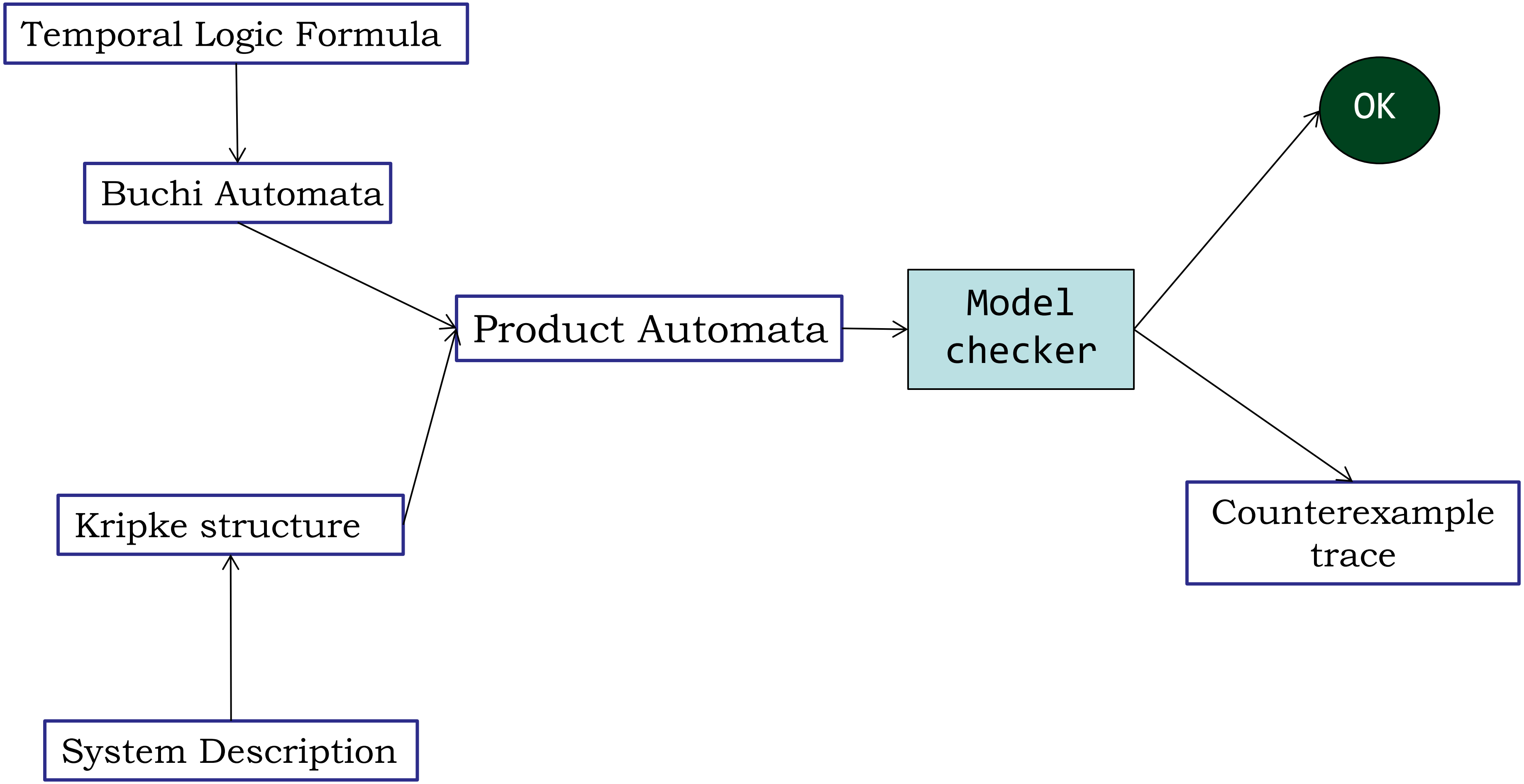
Explicit State Model Checking

Computer Science and Artificial Intelligence Laboratory
MIT
Armando Solar-Lezama

Dec 01, 2011

Explicit State Model checking

The basic Strategy



Buchi Automata

A Buchi Automaton is a 5-tuple $\langle \Sigma, S, I, \delta, F \rangle$

- Σ is an alphabet
- S is a finite set of states
- $I \subseteq S$ is a set of initial states
- $\delta \subseteq S \times \Sigma \times S$ is a transition relation
- $F \subseteq S$ is a set of accepting states

Non-deterministic Buchi Automata are not equivalent to deterministic ones

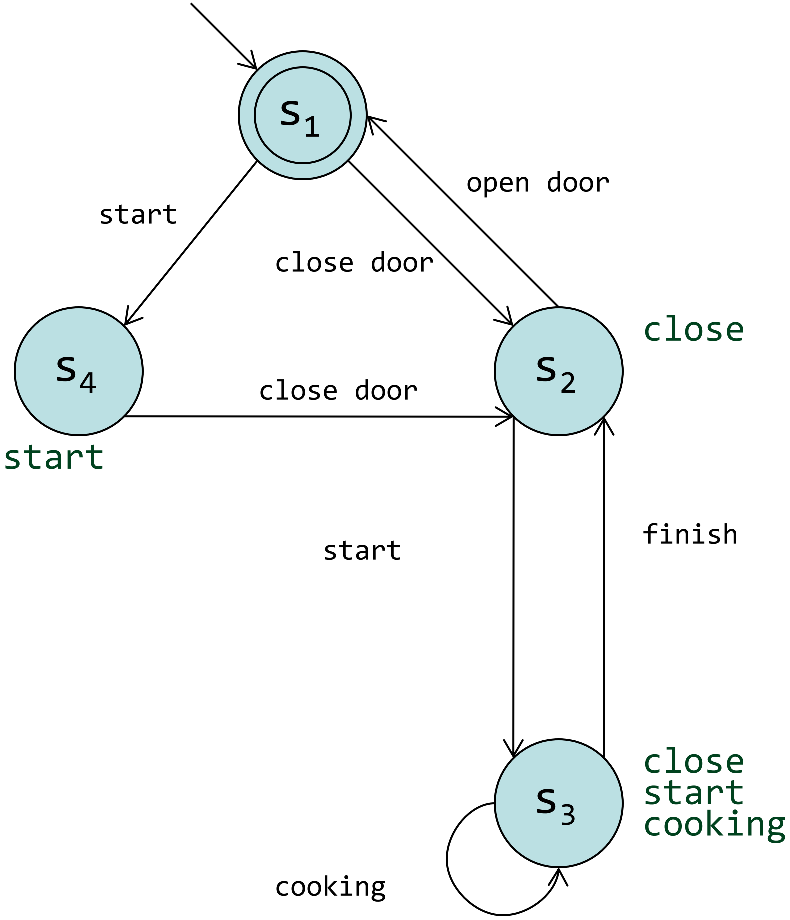
Buchi Automaton from Kripke Structure

Given a Kripke structure:

- M = (S, S0, R, L)

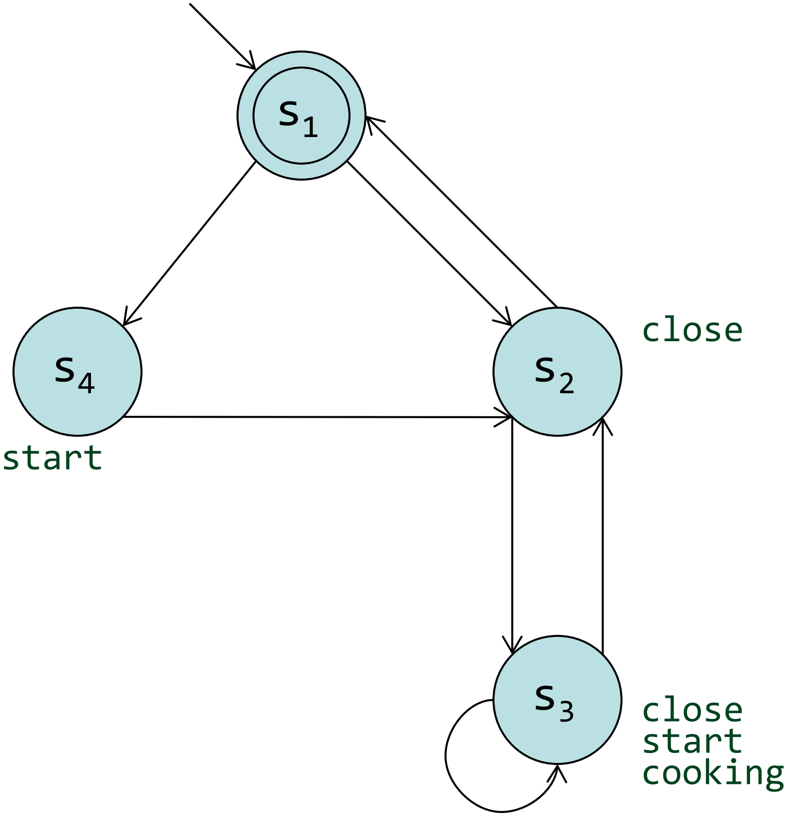
Construct a Buchi Automaton

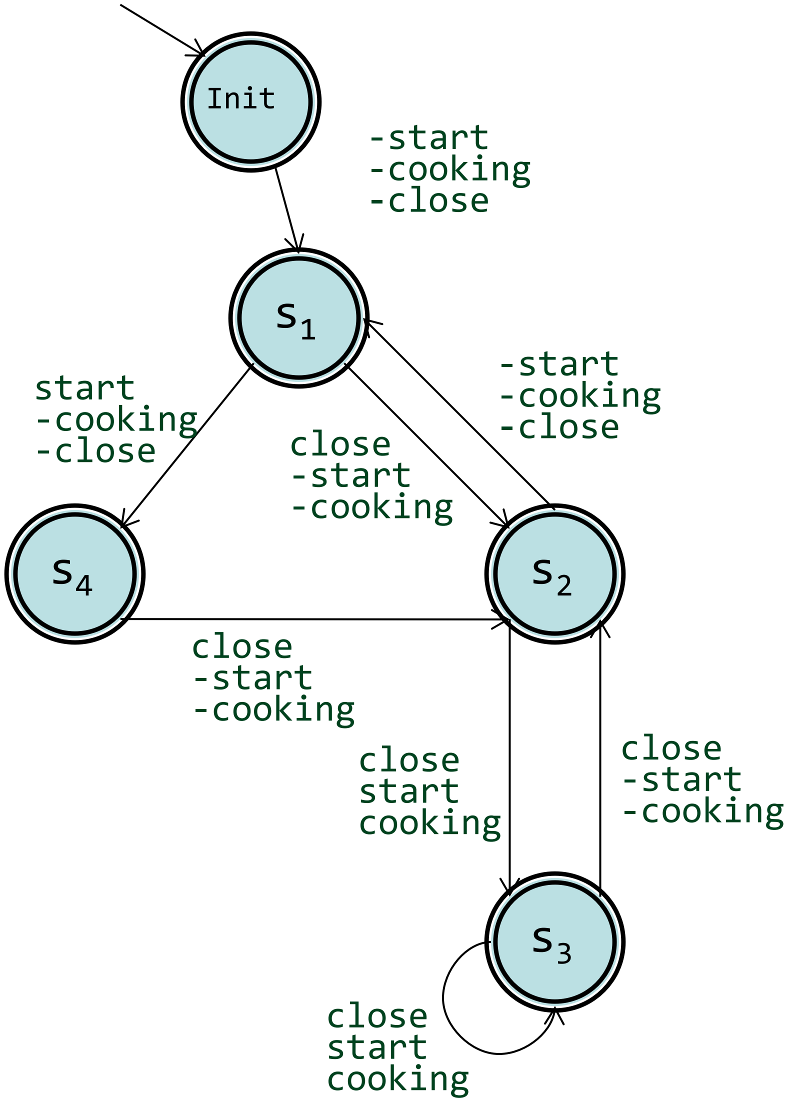
- (***Σ*** , S U {Init}, {Init}, T, S U {Init})
- T is defined s.t.
    - T(s, *σ*, s') iff R(s, s') and *σ*ϵ L(s')
    - T(Init, *σ*,s) iff s ϵ S0 and *σ*ϵ L(s)



Buchi Automaton from Kripke Structure

- (Σ , S U {Init}, {Init}, T, S U {Init})
- T is defined s.t.
  - T(s, σ, s') iff R(s, s') and $\sigma \in$ L(s')
  - T(Init, σ,s) iff s $\in$ S0 and $\sigma \in$ L(s)





Buchi Automaton from Kripke Structure

Given a Kripke structure:

- M = (S, S0, R, L)

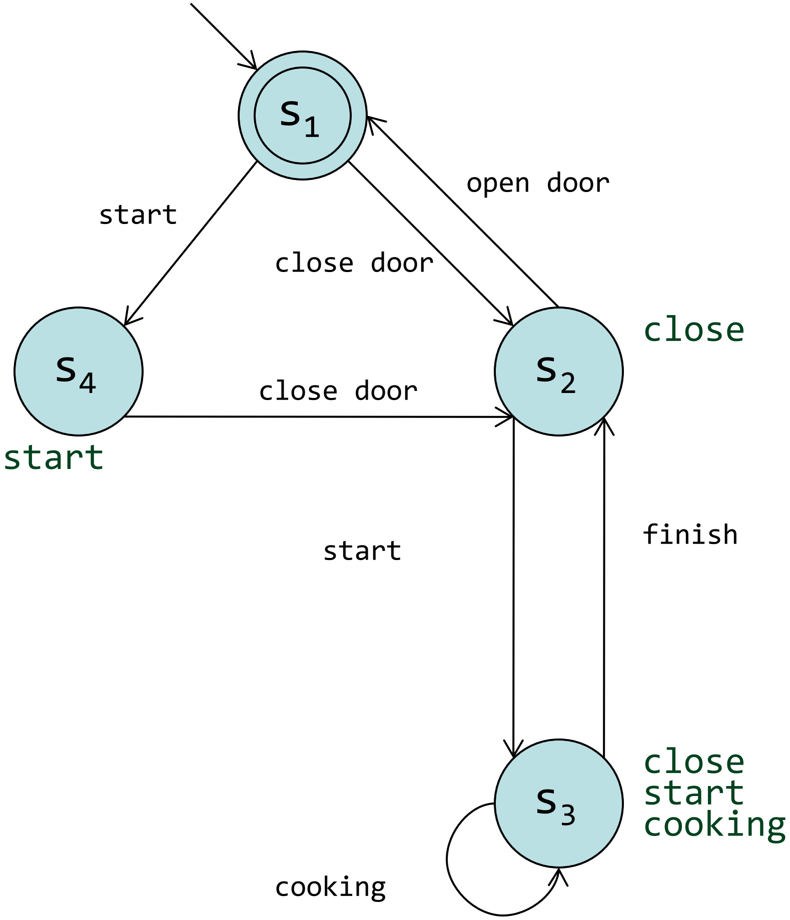
Construct a Buchi Automaton

- (***Σ*** , S U {Init}, {Init}, T, S U {Init})

- T is defined s.t.
  - T(s, σ, s') iff R(s, s') and $\sigma \in$ L(s')
  - T(Init, σ,s) iff s $\in$ S0 and $\sigma \in$ L(s)

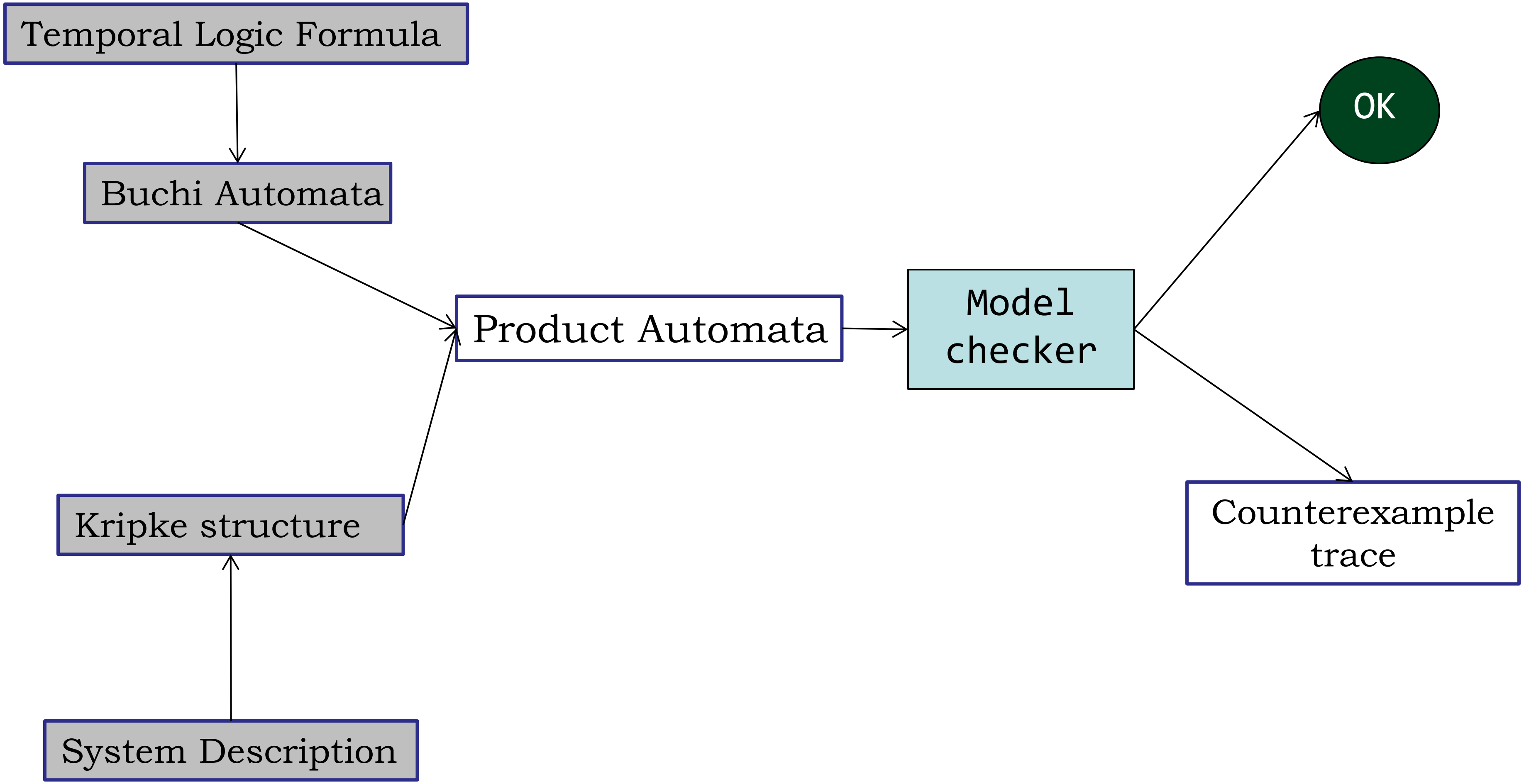
What about missing transitions?

- Need to add a dummy "error state"



Explicit State Model checking

The basic Strategy



Temporal Logic Formula

Buchi Automata

Product Automata

Model checker

OK

Counterexample trace

Kripke structure

System Description

Negated Properties

Given a good property P, you can define a bad property P’

- If the system has a trace that satisfies P’, then it is buggy.

Example

- Good property: G(req ➔ F ack)
- Bad property: F (req & (G !ack))

We are going to ask whether M satisfies P’

- If it does, then we found a bug

Why are we doing the negation?

Computing the Product Automata

Given Buchi automata A and B'

- $A = (\boldsymbol{\Sigma}, S_A, T_A, \{Init_A\}, S_A)$
- $B' = (\boldsymbol{\Sigma}, S_B, T_B, \{Init_B\}, F')$
- $A \times B' = (\boldsymbol{\Sigma}, S_A \times S_B, T, \{(Init_A, Init_B)\}, F)$

Where

- $T((s_1,s_2), \sigma, (s_1', s_2'))$ iff $T_A(s_1, \sigma, s_1')$ and $T_B(s_2, \sigma, s_2')$
- $(s_1,s_2) \in F$ iff $s_2 \in F'$

Check if a state is visited infinitely often

Check for a cycle with an accepting state

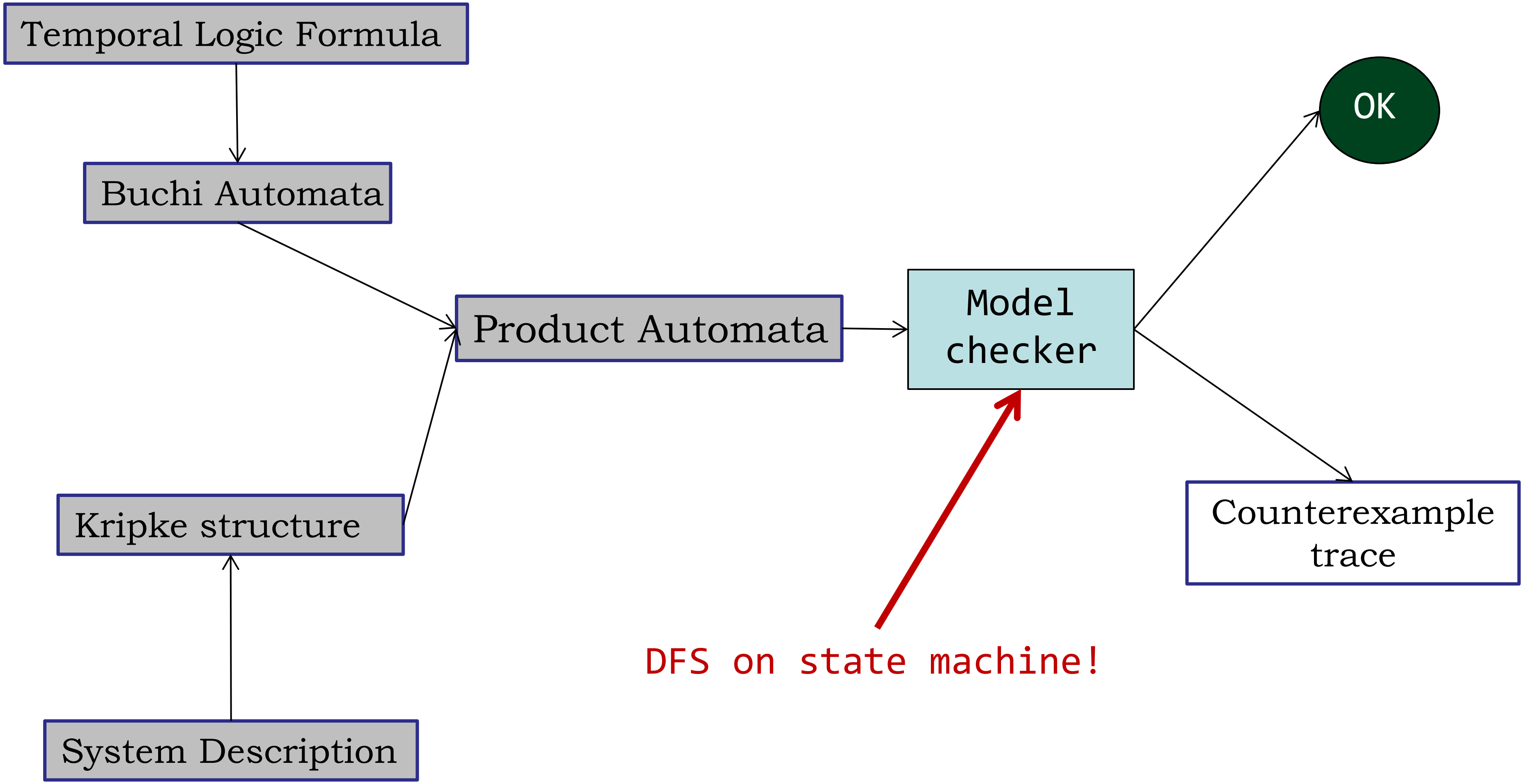
Cycle must be reachable from the initial state

Simple algorithm

- Do DFS to find an accepting state
- Do a DFS from that accepting state to see if it can reach itself

Explicit State Model checking

The basic Strategy



Temporal Logic Formula

Buchi Automata

Kripke structure

System Description

Product Automata

Model checker

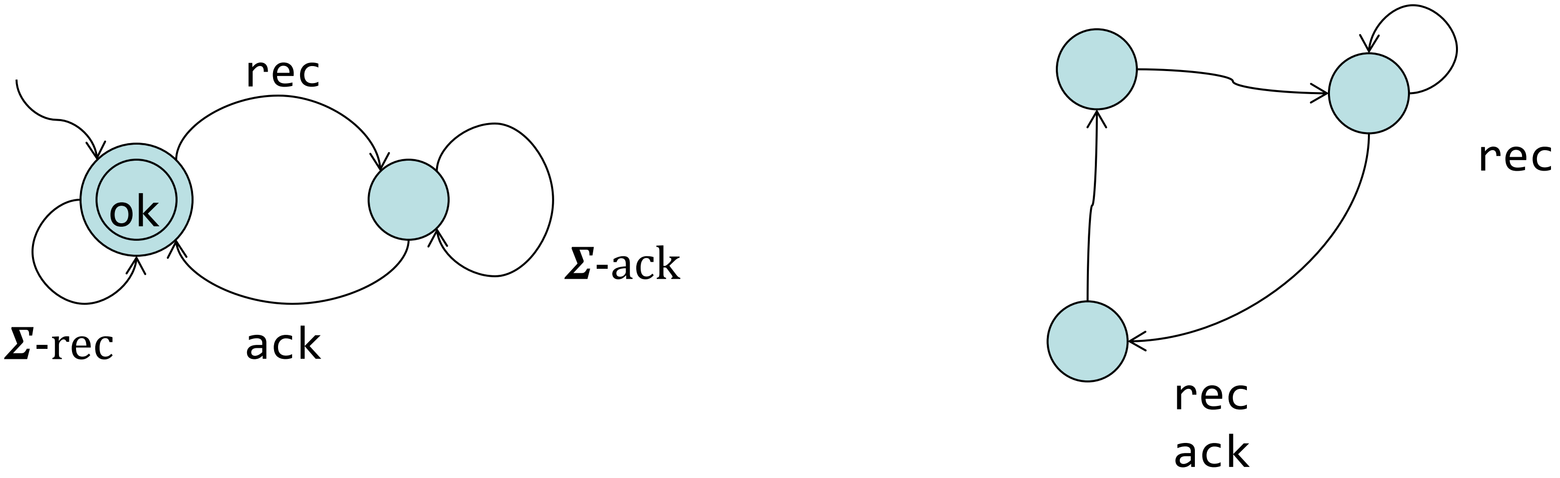
OK

Counterexample trace

DFS on state machine!

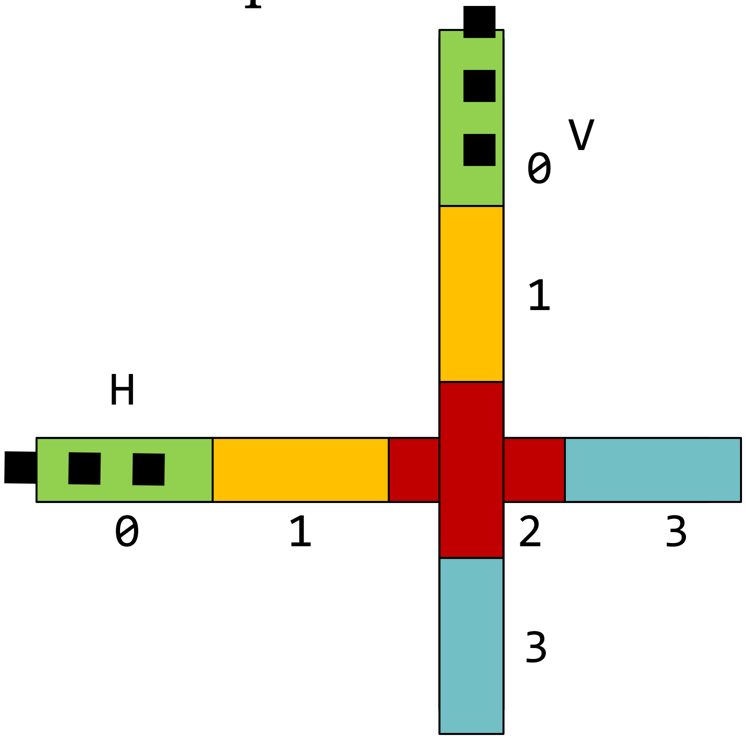
Example

`G rec → F ack`



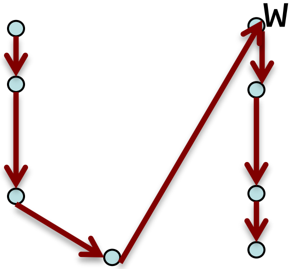
Optimizations: Partial Order Reduction

Example



```
       while(*){
pc=0  if(p=0){
pc=1      p:=1;
      }
pc=2  if(p=1){
pc=3    if(g=free){
pc=4        g:=id;
pc=5        p:=2;
        }
      }
pc=6  if(p=2){
pc=7    p:=3; g:=free
      }
pc=8  if(p=3){
pc=9    p:=0;
      }
    }
```

H train

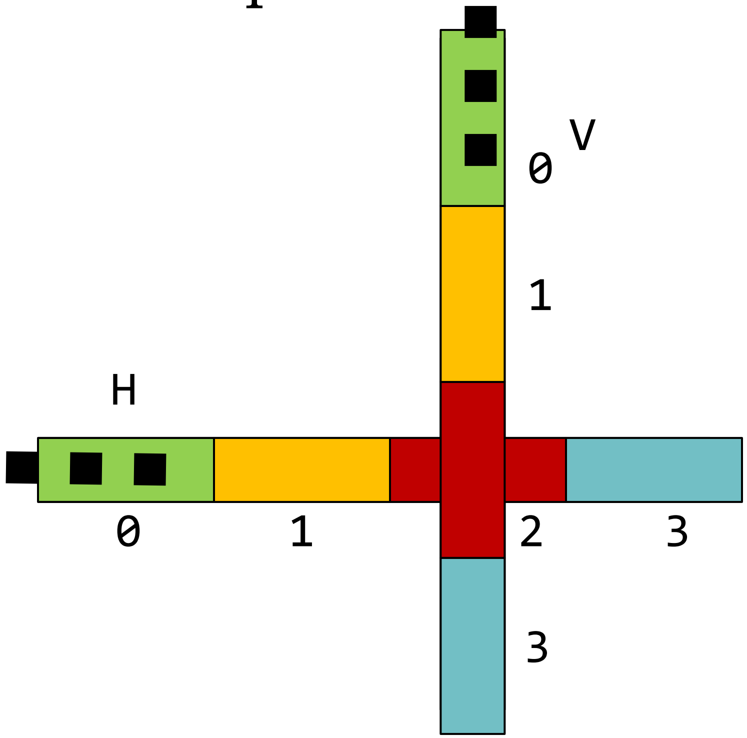


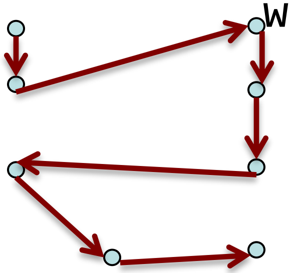
```
while(*){
  if(p=0){
      p:=1;
  }
  if(p=1){
    if(g=free){
        g:=id;
       p:=2;
    }
  }
  if(p=2){
    p:=3; g:=free
  }
  if(p=3){
    p:=0;
  }
}
```

V train

Optimizations: Partial Order Reduction

Example





```
    while(*){
pc=0  if(p=0){
pc=1     p:=1;
      }
pc=2  if(p=1){
pc=3    if(g=free){
pc=4       g:=id;
pc=5       p:=2;
        }
      }
pc=6  if(p=2){
pc=7    p:=3; g:=free
      }
pc=8  if(p=3){
pc=9    p:=0;
      }
    }
```

```
while(*){
  if(p=0){
     p:=1;
  }
  if(p=1){
    if(g=free){
       g:=id;
       p:=2;
    }
  }
  if(p=2){
    p:=3; g:=free
  }
  if(p=3){
    p:=0;
  }
}
```

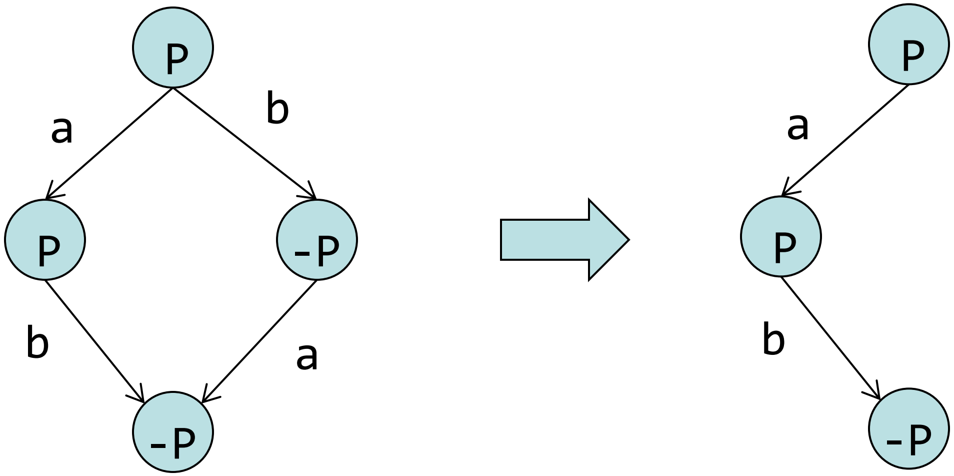
H train **V train**

Partial Order Reduction

Key idea:

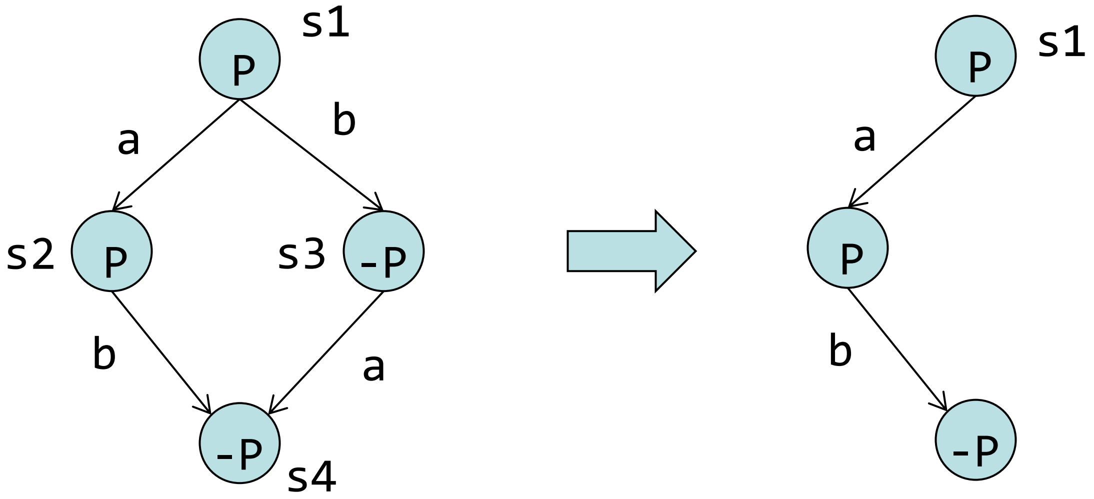
- The order of independent actions on different threads does not matter
- Note: what is considered independent depends on the property

$F\neg p$



Ample set

$F\neg p$



On state s1, the transitions to s2 and s3 are both enabled.

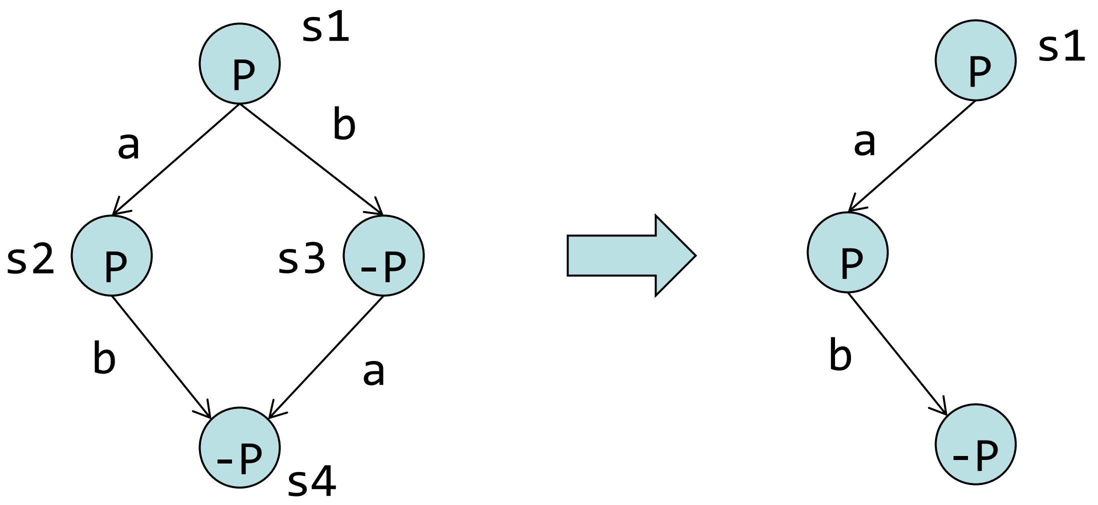
- enabled(s1)

We only want to explore a subset of the enabled set

- ample(s1) ⊆ enabled(s1)

Ample set

$F \neg p$



We have 3 goals in computing *ample(s)*

- Using ample instead of enabled should give us a much smaller graph
- Using ample instead of enabled should still allow us to find what we are looking for
- Computing ample should be easy

Independence and Invisibility

Independence:

- Actions a and b are independent iff:
  - a does not disable b and vice-versa
  - Commutativity: a(b(s)) = b(a(s))

Invisibility:

- a and b should not affect the values of any relevant property

Ample is computed heuristically

Computing it precisely is too hard, but we can find actions that are definitely not in ample(s) and can therefore be ignored.

What we need to consider:

- Actions that share variables with the property
- If two actions share variables, they are dependent
- If two actions appear in the same thread they are dependent

6.820 Fundamentals of Program Analysis
Fall 2015