Model Checking

Testing & Verification

Dept. of Computer Science & Engg, IIT Kharagpur

Pallab Dasgupta

Professor, Dept. of Computer Science & Engg.,
Professor-in-charge, AVLSI Design Lab,
Indian Institute of Technology Kharagpur

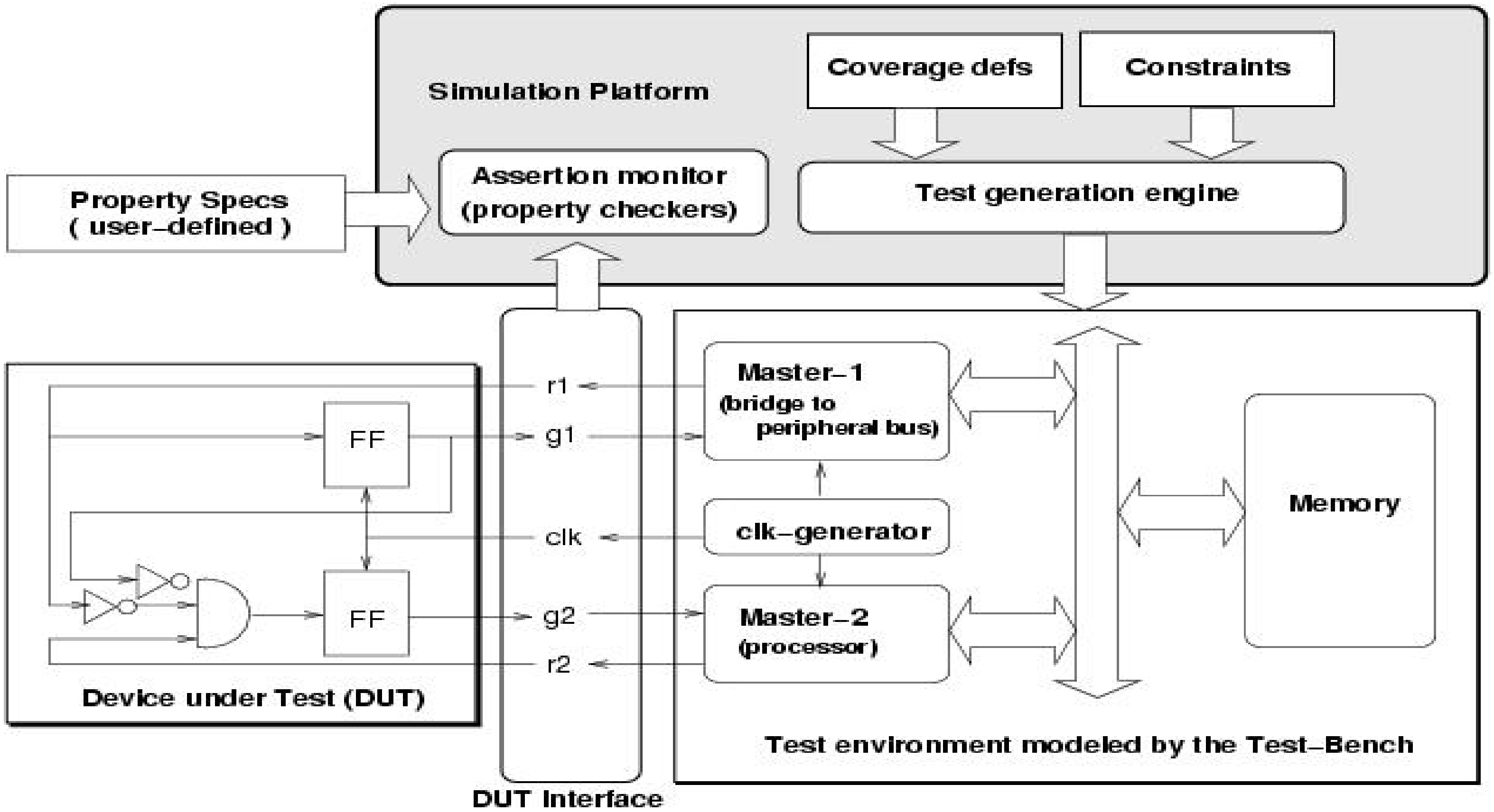
Agenda

- Quick Overview
- Properties, Automata and State Machines
- Model Checking

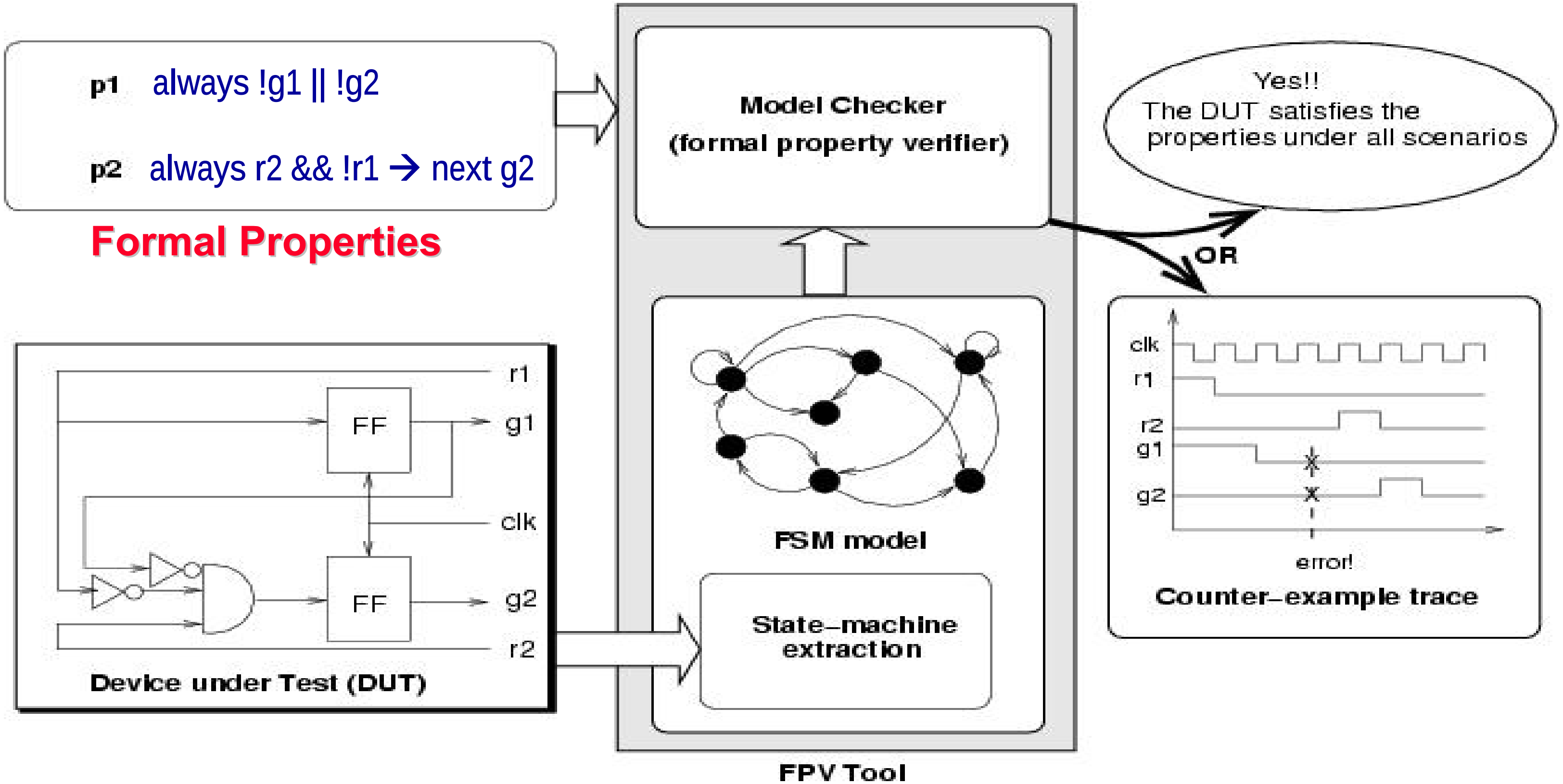
Formal Property Verification

- What is *formal property verification?*
  - Verification of *formal properties?*
  - *Formal methods* for property verification?
- Both are important requirements
- Broad Classification
  - Dynamic property verification (DPV)
  - Static/Formal property verification (FPV)

Dynamic Property Verification (DPV)



Formal Property Verification (FPV)



Temporal Logics (Timed / Untimed, Linear Time / Branching Time): ***LTL, CTL***

Early Languages: ***Forspec (Intel), Sugar (IBM), Open Vera Assertions (Synopsys)***

Current IEEE Standards: ***SystemVerilog Assertions (SVA),***
Property Specification Language (PSL)

Formal Property Verification

- The formal method is called *"Model Checking"*
  - The algorithm has two inputs
    - A finite state transition system (FSM) representing the implementation
    - A formal property representing the specification
  - The algorithm checks whether the FSM "*models*" the property
    - This is an exhaustive search of the FSM to see whether it has any path / state that refutes the property.

Advent of FPV

Goal: *Exhaustive verification of the design intent within feasible time limits*

Philosophy: *Extraction of formal models of the design intent and the implementation and comparing them using mathematical / logical methods*

Formal Properties — **Design Intent**

```
always @( posedge clk )
begin
 if (!rst) begin a1 <= a2;
     a2 <=  ~a1; end;
end
```

Implementation

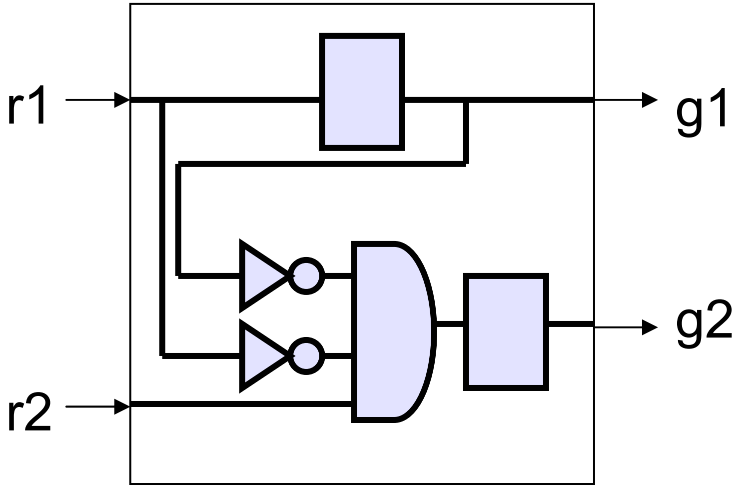
Model Checking

- Temporal Logics
  (*Turing Award: Amir Pnueli*)
- Adopted by Accelera / IEEE
- Integrated into SystemVerilog
- Tools:
  Academia: NuSMV, VIS
  Industry: Magellan (Synopsys)
  IFV (Cadence)
- 2008: Clarke & Emerson get Turing Award

LTL Model Checking: *Philosophy*

- Given a LTL property, φ, to be checked over a module, *M,* we do the following:
  - Create a checker automaton, $B_{\neg\varphi}$, which accepts runs satisfying $\neg\varphi$
  - Extract a (possibly non-deterministic) finite state machine, *J*, from the module, *M.*
  - Compute the product of J with $B_{\neg\varphi}$ and check whether the product has any accepting run.
    - If not then $M \models \varphi$.
    - Otherwise, the accepting run is a counter-example trace.

Example: *Priority Arbiter*



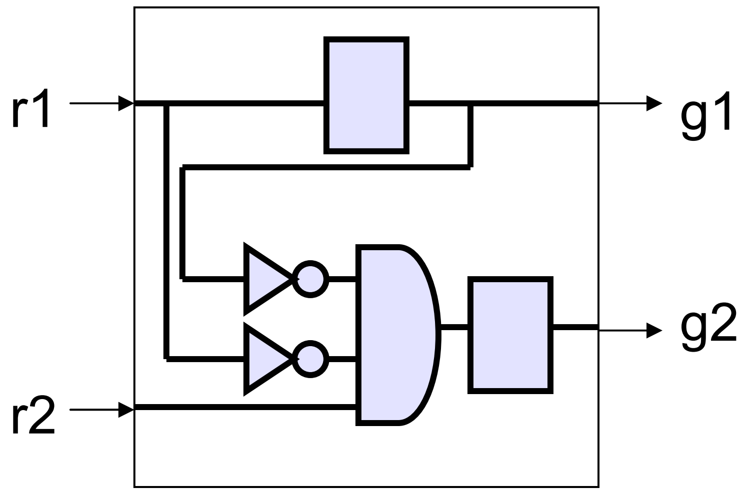
Implementation

Specification

Property:

- ***Either of the grant lines is always asserted***
- ***In Linear Temporal Logic:*** **$G(g1 \vee g2)$**

Step1: FSM Extraction



<u>***Transition Relation:***</u>

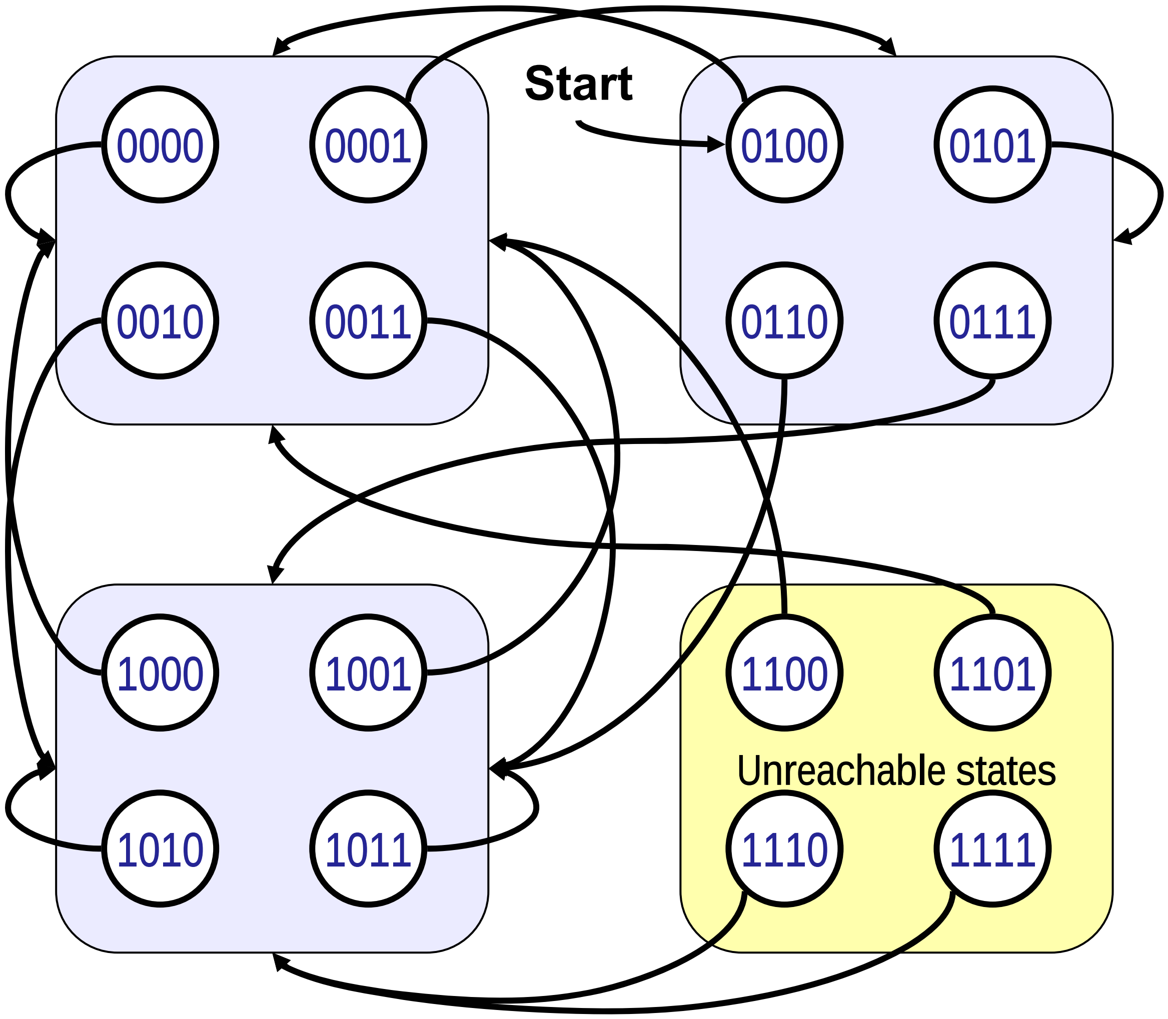
$g'_1 \Leftrightarrow r_1$

$g'_2 \Leftrightarrow \neg r_1 \wedge r_2 \wedge \neg g_1$

Start state: $r_1=0$, $r_2=0$, $g_1=0$, $g_2=1$

PS g_1g_2	I/P r_1r_2	NS $g'_1g'_2$
00	00	00
00	01	01
00	10	10
00	11	10
01	00	00
01	01	01
01	10	10
01	11	10
10	00	00
10	01	00
10	10	10
10	11	10
11	00	00
11	01	00
11	10	10
11	11	10

Step1: *Transition Relation*



PS g_1g_2	I/P r_1r_2	NS $g'_1g'_2$
00	00	00
00	01	01
00	10	10
00	11	10
01	00	00
01	01	01
01	10	10
01	11	10
10	00	00
10	01	00
10	10	10
10	11	10
11	00	00
11	01	00
11	10	10
11	11	10

Step2: Create automaton for property

- **Every LTL property can be converted to a Büchi Alternating Automata**

  - **States of the automata represents the states of the property checker**

  - **The automaton accepts all the valid runs (that is, those that satisfy the property)**

  - *How does this help to check the property?*

Step2: Applying the strategy

- Our property: $\varphi = G[\, g_1 \vee g_2 \,]$
  - Either of the grant lines is always active
- We will create the automaton, $\mathcal{A}$, for $\neg\varphi$
  - $\neg\varphi = F[\, \neg g_1 \wedge \neg g_2 \,]$
  - Sometime both grant lines will be inactive
- We will then search for a common run between this automaton and the FSM for the implementation

How to create the checker automaton?

- Let us consider a property Fq   *// Eventually q is true*

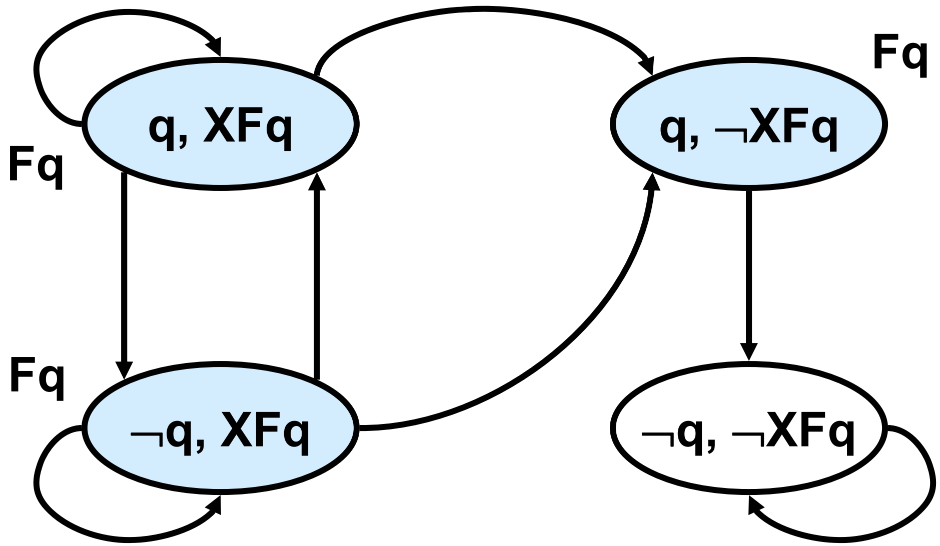
- We can rewrite it as:

  $Fq = q \vee XFq$ *// Either q is true now or Fq is true in the next state*

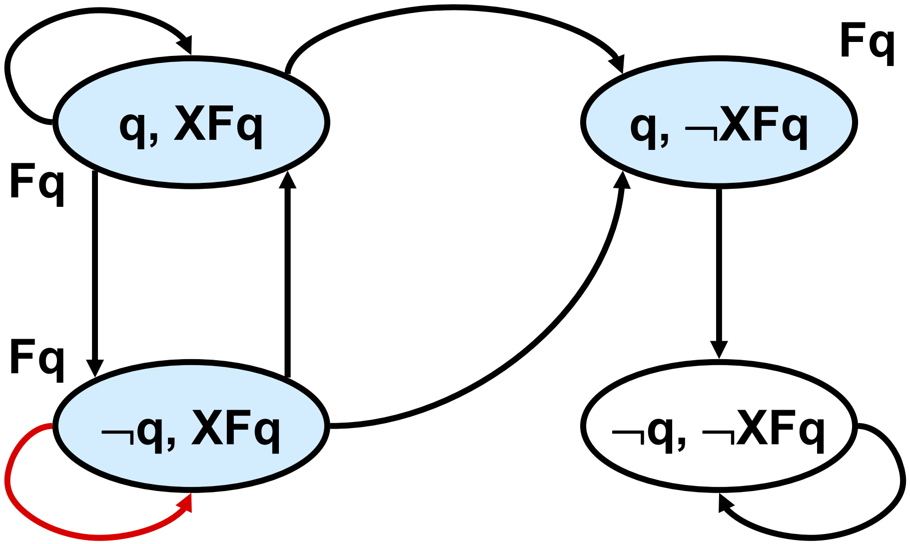
- Therefore we can have the following types of states:
  - States that satisfy q
  - States that do not satisfy q but satisfy XFq
  - States that do not satisfy q and do not satisfy XFq
  - *The first two types are labeled by Fq*

The automaton for our property

Our property: **Fq** *where* **q = ¬g$_1$ ∧ ¬g$_2$**



What is this automaton?



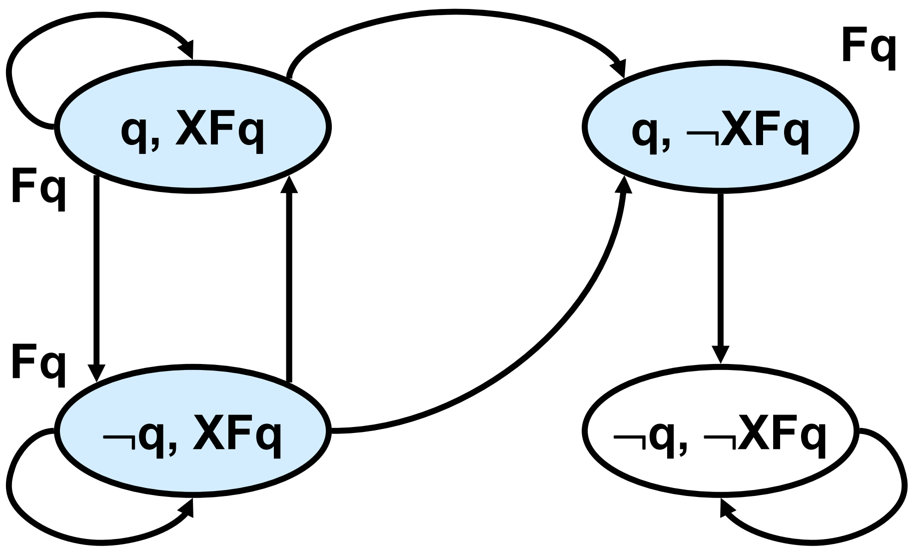
Every run satisfying Fq belongs to this automaton

- which are these runs?
- runs starting from Fq labeled states

But all runs starting from Fq–labeled states do not satisfy Fq

- **Eg. runs that stay in state s forever do not satisfy Fq**

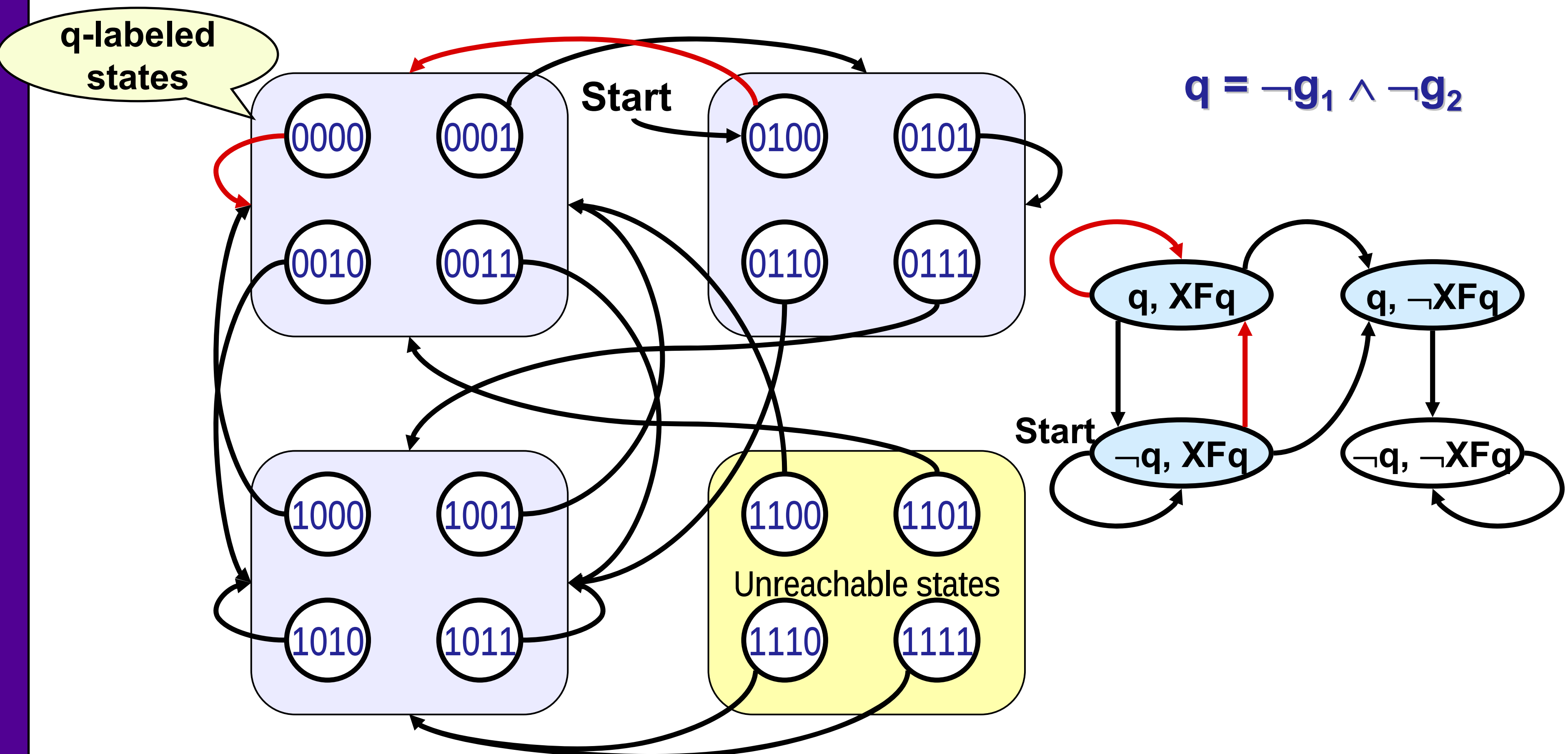
Which runs satisfy Fq?



Runs that start from Fq–labeled states and visit states labeled by q or by ¬Fq infinitely often.

- ***This can be expressed as a fairness constraint***

Step3: *Is the product non-empty?*



The common run is shown in red. Product is non-empty.

Conclusion: Our implementation does not model G[$g_1 \vee g_2$]

Computational facts

- If a LTL property has k sub-formulas, then the checker automaton for it has $O(2^k)$ states
  - Decomposing the property into a conjunction of smaller properties helps in containing the size of this automaton
  - It also helps the FPV tool to prune away parts of the implementation before taking the emptiness check
- LTL model checking is PSPACE-complete, but linear in the size of the implementation
  - The main bottleneck is in the size of the implementation

Capacity is the main issue

- The size of the global state transition system is exponential in the total number of bits in the RTL
  - This is the major bottleneck, even in control dominated designs
  - Efficient compact representations of the state space is the key challenge

- Also the checker automaton grows exponentially with the length of the property
  - With increasingly complicated properties, this is also becoming a growing issue

Background Theory

- Creating the checker automaton
  - LTL properties can be converted to non-deterministic Buchi automata
  - The determinization problem of Buchi automata
- Model checking
  - Finding strongly connected components
  - Tableau construction
- Fixpoint algorithms and CTL model checking
- LTL model checking → CTL model checking

Definitions

- The symbol ω is used to denote the set of non-negative integers, that is, $\omega = \{0, 1, 2, 3, \dots\}$
- By Σ we denote a finite alphabet
  - Σ^* is the set of finite words over Σ
  - Σ^ω denotes the set of infinite words (or ω-words) over Σ
  - We write $\alpha \in \Sigma^\omega$, as $\alpha = \alpha(0)\alpha(1) \dots$ with $\alpha(i) \in \Sigma$.
  - Finite set of letters occurring infinitely often:

    $$\mathrm{Inf}(\alpha) = \{\, a \in \Sigma \mid \forall i\ \exists j > i\ \alpha(j) = a \,\}$$

ω-Automata

- An ω-automaton is a quintuple $(Q, \Sigma, \delta, q_I, Acc)$, where Q is a finite set of states, Σ is a finite alphabet, $\delta: Q \times \Sigma \rightarrow 2^Q$ is the state transition relation, $q_I \in Q$ is the initial state, and Acc is the acceptance component.
  - In a non-deterministic ω-automaton, a transition function $\delta: Q \times \Sigma \rightarrow Q$ is used
  - The acceptance component can be given as a set of states, as a set of state-sets, or as a function from the set of states to a finite set of natural numbers

Büchi Acceptance

- An ω-automaton $A = (Q, \Sigma, \delta, q_I, F)$, with acceptance component $F \subseteq Q$ is called a Buchi automaton if it is used with the following acceptance condition (Buchi acceptance):
  - A word $\alpha \in \Sigma^\omega$ is accepted by A iff there exists a run π of A on α satisfying the condition:

    $$\mathrm{Inf}(\pi) \cap F \neq \Phi$$

    that is, at least one of the states in F has to be visited infinitely often during the run.
  - $L(A) = \{\alpha \in \Sigma^\omega \mid A \text{ accepts } \alpha\}$ is the ω-language recognized by A.

Muller Acceptance

- An ω-automaton $A = (Q, \Sigma, \delta, q_I, F)$, with acceptance component $F \subseteq 2^Q$ is called a Muller automaton when used with the following acceptance condition (Muller acceptance):
  - A word $\alpha \in \Sigma^\omega$ is accepted by A iff there exists a run π of A on α satisfying the condition:

    $$\mathrm{Inf}(\pi) \in F$$

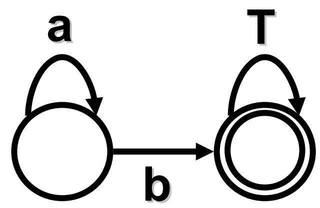
    that is, the set of infinitely recurring states of π is exactly one of the sets in F.

LTL → Buchi Automata

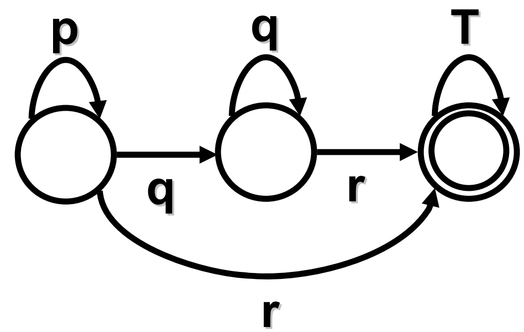
- Theorem [Wolper, Vardi, Sisla '83]: Given an LTL property φ, one can build a Buchi automaton $A = (Q, \Sigma, \delta, q_I, F)$ where
  - $\Sigma = 2^{AP}$
    - the number of atomic propositions, variables, etc in φ
  - $|Q| \leq 2^{O(|\varphi|)}$
    - $|\varphi|$ is the length of the formula

Examples

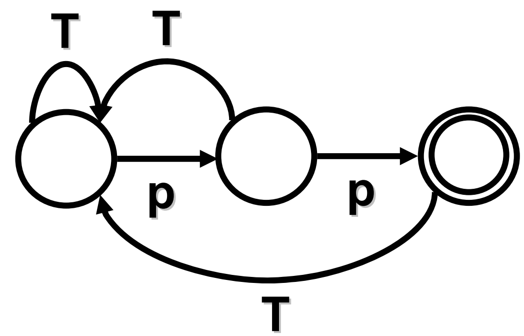
a U b



p U (q U r)



GF(p $\wedge$ Xp)



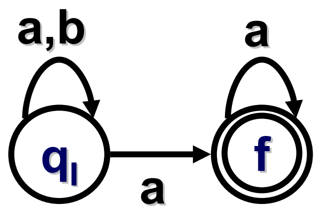
Det. versus Non-det. Buchi Automata

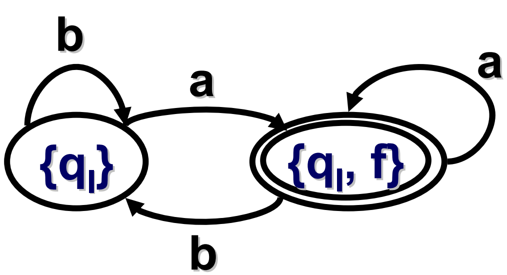
- There exist languages which are accepted by some non-deterministic Buchi automaton but not by any deterministic Buchi-automaton

$A = (\{q_I, f\}, \{a, b\}, \Delta, q_I, \{f\})$

A accepts the language:

$$L = \{ \alpha \in \{a,b\}^\omega \mid \#_b(\alpha) < \infty \}$$





Normal determinization will produce this automaton, which also accepts $(a, b)^\omega \notin L$

The automaton accepts L with $F = \{\{\{q_I, f\}\}\}$ as Muller condition

LTL Model Checking

- Given a model M and an LTL formula φ
  - **Build** the Buchi automaton $B_{\neg\varphi}$
  - **Compute product** of M and $B_{\neg\varphi}$
    - Each state of M is labeled with propositions
    - Each state of $B_{\neg\varphi}$ is labeled with propositions
    - Match states with the same labels
  - The product accepted the traces of M that are also traces of $B_{\neg\varphi}$ $(\Sigma_M \cap \Sigma_{\neg\varphi})$
  - If the product accepts any sequence
    - We have found a counter-example

Symbolic Tableau Construction

Elementary Formulas

- A LTL formula φ is called *elementary,* if it is a variable ($\varphi \in AP$), a negated variable ($\varphi = \neg\psi$, with $\psi \in AP$) or the outermost operator is a *next* operator ($\varphi = X\psi$).

$$el(\varphi) := \{\varphi\}, \text{ if } \varphi \in AP$$

$$el(\neg\varphi) := el(\varphi)$$

$$el(\varphi \vee \psi) := el(\varphi) \cup el(\psi)$$

$$el(X\varphi) := \{X\varphi\} \cup el(\varphi)$$

$$el(\varphi \; U \; \psi) := \{X(\varphi \; U \; \psi)\} \cup el(\varphi) \cup el(\psi)$$

Symbolic Tableau Construction

- The set of states of the tableau is $S_T = 2^{el(\varphi)}$
- The labeling function L_T is defined as follows:

$$\begin{aligned}
Sat(\varphi) &:= \{s \mid \varphi \in s\}, \text{ if } \varphi \in el(\varphi) \\
Sat(\neg\varphi) &:= \{s \mid \varphi \notin Sat(\varphi)\} \\
Sat(\varphi \vee \psi) &:= Sat(\varphi) \cup Sat(\psi) \\
Sat(\varphi \mathbf{U} \psi) &:= Sat(\psi) \cup (Sat(\varphi) \cap Sat(X(\varphi \mathbf{U} \psi)))
\end{aligned}$$

$$R_T(s, s') = \bigwedge_{X\psi \in el(\varphi)} (s \in Sat(X\psi) \Leftrightarrow s' \in Sat(\psi))$$

Language Emptiness

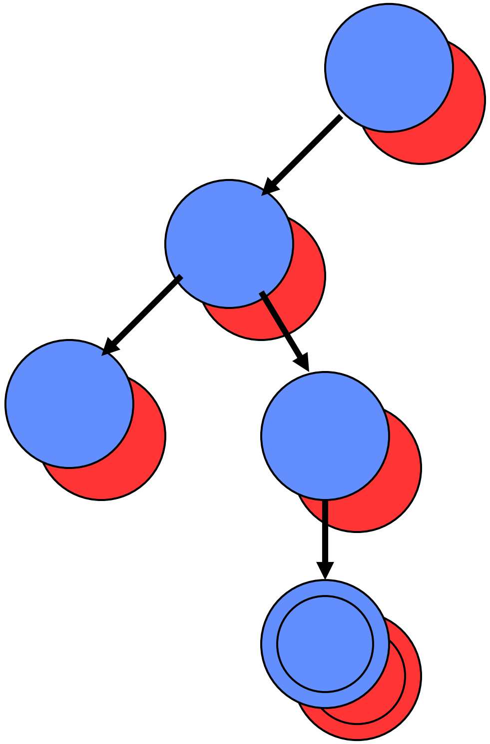
$$\Sigma_M \cap \Sigma_{\neg\varphi} = \varnothing$$

- Compute strongly connected components
  - *Non trivial*
  - Containing an *accepting state*
- None means no sequence is accepted
  - *Proved the property*
- Very expensive

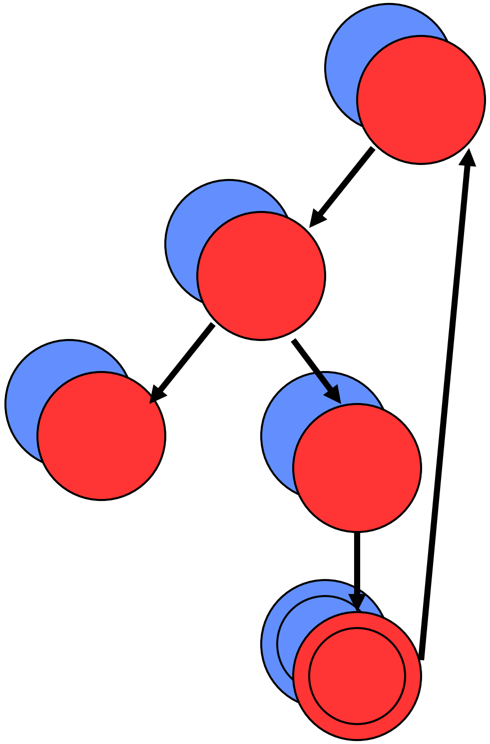
Nested Depth First Search

- The product is a Büchi automaton
- How do we find accepted sequences?
  - Accepted sequences must contain a cycle
    - In order to contain accepting states infinitely often
  - We are interested only in cycles that contain at least an accepting state
  - During depth first search start a second search when we are in an accepting states
    - If we can reach the same state again we have a cycle (and a counter-example)

Example



Example



Nested Depth First Search

```
procedure DFS(s)
  visited = visited ∪ {s}
  for each successor s’ of s
    if s’ ∉ visited then
      DFS(s’)
      if s’ is accepting then
        DFS2(s’, s’)
      end if
    end if
  end for
end procedure
```

Nested Depth First Search

```
procedure DFS2(s, seed)
  visited2 = visited2 ∪ {s}
  for each successor s’ of s
    if s’ = seed then
      return “Cycle Detect”;
    end if
    if s’ ∉ visited2 then
      DFS2(s’, seed)
    end if
  end for
end procedure
```

CTL Model Checking

- Need only Modalities EX, EU, EG.
- Other Modalities can be expressed in terms of EX, EU, EG.

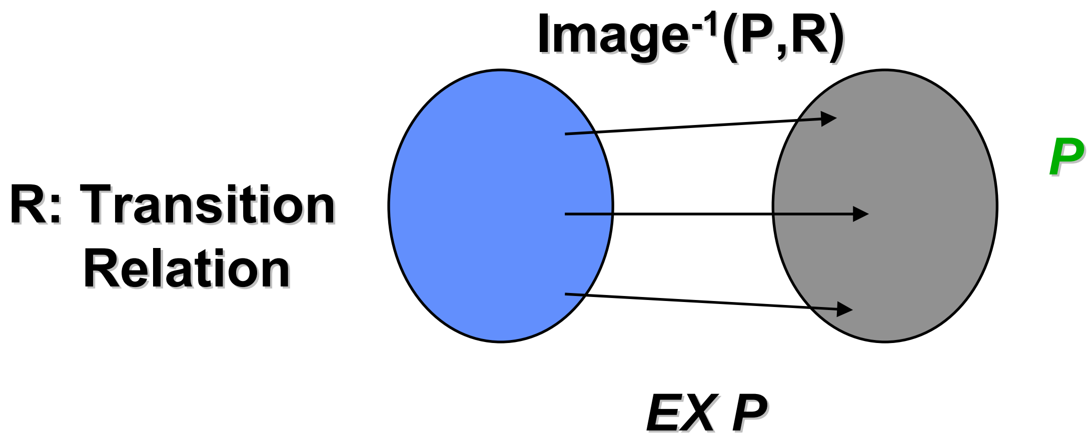
  - $AFp = \neg EG \neg p$
  - $AGp = \neg EF \neg p$
  - $A(p \; U \; q) = \neg E[\neg q \; U \; (\neg p \wedge \neg q)] \wedge \neg EG \neg q$

[Clarke,Emerson]

Example EX p

- **Reverse image**

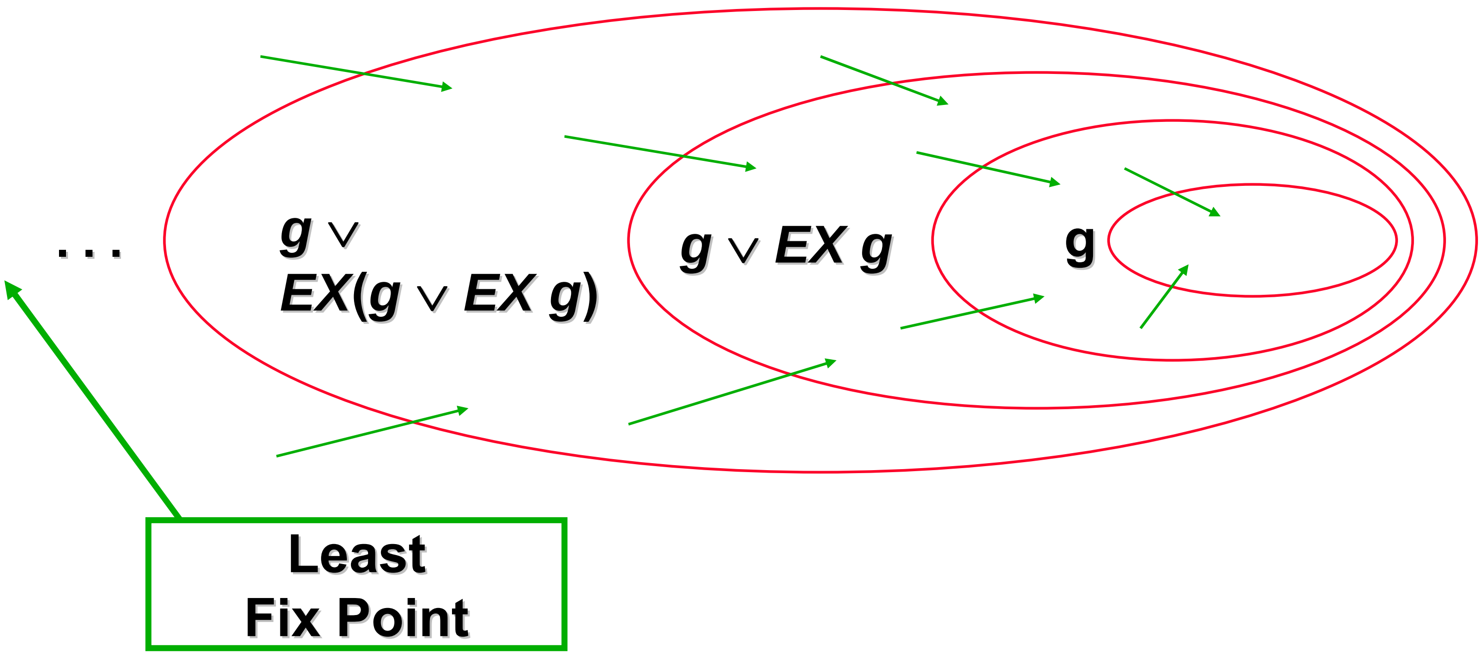
$$\text{Image}^{-1}(P,R) = \{v : \text{ for some } v', v' \in P \text{ and } (v,v') \in R\}$$



EXp = $\exists v\ (\ (v,v') \in R \wedge p \in L(v)\ )$

Example: EF g

- EF g is calculated as



Model checking f = EF g

Given a model M= < AP,S,S0, R, L >

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';

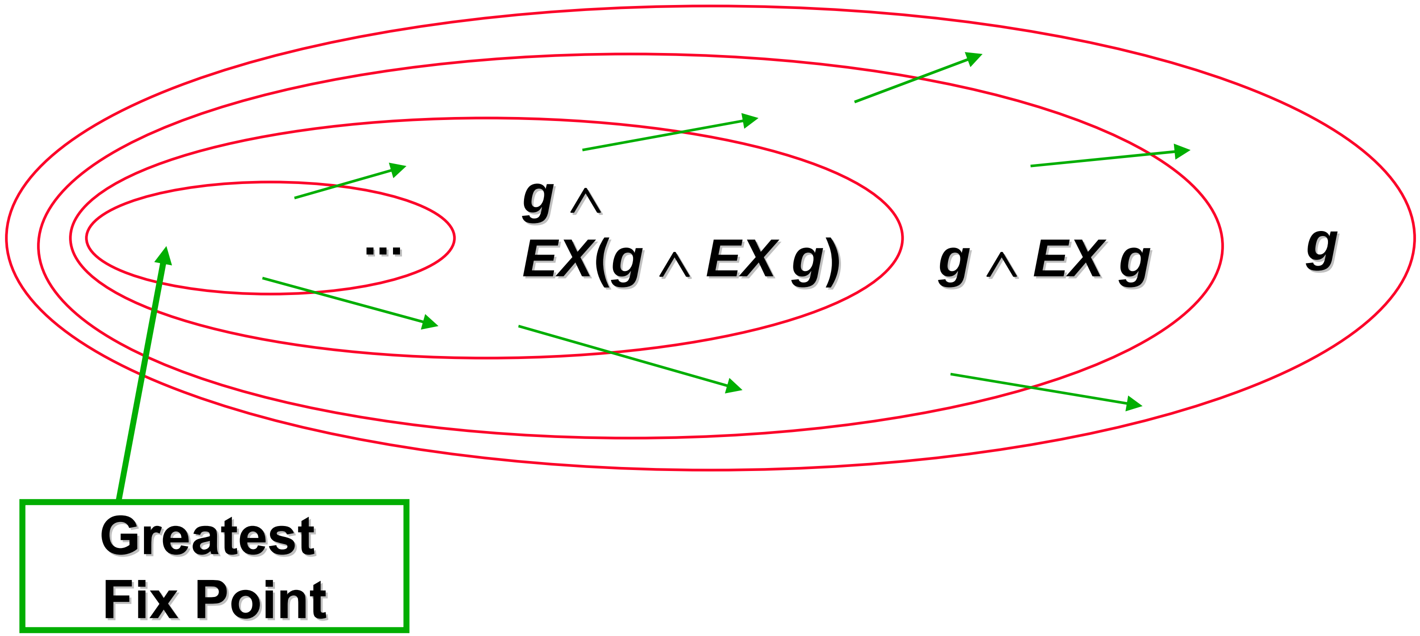
Q' := Q $\cup$ { s | $\exists$s' [R(s,s') $\wedge$ Q(s')] }

end while

S_f := Q ; return(S_f)

Example: EG g

- EG g is calculated as



Model checking f = EG g

Given a model M= < AP,S,S_0, R, L >

and S_g the sets of states satisfying **g** in M

procedure CheckEG (S_g)

Q := S ; Q' := Sg ;

while Q $\neq$ Q' do

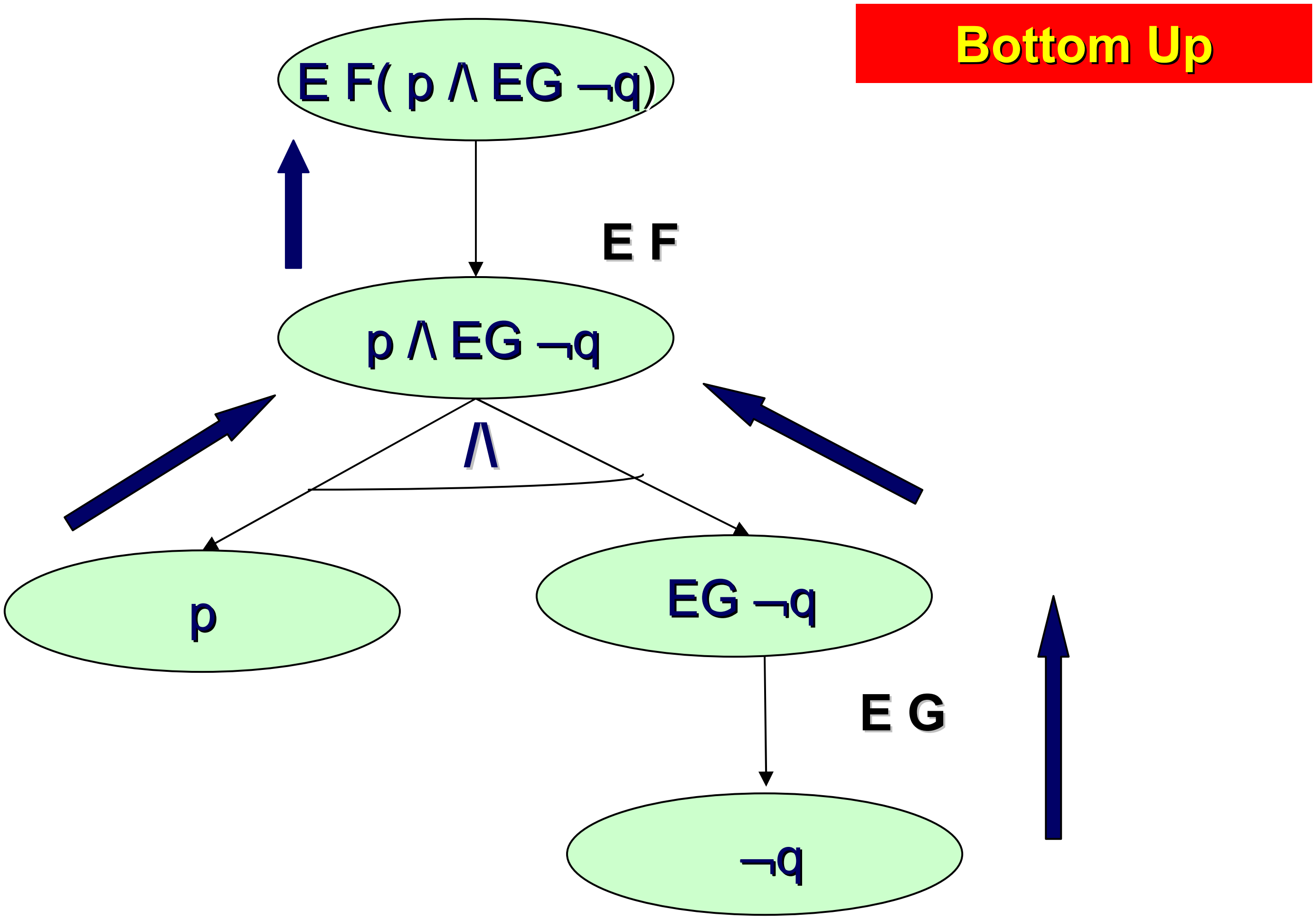
  Q := Q';

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

end while

S_f := Q ; return(S_f)

Checking Nested Formulas



Checking Nested formulas

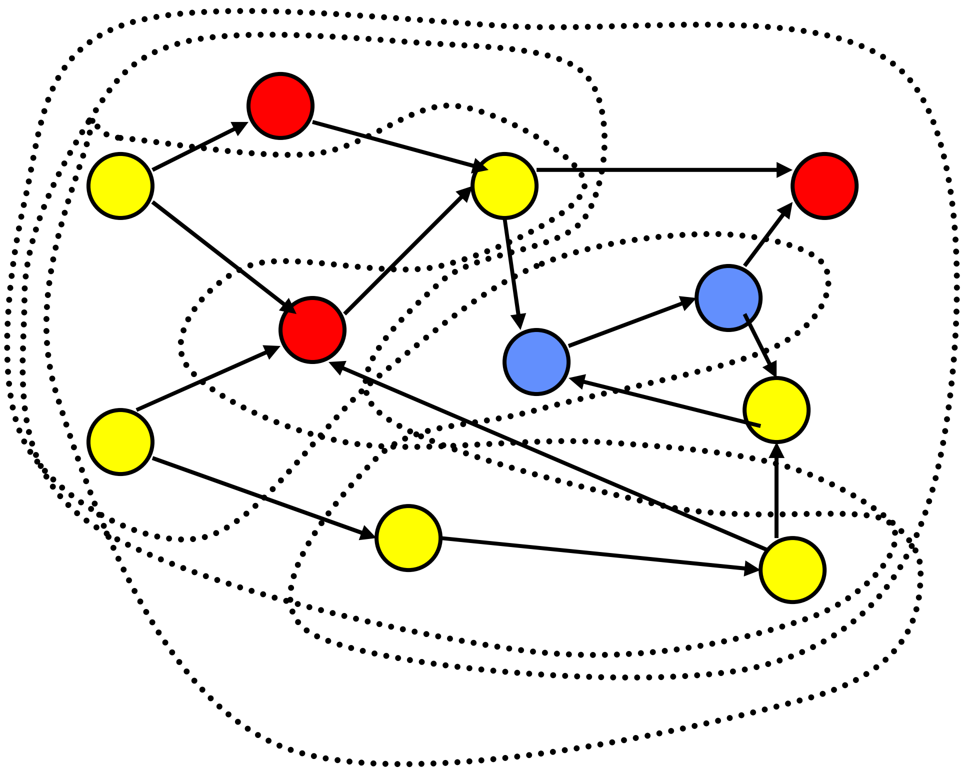
EF (p ∧ EG ¬q)

p state

q state

¬p ∧ ¬q state

EF(p ∧ EG ¬q)

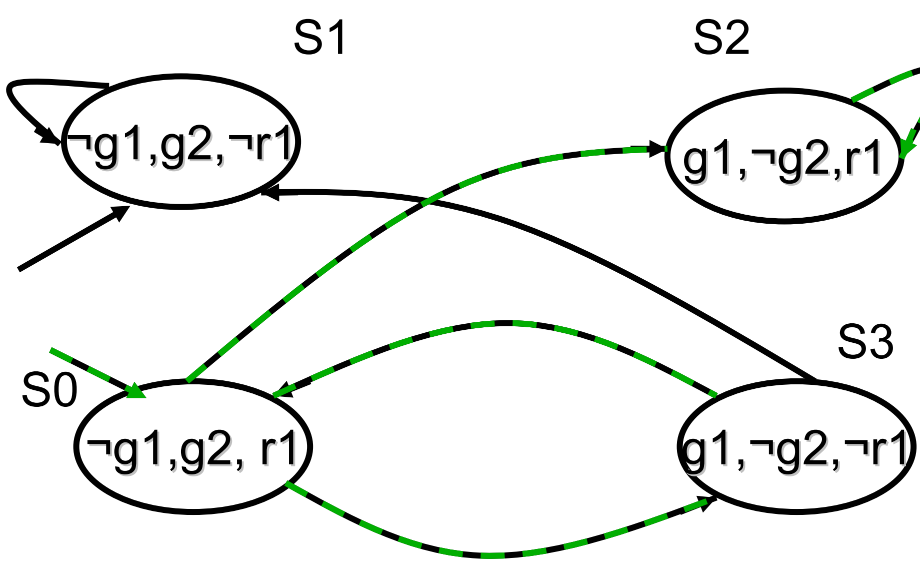


Complexity

- Linear in the size of the Model
- Linear in the size of the CTL Formula
  - Model Size = M
  - Formula Size = |F|
  - Complexity = O (M x |F|)

Fairness in CTL Model Checking

- Fairness F is a set of states {s1,s2,…,sn}
  - A fair path of a model is a path which visits the states in F infinitely often.
  - A CTL formula f is true under the fairness constrain F if f is true only in the FAIR paths of the model.



False Property:

AF(g1)

Fairness: r1 is asserted infinitely often

True Property:

AF(g1) under *fairness* F ={s0,s2}

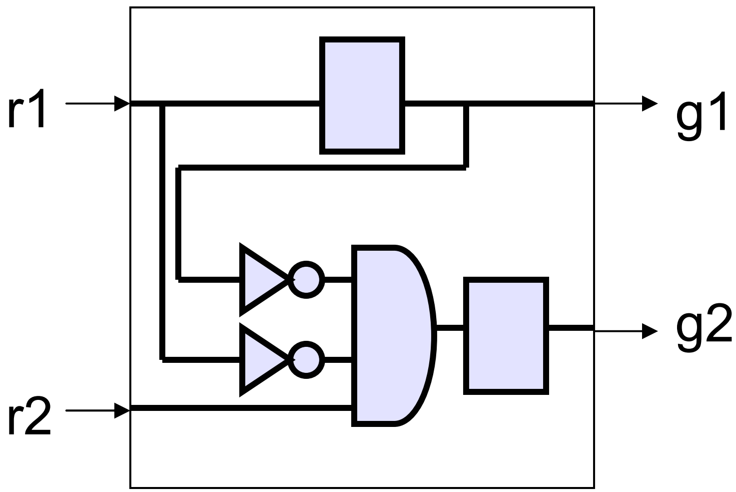
Fairness Formal Semantics

- A fair Kripke structure is a 6 tuple
  - $M=(AP,S,S_0,R,L,F)$ where $F \subseteq 2^S$ is a set of fairness constrains
  - Let $\pi = s_0, s_1, \ldots$ be a path in M
  - $Inf(\pi) = \{s \mid s = s_i$ for infinitely many $i\}$

- We say that π is fair if and only if for every element $P \in F$,

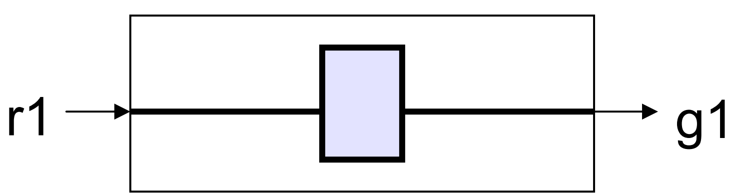
  $inf(\pi) \cap P \neq \Phi$

Cone-of-influence Reductions



The original state machine had 16 states

After COR based on:
$r_1 \Rightarrow Xg_1$



The reduced state machine has 4 states

Bounded Model Checking (BMC)

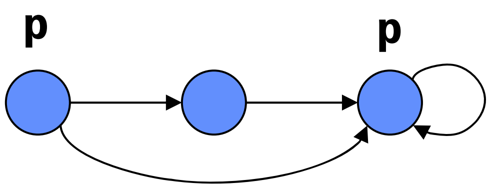
- **Broad Methodology**
  - We construct a Boolean formula that is satisfiable iff the underlying state transition system can realize a finite sequence of state transitions that satisfy the temporal property we are trying to validate
  - We use powerful SAT solvers to determine the satisfiability of the Boolean formula
  - The bound may be increased incrementally until we reach the diameter of the state transition graph

BMC: Translation to SAT

- We unfold the property into Boolean clauses over different time steps
- We unfold the state machine into Boolean clauses over the same number of time steps
- We check whether the clauses are together satisfiable

BMC: Example

- $F\,(p \land q) = (p_0 \land q_0) \lor F\,(p \land q)$
  $= (p_0 \land q_0) \lor (p_1 \land q_1)$
  up to 2 time steps



- From state machine (*up to 2 time steps*)
  $(p_0 \land \neg q_0) \land ((\neg p_1 \land \neg q_1) \lor (p_1 \land \neg q_1))$
  $= (p_0 \land \neg q_0) \land (\neg q_1)$

- The total set of clauses is unsatisfiable

Advantages

- Able to handle larger state spaces as compared to BDD’s.
- Takes advantage of several decades of research on efficient SAT solvers.
- The witness/counterexample produced are usually of minimum possible length, making them easier to understand and analyze.

Requirements

- Specification in temporal logic.
- System as a finite state machine.
- Bound, k, on path length.
  - In bounded model checking, only paths of bounded length k or less are considered.

Limitations of BMC

- Sound but not complete
  - Works for a bounded depth
  - In order to have a complete procedure, we need to run it at least up to the diameter (unknown) of the transition system
- For larger depths the number of clauses can grow rapidly, thereby raising capacity issues
- Nevertheless, SAT-based FPV tools can handle much larger designs as compared to BDD-based tools

On-the-fly FPV tools

- **Automata Theoretic on-the-fly FPV Tools**
  - Creates the checker automaton
  - The emptiness search is done depth-first, thereby saving space
  - Trades model checking time for space efficiency

ATPG-based FPV tools

- ATPG based FPV Tools
  - Synthesizes the checker automaton as a non-deterministic FSM (behavioral)
  - Uses sequential ATPG to generate simulation vectors
  - Not complete unless we have 100% test coverage