LTL Model Checking

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

Abstract Model

- Temporarily back off from concrete SPIN level, and look instead at a more abstract view on the problem.
- Model a "program" as a <u>finite</u> automaton.
- More interested in "run-time properties" (as opposed to e.g. pre/post conditions):
  - Whenever R receives, the value it receives is never 0.
  - If a holds, then eventually b

To keep in mind: the term "model" is heavily overloaded...

(but generally, a model simply means a simplified version of a real thing)

Automaton for explaining SPIN

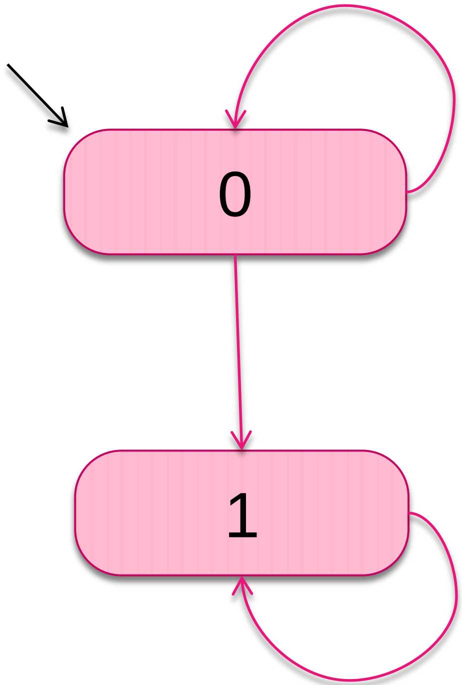
Automaton SPIN constructs during verification

SPIN's Promela model

UML model

Real Program

Finite State Automaton



Many variations, depending on modeling purposes:

- Single or multiple init states
- With our without accepting states
- With or without label on arrows, or on states
- How it executes

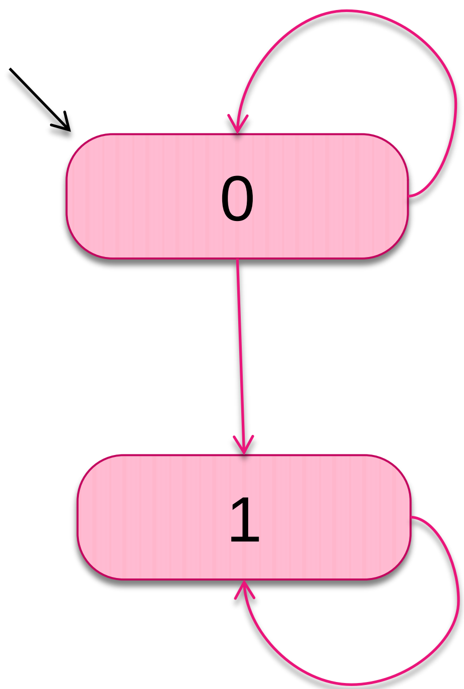
__Execution__ : a path through the automaton, starting with an initial state.

State?

- Concrete state of a program $\rightarrow$ too verbose.
- More abstract $\rightarrow$ the values of program's variables
  - How SPIN works
- Even more abstractly, through a fixed set of propositions
  - Define your set of propositions
  - Specify which propositions hold on which states
  - How I will explain SPIN

Example

Convention: this implies that $x>0$ does ***not*** *hold in state 0.*



With *Prop* = { isOdd x, $x>0$ }

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

Abstract: we can't say anything about properties which were not taken in *Prop*.

Kripke Structure

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

  - S : the set of possible states, with s_0 the initial state.
  - R : the arrows
    - $R(s)$ gives the set of possible next-state from s
    - non-deterministic
  - $Prop$: set of *atomic* propositions
  - V : labeling function

  $a \in V(s)$ means a holds in s, else it does *not* hold.

Prop

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

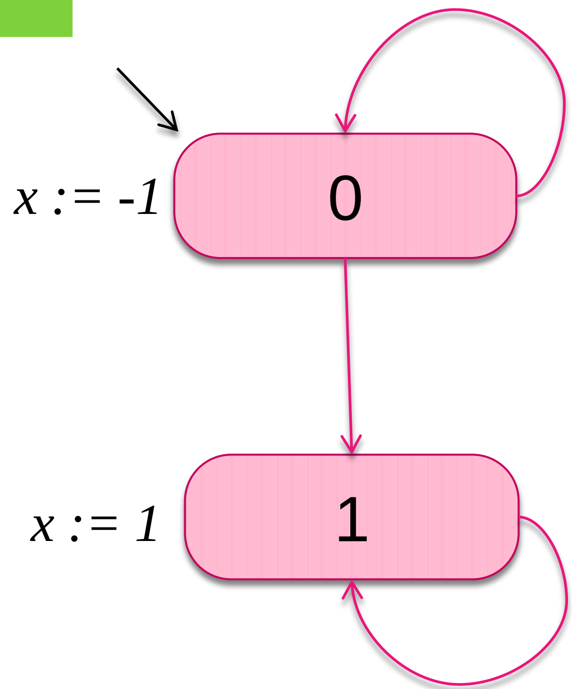
  $$(\bigwedge Q) \wedge (\bigwedge\{ \neg p \mid p \notin Q\})$$

  is satisfiable. Else you may get inconsistent labeling.
- Example:
  - *Prop* = { x>0 , y>0 } is ok.
  - *Prop* = { x>0 , x>1 } is not ok. E.g. the subset { x>1 } is inconsistent.

Modelling execution

- Recall: an *execution* is a path through your automaton.
- Limit to infinite executions → to simplify discussion.
- This induces what we will call an ‘abstract’ execution, which is a *sequence of set of propositions* that hold along that path.
- Overloading the term “execution”...

Example



Prop = { isOdd *x*, *x*>0 }

V(0) = { isOdd x }
V(1) = { isOdd x, x>0 }

Consider execution: 0, 0, 1, 1, ...

It induces abs-exec:

{isOdd *x*} , {isOdd *x*}, {isOdd *x*, *x*>0}, {isOdd *x*, *x*>0} , ...

Note that it is a sequence over power(*Prop*).

Properties

- Recall that we want to express "run-time" properties → we'll use "temporal properties" from Linear Temporal Logic (LTL)

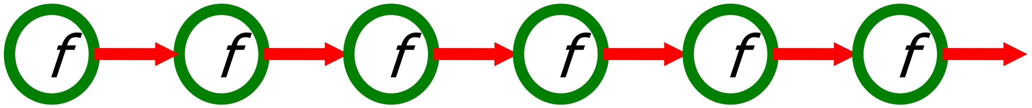
- Originally designed by philosophers to study the way that time is used in natural language arguments ☺

  Based on a number of operators to express relation over time: "next", "always", "eventually"

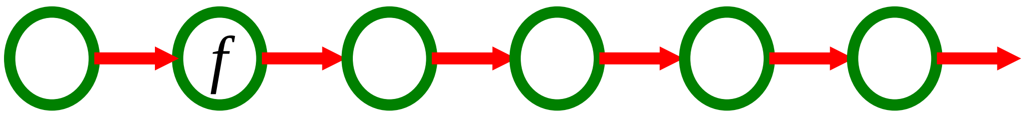
- Brought to Computer Science by Pnueli, 1977.

Informal meaning

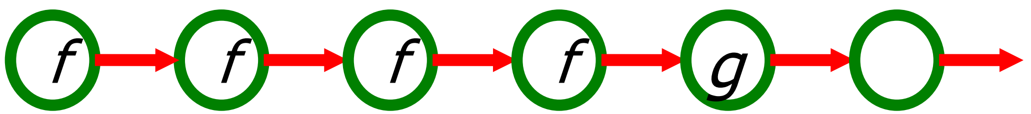
[] *f* // always *f*



X *f* // next *f*



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



Example

```
active proctype S () {
   do
   ::  { c!x ;
         passed: x++ }
   od
}
```

```
active proctype R () {
   do
    ::  c?y
   od
}
```

- [] (y==x || y==x-1)
- [] (true **U** S@passed)
- Could have been expressed as a Monitor process and progress-labels.
- But e.g. [](a U q) cannot be expressed with progress labels.

Quite expressive! For example…

- $[] \, (p \rightarrow (\text{true} \; \mathcal{U} \; q))$ // whenever p holds, eventually q will hold
- $p \; \mathcal{U} \; (q \; \mathcal{U} \; r)$
- $\text{true} \; \mathcal{U} \; []p$ // eventually stabilizing to p

Let's do this more formally...

- 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:
  - $\Pi, i \ \ |== \ \ \varphi$
  - $\Pi \ \ |== \ \ \varphi \qquad = \qquad \Pi, 0 \ \ |== \ \ \varphi$
- If *P* is a Kripke structure,

P $|==$ φ means that φ holds on all abs. executions of *P*

Meaning

- Let Π be an (abstract) execution.
  - $\Pi, i \mid== p \quad = \quad p \in \Pi(i)$ // $p \in Prop$
  - $\Pi, i \mid== \neg\varphi \quad = \quad$ not $(\Pi, i \mid== \varphi)$
  - $\Pi, i \mid== \varphi \wedge \psi \quad = \quad \Pi, i \mid== \varphi$
    and
    $\Pi, i \mid== \psi$

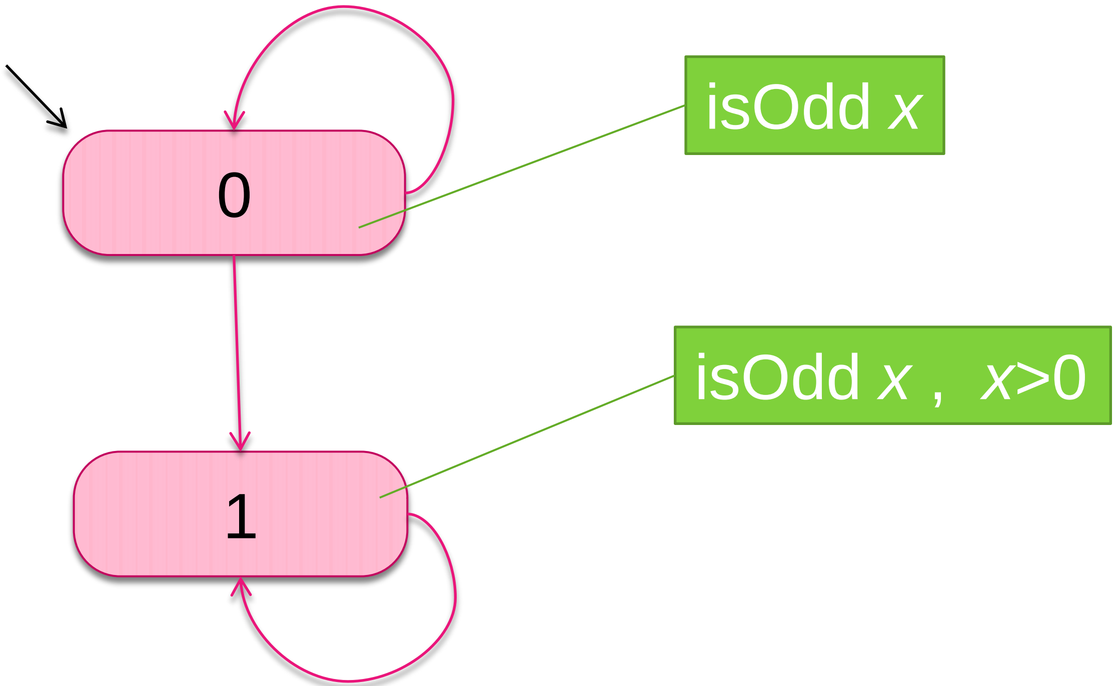
Meaning

- $\Pi, i \ |{=}{=} \ X\varphi \quad = \quad \Pi, i+1 \ |{=}{=} \ \varphi$
- $\Pi, i \ |{=}{=} \ \varphi \ U \ \psi \quad = \quad$ there is a $j \geq i$ such that $\Pi, j \ |{=}{=} \ \psi$

  and

  for all h, $i \leq h < j$, we have $\Pi, h \ |{=}{=} \ \varphi$.

Example



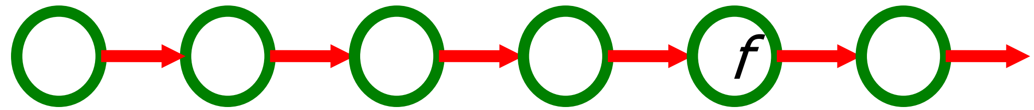
Consider abs-exec Π : {isOdd x} , {isOdd x}, {isOdd x, x>0}, {isOdd x, x>0} , ...

Π |== isOdd x U x>0

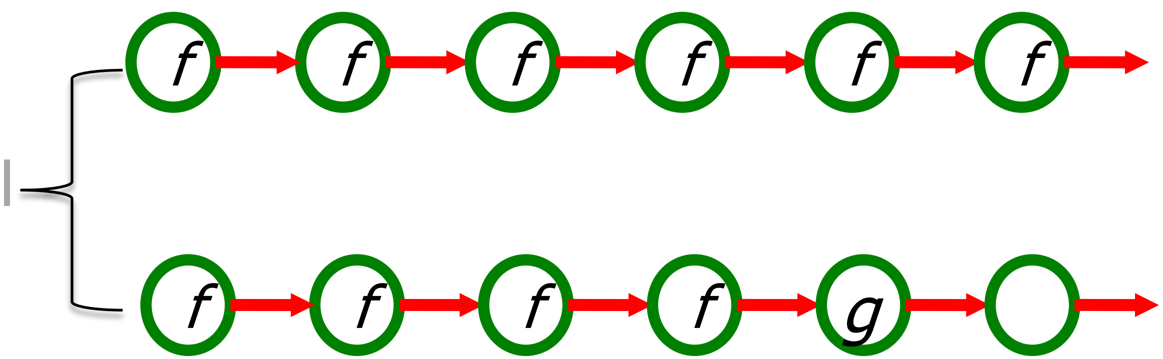
However, this is not a valid property of the program.

Some derived operators

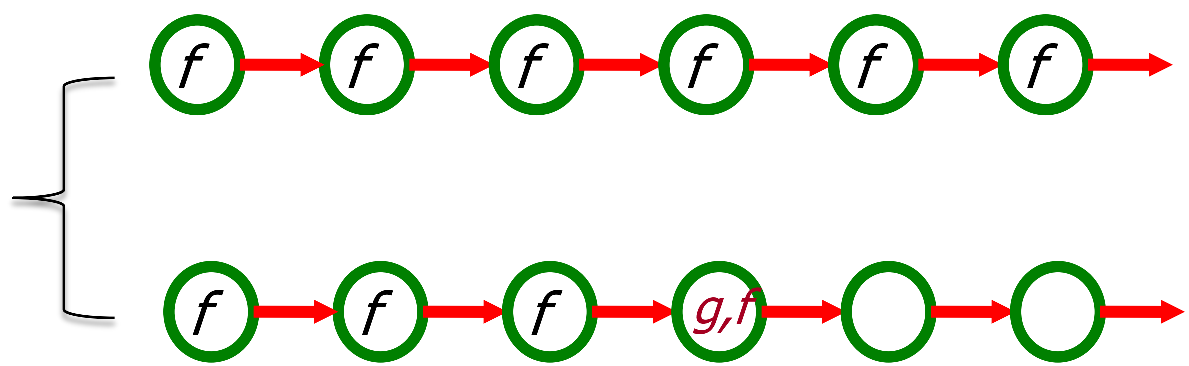
<> *f* // eventually *f*



f W *g* // weak until



g R *f* // releases



Derived operators

$<>\varphi \quad = \quad true \; U \; \varphi$

$[]\varphi \quad = \quad \neg<>\neg\varphi$

$\varphi \; W \; \psi \quad = \quad []\varphi \; \vee \; (\varphi \; U \; \psi)$

$\varphi \; R \; \psi \quad = \quad \psi \; W \; (\varphi \wedge \psi)$

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

- "since"
  $\Pi, i \models\!= \varphi \; \mathrm{S} \; \psi$ = ψ held in the past, and since that to now φ holds

- Unfortunately, not supported by SPIN.

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

- 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 P as a sentence-generator. Define:

$$L(P) \quad = \quad \{ \Pi \ \mid \ \Pi \text{ is an abs-exec of } P \}$$

these are sentences over pow(erProp)

Representing φ as an automaton …

- Let φ be the temporal formula we want to verify.

- Suppose we can construct automaton A_φ that 'accepts' exactly those infinite sentences over power(*Prop*) for which φ holds.

- So A_φ is such that :

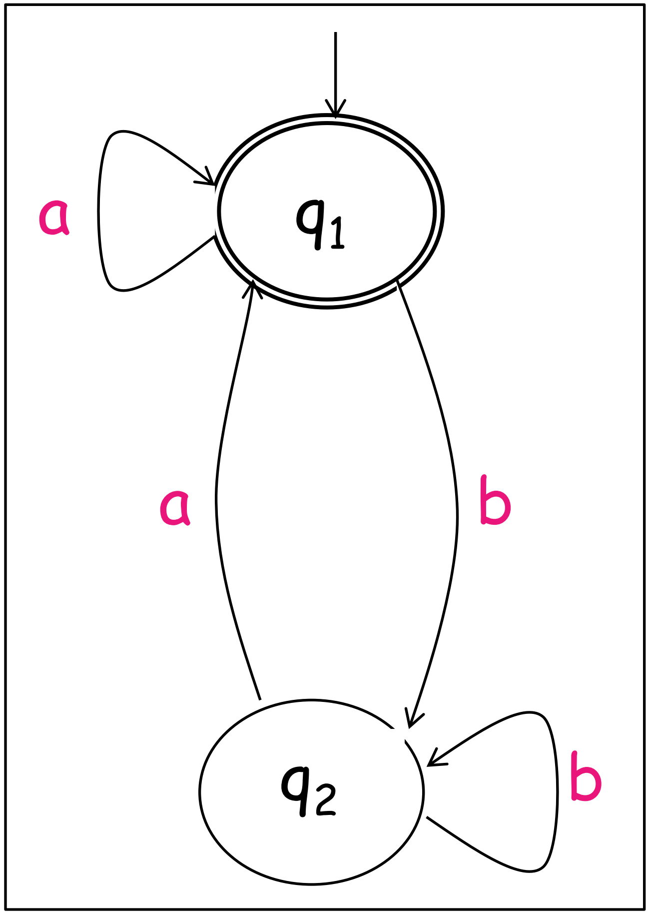
$$\boxed{L(A_\varphi) = \{ \Pi \mid \Pi \models \varphi \}}$$

Re-express as a language problem

- Well, $P \models\!= \varphi$ iff
  - There is no $\Pi \in L(P)$ which will violate φ.
  - In other words, there is no $\Pi \in L(P)$ that will be accepted by $L(A_{\neg\varphi})$.
- So:

$$\boxed{P \models\!= \varphi \quad \text{iff} \quad L(P) \cap L(A_{\neg\varphi}) = \emptyset}$$

Automaton for accepting sentences



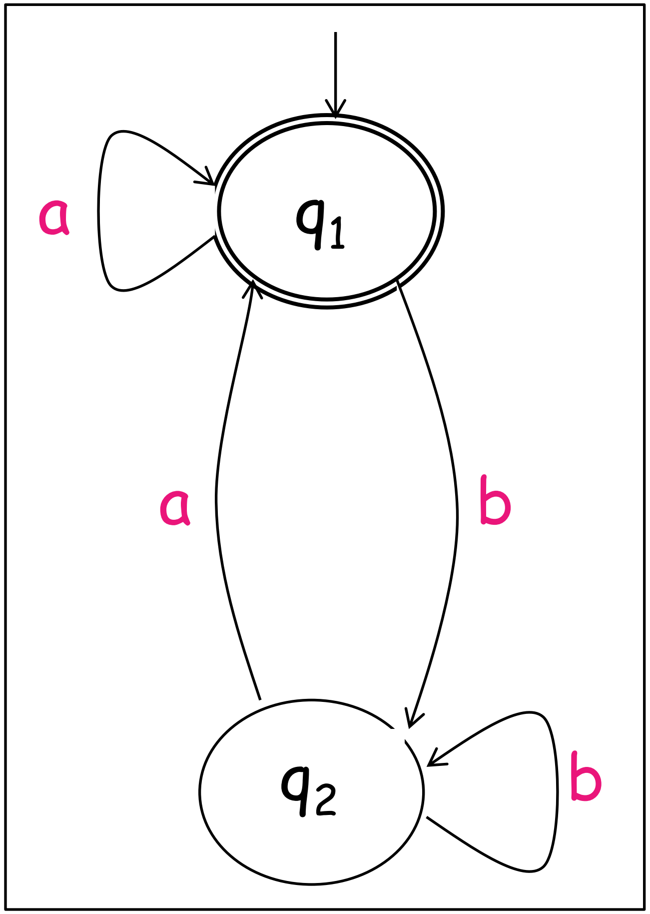
- Add *acceptance states*.
- Accepted sentence:

  "*aba*" and "*aa*" is accepted
  "*bb*" is <u>not</u> accepted.

- But this is for finite sentences.

  For infinite ...?

Buchi Automaton



- Just different criterion for acceptance
- Examples

"*abab*" → not an infinite

"*ababab*…" → accepted

"*abbbb*…" → not accepted!

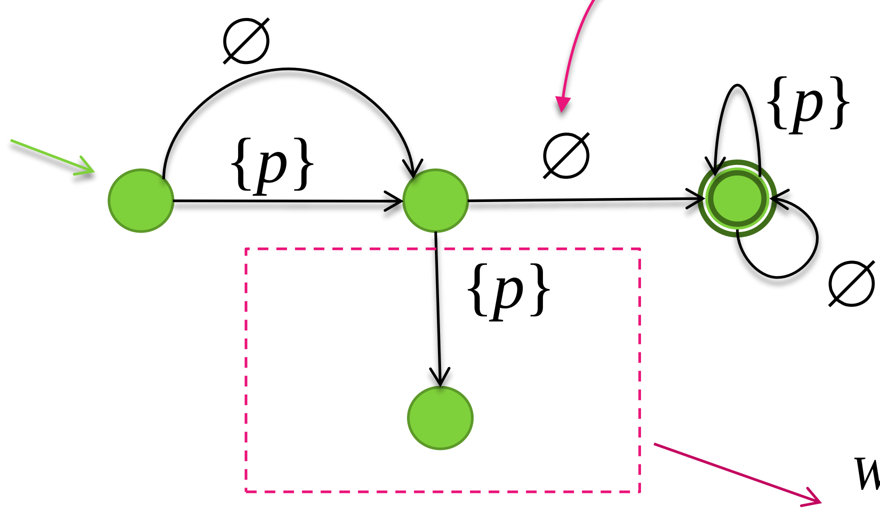
Expressing temporal formulas as Buchi

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

Example: $\neg$X*p* (= X$\neg$*p*)

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

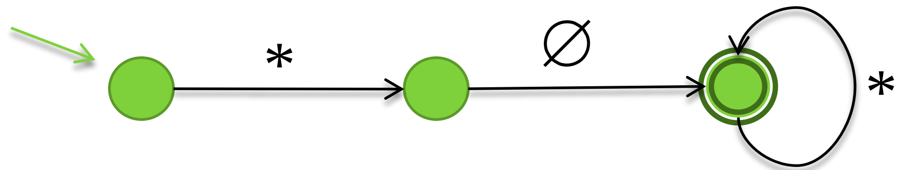
Indirectly saying that p is false.



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

To make the drawing less verbose...

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

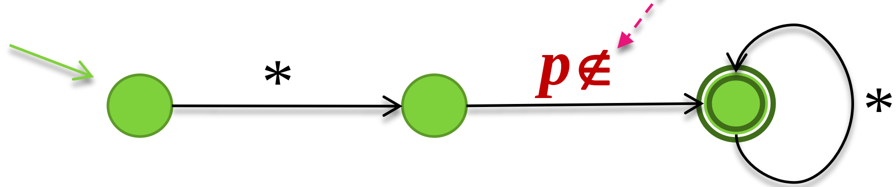


So we have 4 subsets.

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

$p\notin$

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

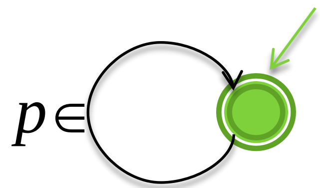


$p\in$

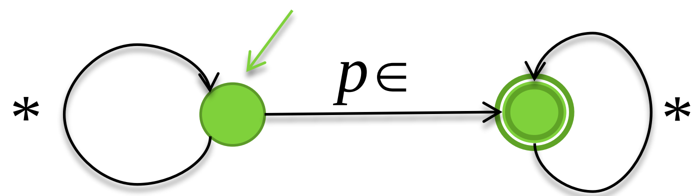
Stands for all subsets of Prop that contain p; thus implying "p holds".

Always and Eventually

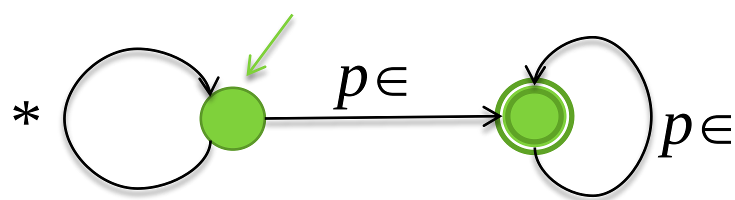
[]p



<>p

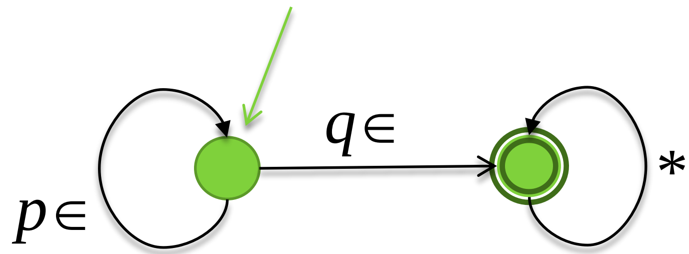


<>[]p

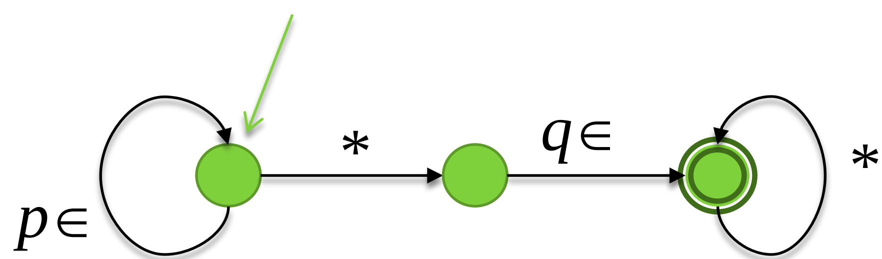


Until

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



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



Not Until

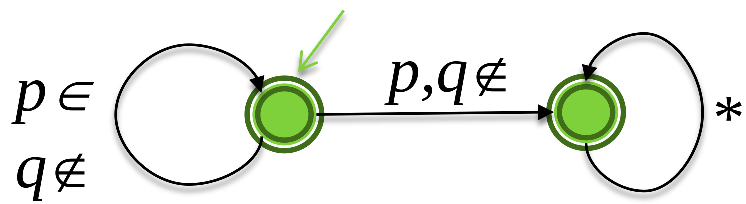
Formula: $\neg\,(\,p\ \mathbf{\color{red}{U}}\ q\,)$

We'll use the help of these properties: $= \neg q\ \mathbf{W}\ \neg p \wedge \neg q$

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

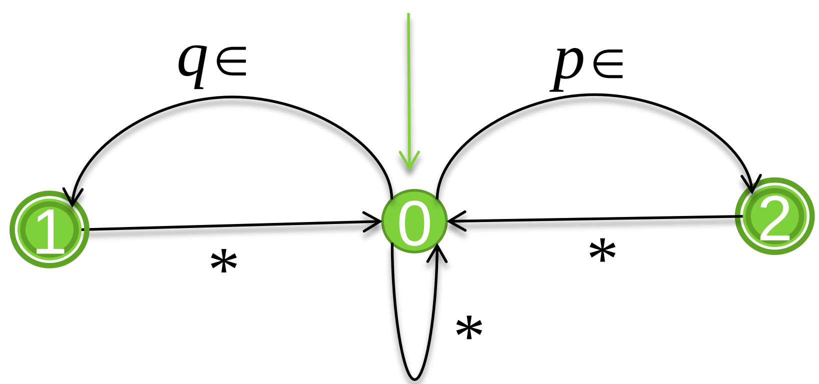
$\neg(p\ \mathbf{W}\ q) \quad = \quad p \wedge \neg q\ \mathbf{U}\ \neg p \wedge \neg q$ $\quad = \neg q\ \mathbf{U}\ \neg p \wedge \neg q$

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



Generalized Buchi Automaton

$[]<>p \quad \wedge \quad []<>q$



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.

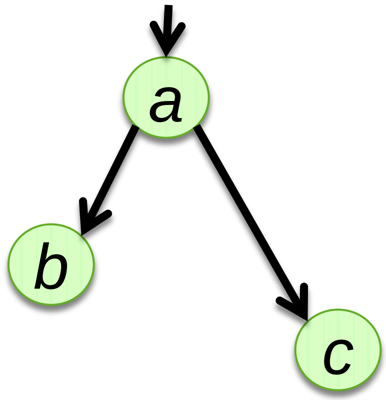
- Still, any LTL formula can be converted to a Buchi. SPIN implements an automated conversion algorithm; unfortunately it is quite complicated.

Check list

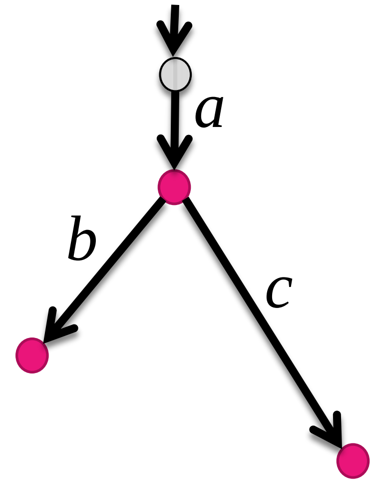
$$P \models\!= \varphi \quad \text{iff} \quad L(P) \cap L(A_{\neg\varphi}) = \varnothing$$

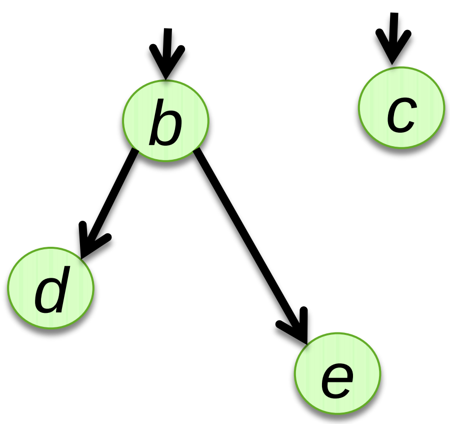
1. How to construct $A_{\neg\varphi}$? → Buchi ✓
2. We still have a mismatch, because *P* is a Kripke structure!
   - Fortunately, we can easily convert it to a Buchi.
3. We still have to construct the intersection.
4. We still to figure out a way to check emptiness.

Label on state or label on arrow...

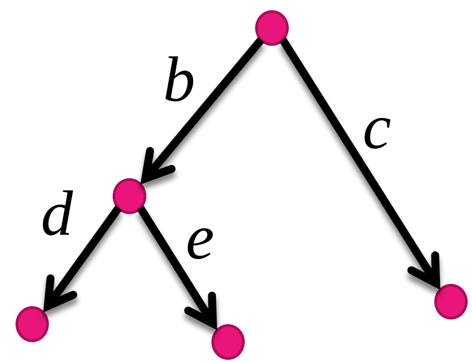


generate the same sentences

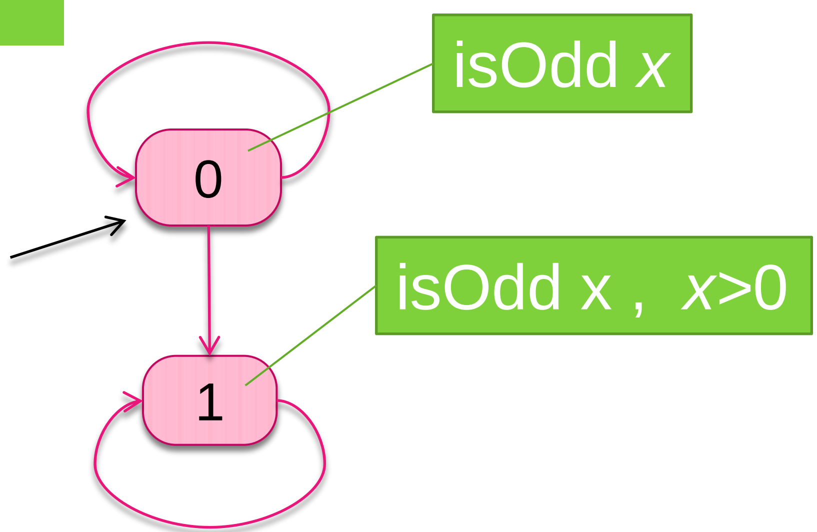


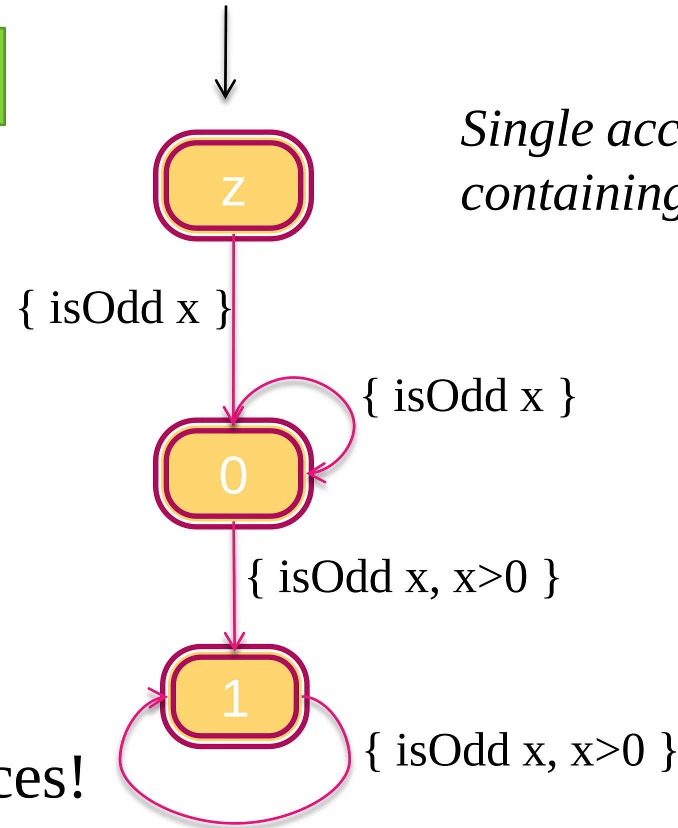


generate the same sentences



Converting Kripke to Buchi





Single accepting set F, containing all states.

Generate the same inf. sentences!

Computing intersection

- Rather than directly checking:

The Buchi version of Kripke P ☺

$$L(A_P) \cap L(A_{\neg\varphi}) = \emptyset$$

We check:

$$L(A_P \cap A_{\neg\varphi}) = \emptyset$$

So we need to figure out how to construct this intersection of two Buchis. Execution over this intersection is also called a "lock-step" execution.

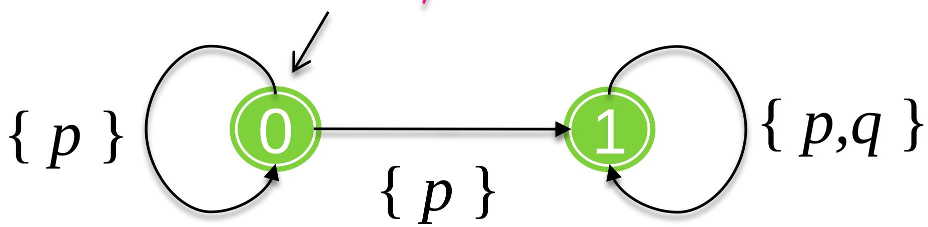
Intersection

- Two buchi automata *A* and *B over the same alphabet*
  - The set of states are respectively Σ_A and Σ_B.
  - starting at respectively s_A and s_B.
  - Single accepting set, respectively F_A and F_B.
  - F_A is assumed to be Σ_A
- A $\cap$ B can be thought as defining lock-step execution of both:
  - The states : $\Sigma_A \times \Sigma_B$, starting at (s_A,s_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 $t \in F_B$.

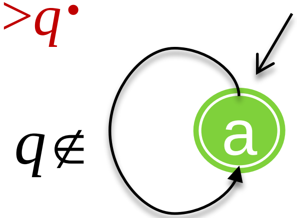
Constructing Intersection, example

p : isOdd x
q : x>0

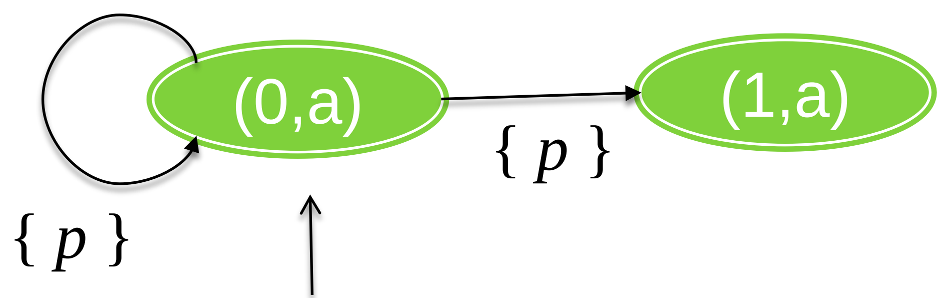
A_p:



$A_{\neg<>q}$:



$A_p \cap A_{\neg<>q}$:



Verification

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

L(*C*) ≠ ∅ 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 $C = A_P \cap A_{\neg\varphi}$ 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.

    (so you don't immediately need the full graph)

DFS-approach (SPIN)

- DFS is a way to traverse a graph :

DFS(u) {

if ($u \in visited$) **return** ;

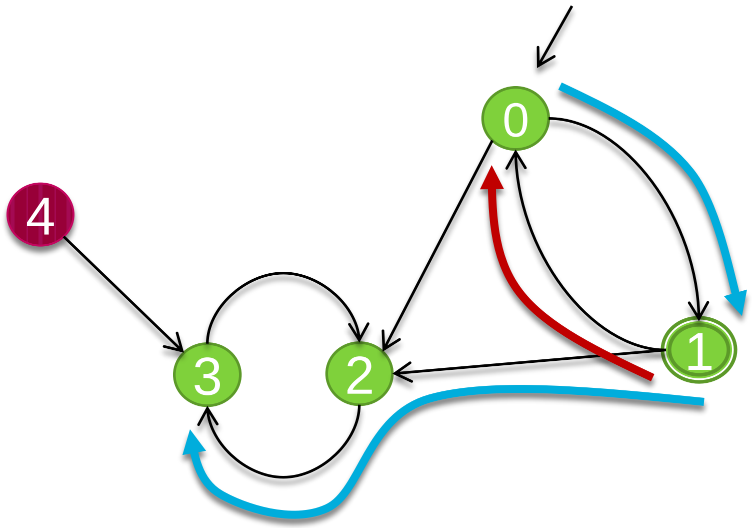
$visited.add(u)$;

for ($s \in next(u)$) DFS(s) ;

}

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

Example



Adding cycle detection

```
DFS(u)  {
    if (u ∈ visited) return ;
    visited.add(u) ;
    for each (s ∈ next(u))  {
        if (u ∈ accept)  {
            visited2 = ∅ ;
            checkCycle (u,s) ;
        }
        DFS(s ) ;
    }
}
```

checkCycle is another DFS

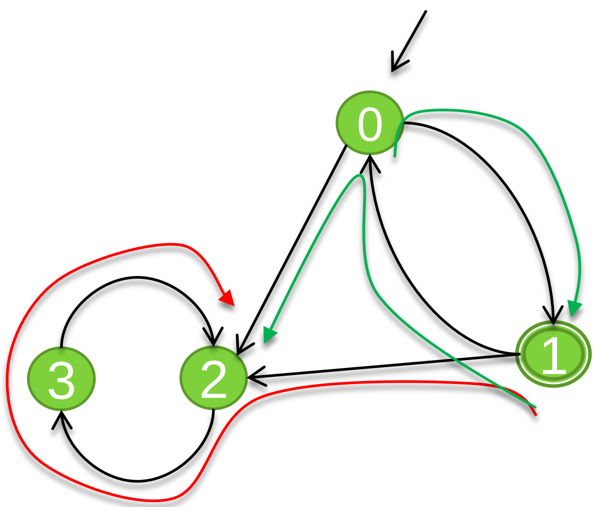
```
checkCycle(root,u)  {

    if  (u = root)  throw CycleFound ;

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

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

Lazy (on-the-fly) construction

- Remember that automaton to explore is $C = A_P \cap A_{\neg\varphi}$
- A_P and $A_{\neg\varphi}$ are not literally expressed as an automata; they are Promela models. In particular A_P , when it is "expanded" to an automaton, it is usually *huge!*

  Can we avoid the construction of A_P?

- Can we avoid the construction of C?

  We can at least do lazy construction (SPIN does so). Benefit : if a cycle is found (verification fails), effort is not wasted to first construct the full *C.*

Lazy construction, representing states

- For now assume *P* is just a single process (no concurrency). If we would construct A_P, each of its state *u* is a pair (*pc*,$\underline{vars}$)
  - *pc* is the "program counter" of *P*, to keep track where *P* is during an execution.
  - $\underline{vars}$ is a vector of the values of all variables at that state.

- *pc* is associated to location in the Promela code; it is straight forward to locate and check all "next" statements/actions α which are possible to execute.

$$\alpha \in \textbf{enabledOn}(pc, \underline{vars})$$

Lazy construction

- If α is an action that is syntactically possible on program counter *pc*, let :

  $$\textbf{exec}\ \alpha\ (pc,\underline{vars})$$

  denote the execution of α at the state $(pc,\underline{vars})$, and this result a new state $(pc',\underline{vars'})$.

- We now modify this quantification in the DFS algorithm:

  for each ($s \in next(u)$) ….

Lazy construction

- To (still incorrect):

  for each ($\alpha \in$ **enabledOn**(*pc*,*<u>vars</u>*)) {
  s = **exec** α (pc,vars)
  …
  }

- $\alpha \in$ *possible*(*pc*,*<u>vars</u>*) means that α is syntactically a possible next action at *pc,* and can execute on state *<u>vars</u>*.

- But, we also have to deal with the intersection.

Lazy construction + intersection

```
DFSlazy(path, <u,v> )  {
   if (<u,v> ∈ visited) return ;
   visited.add(<u,v>) ;

   for each (α ∈ enabledOn(u),  β ∈ outArrow(v),
                 suchthat label(β) holds on u )
     s =  exec(α,u)
     t  =  destination (β)
     if (t ∈ accept)  {
         visited2 = ∅ ;
         checkCycle ( ....) ;
         }
     else  DFSlazy( ... , < s, t > ) ;
}
```

```
DFS1(path,u)  {
   if (u ∈ visited) return ;
   visited.add(u) ;
   for each (s ∈ next(u))
     if (s ∈ accept)  {
         visited2 = ∅ ;
         checkCycle ( path++[u] , u, s) ;
         }
     else  DFS1( path++[u], s ) ;
}
```

Concurrency

- So far, we assumed P is just a single process. What if we have a system S of N concurrent processes $A_1 \ldots A_N$.
- Recall : interleaving model of execution.
- Solution: construct an automaton that equvallently models S, by taking the product automaton:

$$A_1 \times \ldots \times A_N$$

- But... again you may prefer to construct this product lazily as well.

Interleaved execution in SPIN

- The state of $A_1 \| \ldots \| A_N$ is represented by a vector

$$u \;=\; \langle pc_1,\ldots,pc_N, \underline{vars}\rangle$$

- *<u>vars</u>* represent the vector of all variables (of all processes).
- We just need to redifine $\alpha \in$ ***enabledOn****(u):*
  - α belongs to some process A_i , and syntacticaly can execute at pc_i.
  - α is enabled on *<u>vars</u>*

Fairness

Consider this concurrent system :

P { do :: $x = (x+1) \% N$ od } || Q { (x==0) ; print x }

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

- The runtime system determines which fairness assumption is reasonable :
  - No fairness
  - 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.
  - There are other variations…
- A fair execution : an execution respecting the assumed fairness condition.

Fairness in SPIN

```
active proctype P(){
   do
    :: x = (x+1) % N
    od
}
active proctype Q() {
    (x==0) ;
    lab1 : print x   }
```

SPIN only impose "process level weak fairness" : when a process is continually enabled (it has at least one runnable action), an action of the process will eventually be executed.

More elaborate fairness assumptions can be encoded as LTL formulas (but gives additional verification overhead).

- *WFair* = [] ([](*x*==0) → <>*Q*@lab1)
- *SFair* = [] ([]<>(*x*==0) → <>*Q*@lab1)
- To verify, e.g. *SFair* → <> *x is printed*

Closing remarks

- In principle the use of LTL model checking technique is not limited to SPIN.

- Model checking real programs (as in Java Pathfinder)
  - You need a way to fully control thread scheduling
  - You have to constraint values range to make them finite state.
  - You may also need to limit the depth/length of executions

- Testing concurrent programs
  - Chose a selected set of inputs P(x). These are your test-cases. For each test-case, use model checking to verify all possible scheduling of the threads.