(u_1)$) = true
  - for all propositions q∈Prop/L, **eval**(q, $\alpha(u_1)$) = false

What if $M = P_1 \parallel P_2 \parallel \ldots$?

- We discussed the parallel composition of FSAs, e.g. :

  $$\begin{tikzcd} o \arrow[r, "\alpha"'] & o \arrow[r, phantom, "\parallel"] & o \arrow[r, "\beta"'] & o \end{tikzcd}$$

  Note that here α and β represent actions.

- Literally applying such parallel composition on Buchi automata makes less sense because the labels are properties rather than actions.

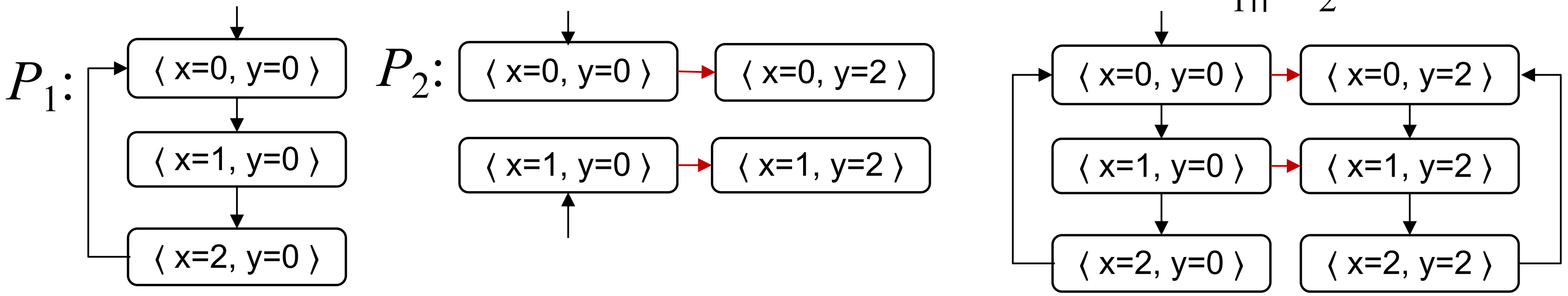
Example constructing $P_1 \parallel P_2$

Consider a concrete program $Prg = P_1 \parallel P_2$:

var x = 0 , y = 0 ;

P_1 : **repeat** (x := x+1 **mod** 3)

P_2 : (x≠2) ; y := 2



- In the above example, we explicitly construct the concrete state FSM of $P_1 \parallel P_2$
- We can instead construct $P_1 \parallel P_2$ lazily as we construct the intersection automaton with $B_{\neg\varphi}$

What if $M = P_1 \,||\, P_2$?

- E.g. checking this in the DFS:

  $$(s_1,s_2) \in next_C(u_1,u_2)$$

  now comes down to checking if there is an atomic transition α of either P_1 or P_2, and a label L of $B_{\neg\varphi}$ such that $s_2 \in next_{\neg\varphi}(u_2,L)$), and :
  - for all propositions p∈L, **eval**(p, $\alpha(u_1)$) = true
  - for all propositions q∈Prop/L, **eval**(q, $\alpha(u_1)$) = false

Fairness

Consider this concurrent system :

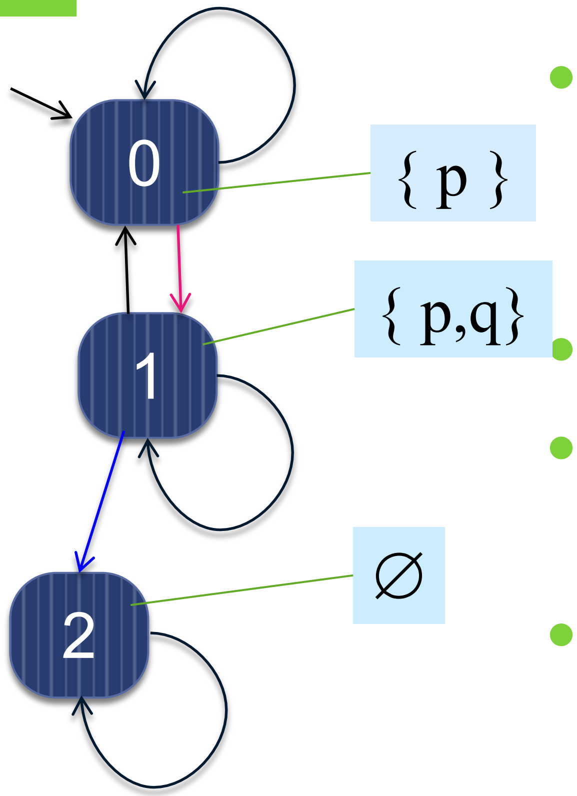
execution blocks if false

P { **while** true **do** *x:=(x+1)%3* } || Q { (*x*==0) ; print *x* }

Is it possible that print *x* is ignored forever?

- You may want to assume some concept of fairness. There are various possibilities. Importantly, it has to be reasonable.
  - *Weak fairness* : any action that is persistently enabled will eventually be executed.
  - *Strong fairness* : any action that is kept recurrently enabled (but not necessarily persistently enabled) will eventually be executed.
- Imposing fairness mean: when verifying *M* |== φ, we only need to verify φ wrt fair executions of M.
  - A *fair execution* : an execution respecting the assumed fairness condition.

Verifying properties under fairness



- If a fairness assumption can be expressed with some LTL formula *F*, we can instead verify *M* |== *F* $\rightarrow$ φ
- No need for a special algorithm!
- E.g. weak fairness F_1on the red arrow:
  - $\Box\ (\Box(p\wedge\neg q)\ \ \rightarrow \Diamond(p\wedge q))$
- Strong fairness F_2 on the blue arrow:
  - $\Box\ (\Box\Diamond(p\wedge q) \rightarrow \Diamond(\neg p\wedge\neg q))$
- Example of property to verify: $F_1 \wedge F_2 \rightarrow\ \Diamond(\neg p\wedge\neg q)$

Conclusion

- We can use FSAs to abstractly model concurrent programs.
- We can use LTL to express run-time properties: safety and progress.
- The model checking algorithm is thorough.
- Rather than FSAs with atomic predicates, you can imagine letting the FSAs to have concrete states.
  - You can then model check real programs.
  - The FSAs could be very large, but we can bound the input domains, and the depth of the search, → bounded model checking.
  - Combination with testing: to construct an execution so that M behaves as φ, model-check this: $L(B_M \cap B_\varphi) = \emptyset$