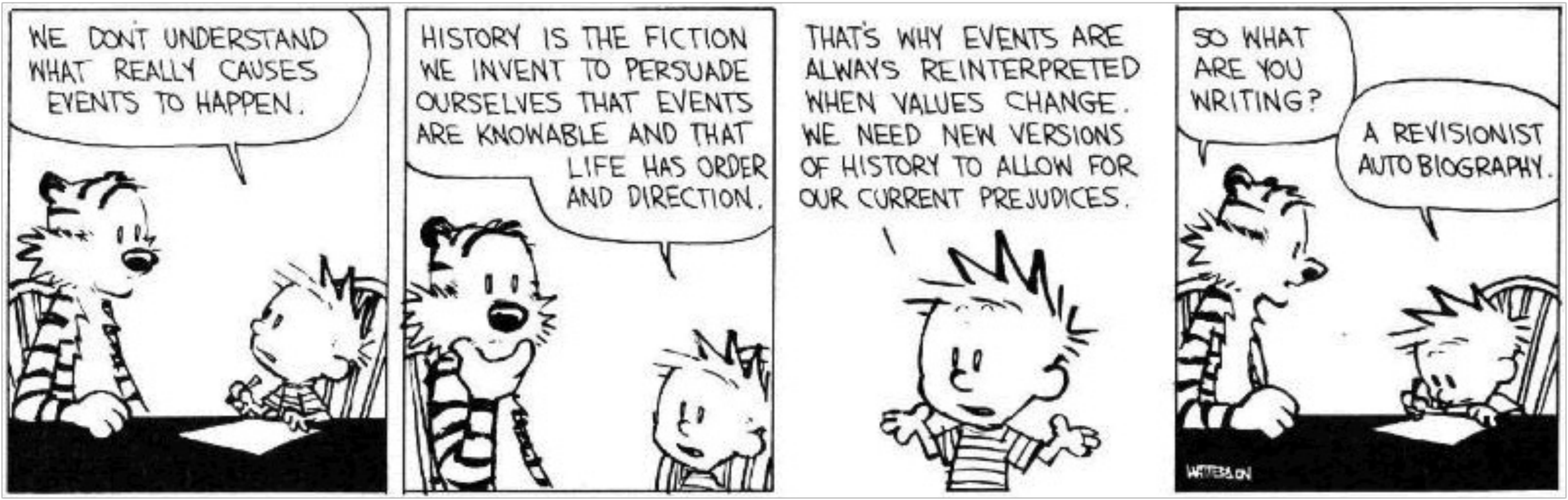
Model Checking: An Introduction

Meetings 2-4, CSCI 5535, Fall 2023



Week 1

- Homework 0 (“Preliminaries”) out, due next Friday

- Today
  - Skim an application motivating CSCI 5535
- Next Week
  - Begin foundations

Course Summary

Course At-A-Glance

- Part I: Language Specification and Design
  - Semantics = Describing programs
  - Evaluation strategies, imperative languages
  - Textbooks:
    - Robert Harper. *Practical Foundations of Programming Languages.*
    - Glynn Winskel. *The Formal Semantics of Programming Languages.*
- Part II: Applications

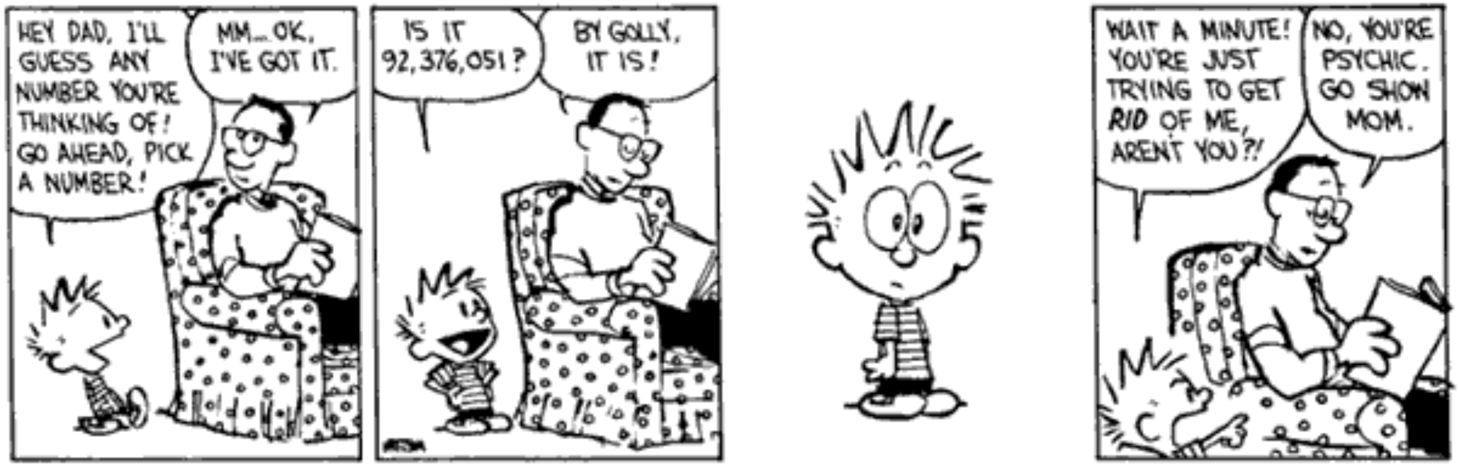
Core Topics

- Semantics
  - Operational semantics and types
    - rules for execution on an abstract machine
    - useful for implementing a compiler or interpreter
  - Axiomatic semantics
    - logical rules for reasoning about the behavior of a program
    - useful for proving program correctness
  - Abstract interpretation
    - application: program analysis

But first …

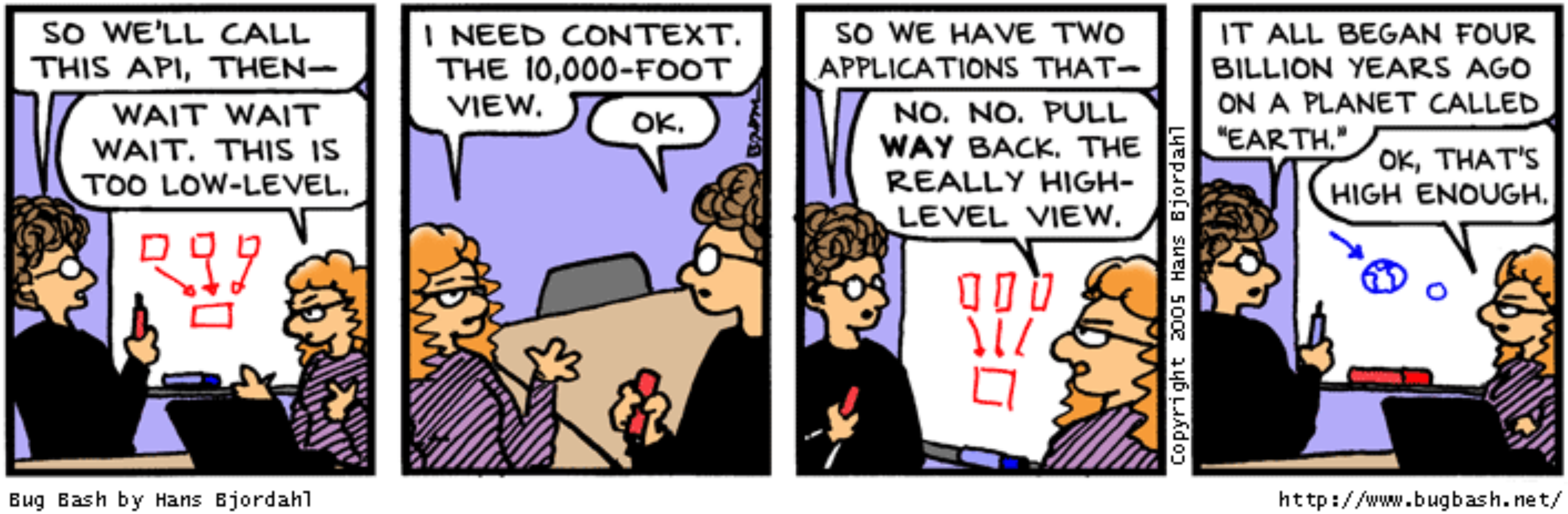
First Topic: Model Checking

- Verify properties or find bugs in software
- Take an important program (e.g., a device driver)
- Merge it with a property (e.g., no deadlocks)
- Transform the result into a boolean program
- Use a model checker to exhaustively explore the resulting state space
  - Result 1: program provably satisfies property
  - Result 2: program violates property "right here on line 92,376"!



Who are we again?

- We're going to find critical bugs in important bits of software
  - using PL techniques!
- You'll be enthusiastic about this
  - and thus want to learn the gritty details



Overarching Plan

Model Checking

- Transition systems (i.e., models)
- Temporal properties
- Temporal logics: LTL and CTL
- Explicit-state model checking
- Symbolic model checking

Counterexample Guided Abstraction Refinement

- Safety properties
- Predicate abstraction
- Software model checking
- Counterexample feasibility
- Abstraction refinement weakest pre, thrm prv

Spoiler

- This stuff really works!

- Symbolic model checking is a massive success in the model-checking field
- SLAM took the PL world by storm
  - Spawned multiple copycat projects
  - Launched Microsoft's Static Driver Verifier (released in the Windows DDK)

Model Checking

There are complete courses in model checking (see ECEN 5139, Prof. Somenzi).

Model Checking by Edmund M. Clarke, Orna Grumberg, and Doron A. Peled.
Symbolic Model Checking by Ken McMillan.

We will skim.

What is Model Checking? Keywords?

What is Model Checking?

Keywords

Model checking is an automated technique

Model checking verifies transition systems

Model checking verifies temporal properties

Model checking falsifies by generating counterexamples

A model checker is a program that checks if a (transition) system satisfies a (temporal) property

Verification vs. Falsification

- What is verification?

proof that a property holds on a system

(all executions)

an error

absent of that error

- What is falsification?

proof that a property doesn't hold

→ a witness that an error is possible

Verification vs. Falsification

- An automated verification tool
  - can report that the system is verified (with a proof);
  - or that the system was not verified.

- When the system was not verified, it would be helpful to explain why.
  - Model checkers can output an error counterexample: a concrete execution scenario that demonstrates the error.
- Can view a model checker as a falsification tool
  - The main goal is to find bugs

- So what can we verify or falsify?

Temporal Properties

Temporal Property

A property with time-related operators such as "invariant" or "eventually"

Invariant(p)

is true in a state if property p is true in **every** state on all execution paths starting at that state

G, AG, □ ("globally" or "box" or "forall")

Eventually(p)

is true in a state if property p is true at **some** state on every execution path starting from that state

F, AF, ◇ ("future" or "diamond" or "exists")

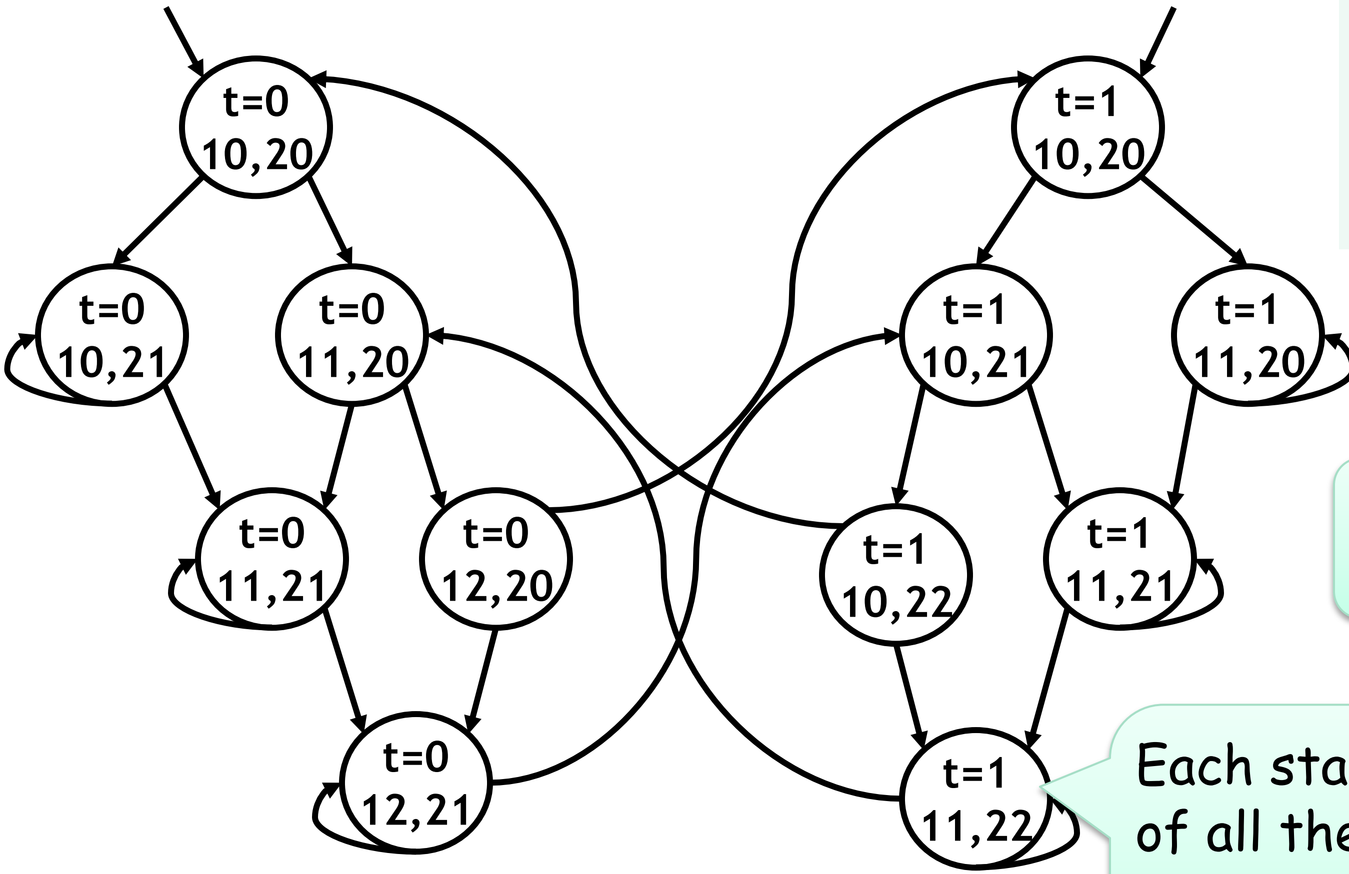
An Example Concurrent Program

- A simple concurrent mutual exclusion program
- Two processes execute asynchronously
- There is a shared variable `turn`
- Two processes use the shared variable to ensure that they are not in the critical section at the same time
- Can be viewed as a "fundamental" program: any bigger concurrent one would include this one

```
10: while (true) {
11:   wait(turn == 0);
      // critical section
12:   work(); turn = 1;
13: }

|| // concurrently with

20: while (true) {
21:   wait(turn == 1);
      // critical section
22:   work(); turn = 0;
23: }
```

Reachable States of the Example Program

```
10: while (true) {
11:   wait(turn == 0);
      // critical section
12:   work(); turn = 1;
13: }

|| // concurrently with

20: while (true) {
21:   wait(turn == 1);
      // critical section
22:   work(); turn = 0;
23: }
```



Next: formalize this intuition …

Each state is a valuation of all the variables: `turn` and the two program counters for two processes

Analyzed System is a Transition System

- Labeled transition system

  $T = (S, I, R, L)$

  Also called a Kripke Structure

  - S = Set of states // standard FSM
  - $I \subseteq S$ = Set of initial states // standard FSM
  - $R \subseteq S \times S$ = Transition relation // standard FSM
  - $L: S \rightarrow 2^{AP}$ = Labeling function // this is new!
- AP: Set of atomic propositions (e.g., "x=5"$\in$AP)
  - Atomic propositions capture basic properties
  - For software, atomic props depend on variable values
  - The labeling function labels each state with the set of propositions true in that state

Example Properties of the Program

- "In all the reachable states (configurations) of the system, the two processes are *never in the critical section at the same time*"
  - "pc1=12", "pc2=22" are atomic properties for being in the critical section

- "*Eventually the first process enters* the critical section"

Temporal Logics

For what?

There are four basic temporal operators:

- X p

  Next p, p holds in the next state
- G p

  Globally p, p holds in every state, p is an invariant
- F p

  Future p, p will hold in a future state, p holds eventually
- p U q

  p Until q, assertion p will hold until q holds
- Precise meaning of these temporal operators are defined on execution paths

Execution Paths

- A path in a transition system is an infinite sequence of states
  $(s_0, s_1, s_2, \ldots)$, such that $\forall i \geq 0.\ (s_i, s_{i+1}) \in R$
- A path $(s_0, s_1, s_2, \ldots)$ is an execution path if $s_0 \in I$
- Given a path $h = (s_0, s_1, s_2, \ldots)$
  - h_i denotes the i^{th} state: s_i
  - h^i denotes the i^{th} suffix: $(s_i, s_{i+1}, s_{i+2}, \ldots)$
- In some temporal logics one can quantify paths starting from a state using path quantifiers
  - A : for all paths (e.g., A h. ....)
  - E : there exists a path (e.g., E h. ...)

Paths and Predicates

- We write

  models

  $$h \models p$$

  "the path h makes the predicate p true"
  - h is a path in a transition system
  - p is a temporal logic predicate

- Example:

  $$A\ h.\quad h \models G\,(\neg(pc1{=}12 \wedge pc2{=}22))$$

Linear Time Logic (LTL)

- LTL properties are constructed from atomic propositions in AP; logical operators $\wedge$, $\vee$, $\neg$; and temporal operators X, G, F, U.
- The semantics of LTL is defined on paths.

Given a path h:

$$h \models p$$

Linear Time Logic (LTL)

h_i denotes the i^{th} state: s_i
h^i denotes the i^{th} suffix: $(s_i, s_{i+1}, s_{i+2}, ...)$

Given a path h:

$h \models ap$	iff	$L(h_0, ap)$	*atomic prop*
$h \models X\ p$	iff	$h^1 \models p$	*next*
$h \models F\ p$	iff		
$h \models G\ p$	iff		
$h \models p\ U\ q$	iff		

Satisfying Linear Time Logic

- Given a transition system T = (S, I, R, L) and an LTL property p, T satisfies p if all paths starting from all initial states I satisfy p

Computation Tree Logic (CTL)

- In CTL temporal properties use path quantifiers: A : for all paths, E : there exists a path
- The semantics of CTL is defined on states:

Given a state s

$s \models ap$	iff	$L(s, ap)$
$s_0 \models EX\ p$	iff	$\exists$ a path $(s_0, s_1, s_2, \ldots)$. $s_1 \models p$
$s_0 \models AX\ p$	iff	$\forall$ paths $(s_0, s_1, s_2, \ldots)$. $s_1 \models p$
$s_0 \models EG\ p$	iff	$\exists$ a path $(s_0, s_1, s_2, \ldots)$. $\forall i \geq 0$. $s_i \models p$
$s_0 \models AG\ p$	iff	$\forall$ paths $(s_0, s_1, s_2, \ldots)$. $\forall i \geq 0$. $s_i \models p$

…

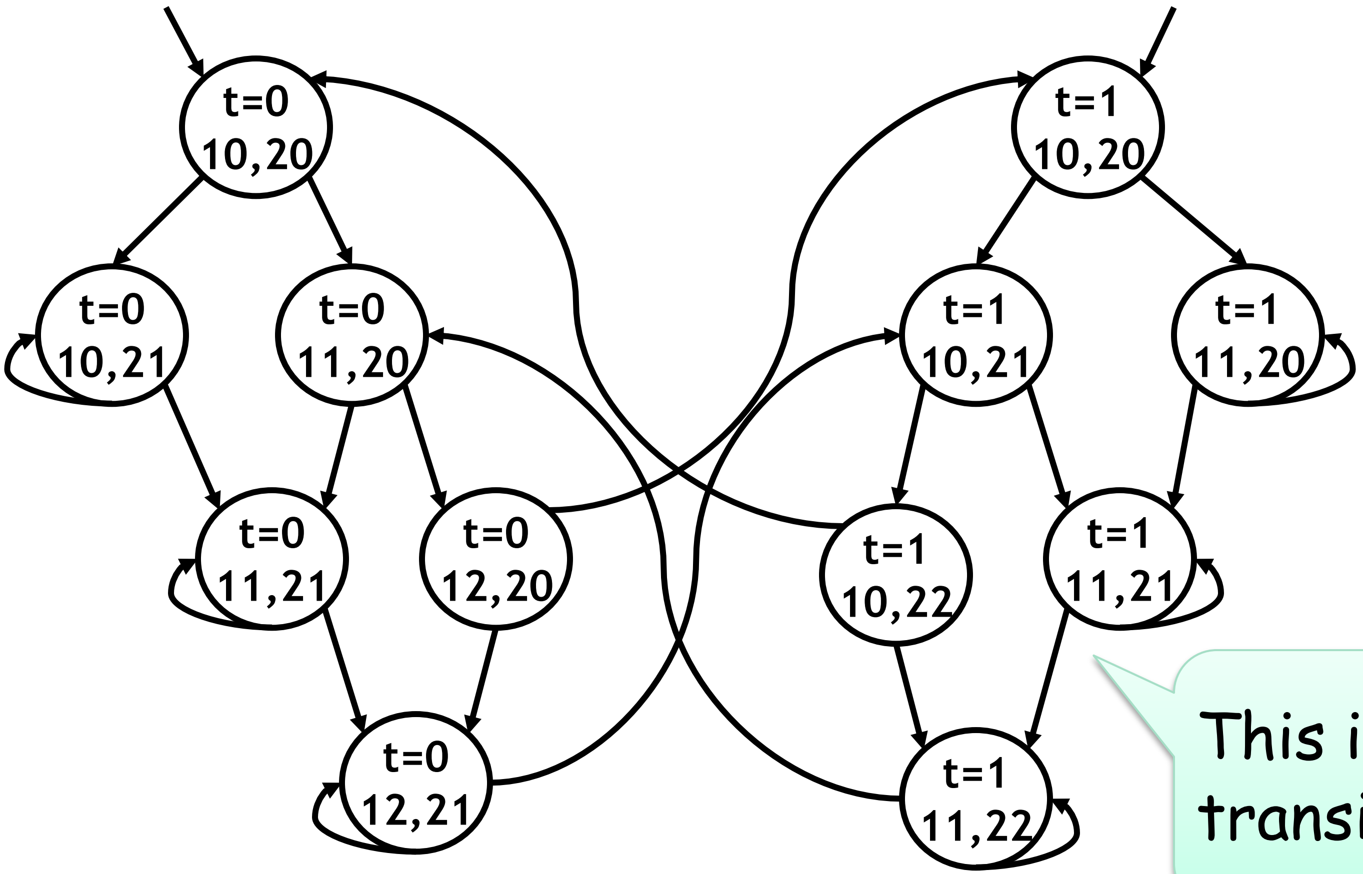
Linear vs. Branching Time

- LTL is a linear time logic
  - When determining if a path satisfies an LTL formula we are only concerned with a single path
- CTL is a branching time logic
  - When determining if a state satisfies a CTL formula we are concerned with multiple paths
  - In CTL the computation is instead viewed as a computation tree which contains all the paths

The expressive powers of CTL and LTL are incomparable ($LTL \subseteq CTL^*$, $CTL \subseteq CTL^*$)

- Basic temporal properties can be expressed in both logics
- Not in this lecture, sorry! (Take a class on Modal Logics)

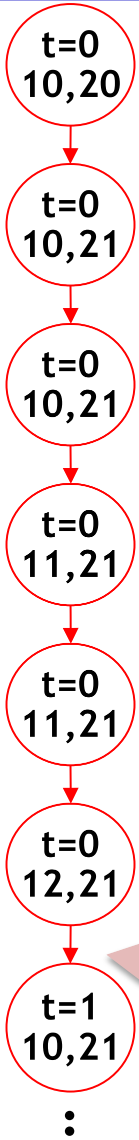
Recall the Example



This is a labeled transition system

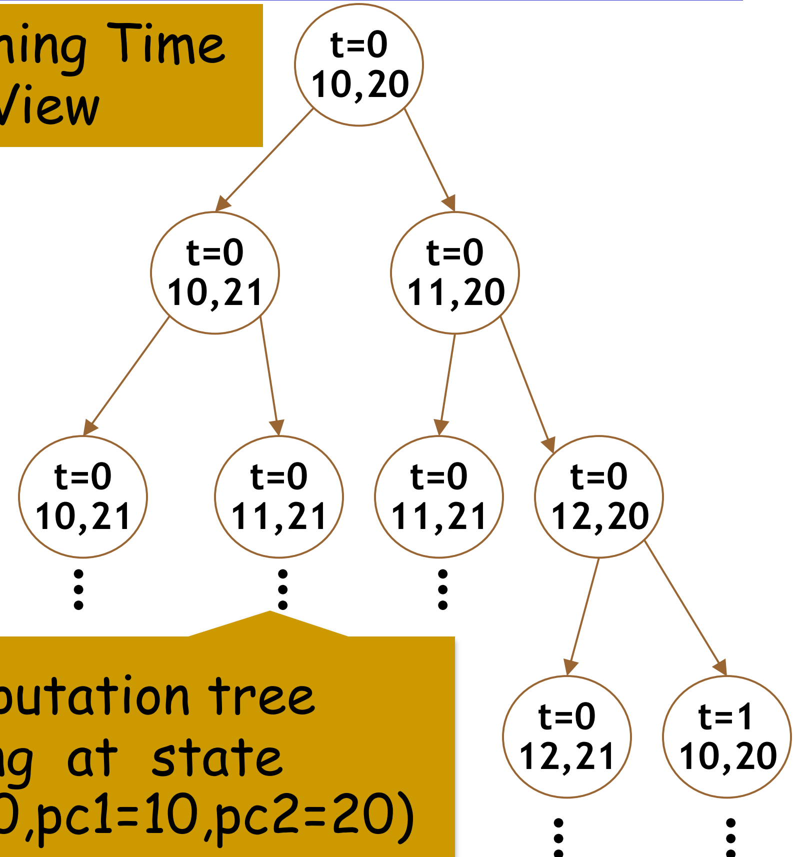
Linear vs. Branching Time

Linear Time View



One path starting at state (turn=0,pc1=10,pc2=20)

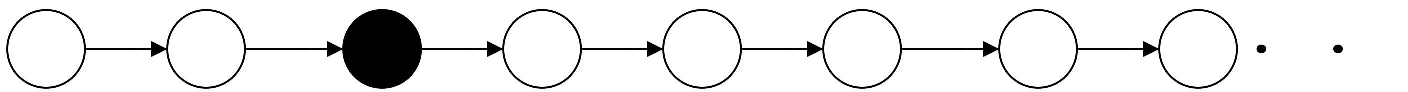
Branching Time View



A computation tree starting at state (turn=0,pc1=10,pc2=20)

LTL Satisfiability Examples

○ p does not hold ● p holds

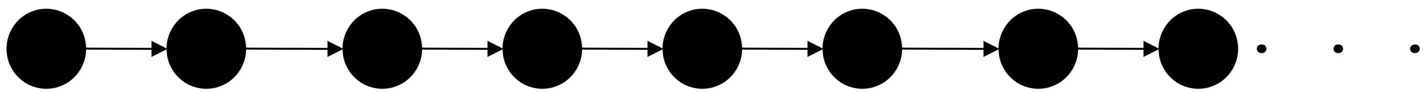


On this path:

Holds	Does Not Hold
$F p$	p
$X X p$	$G p$
	$X p$

LTL Satisfiability Examples

○ p does not hold ● p holds



On this path:

Holds

Gp

Fp

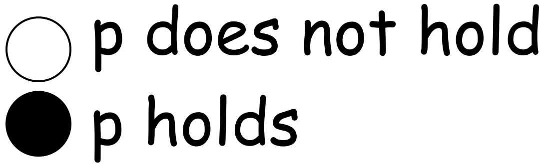
p

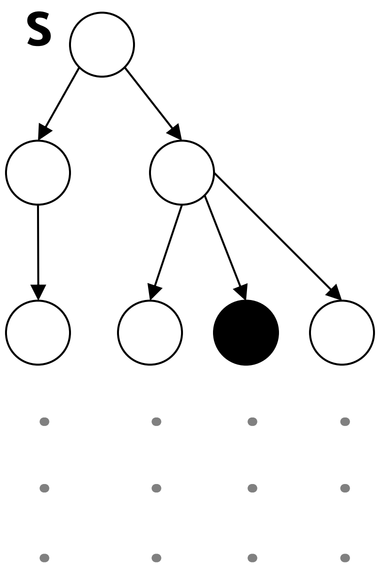
Does Not Hold

$\neg p$

$G \neg p$

CTL Satisfiability Examples



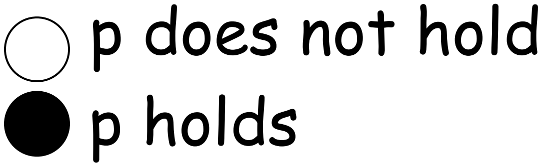


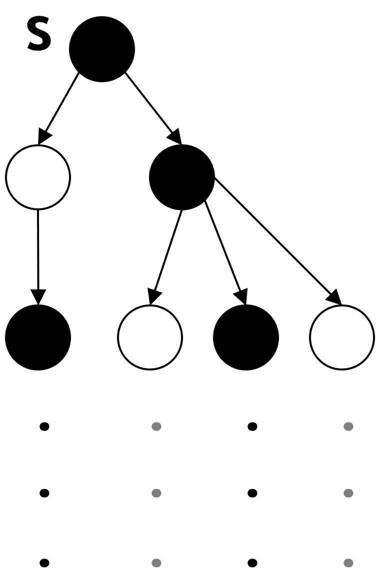
At state s:

Holds

Does Not Hold

CTL Satisfiability Examples





At state s:

Holds

Does Not Hold

Model Checking Complexity

- Given a transition system $T = (S, I, R, L)$ and a CTL formula f
  - One can check if a state of the transition system satisfies the formula f in

    $O(|f| \times (|S| + |R|))$ time
  - Multiple depth first searches (one for each temporal operator)
    - explicit-state model checking

State Space Explosion

- The complexity of model checking increases linearly with respect to the size of the transition system $(|S| + |R|)$
- However, the size of the transition system $(|S| + |R|)$ is **exponential** in the number of variables and number of concurrent processes
- This exponential increase in the state space is called the state space explosion
  - Dealing with it is one of the major challenges in model checking research

Algorithm: Temporal Properties = Fixpoints

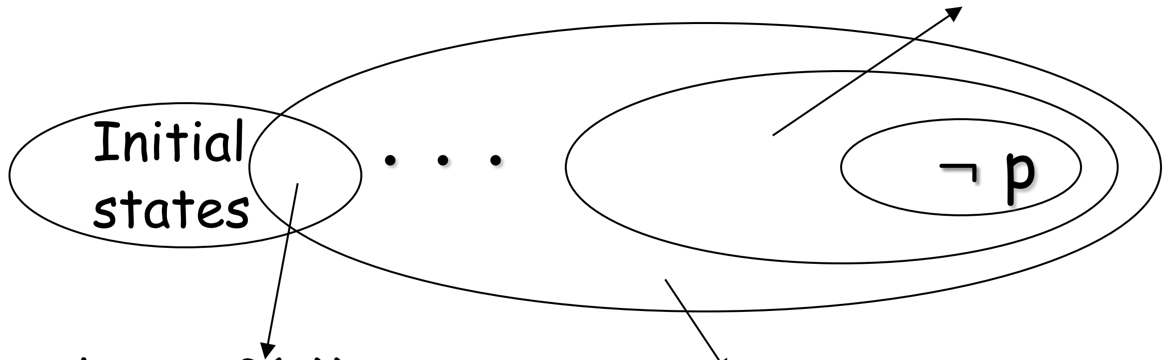
- States that satisfy AG(p) are all the states which are not in EF($\neg$p) (= the states that can reach $\neg$p)

- Compute EF($\neg$p) as the **fixed point** of Func: $2^S \rightarrow 2^S$
- Given $Z \subseteq S$,
  - Func(Z) = $\neg p \cup$ **reach-in-one-step**(Z)

- Actually, EF($\neg$p) is the ***least*-fixed point** of Func
  - smallest set Z such that Z = Func(Z)
  - to compute the least fixed point, start the iteration from $Z=\emptyset$, and apply the Func until you reach a fixed point
  - This can be computed (unlike most other fixed points)

Pictorial Backward Fixed Point

(Inverse Image of $\neg p$) = EX($\neg p$)



(initial states that violate *AG*(p)) = (initial states that satisfy EF($\neg p$))

(states that can reach $\neg p$ = EF($\neg p$)) = (states that violate *AG*(p))

This fixed point computation can be used for:

- verification of EF(¬p)
- or falsification of *AG*(p)

... and similar fixed points handle the other cases

Symbolic Model Checking

- Symbolic model checking represent state sets and the transition relation as Boolean logic formulas
  - Fixed point computations manipulate sets of states rather than individual states
  - Recall: we needed to compute reach-in-one-step(Z), but $Z \subseteq S$
- Fixed points can be computed by iteratively manipulating these formulas
- Use an efficient data structure for manipulation of Boolean logic formulas
  - Binary Decision Diagrams (BDDs)
- SMV (Symbolic Model Verifier) was the first CTL model checker to use BDDs

Building Up To Software Model Checking via Counterexample Guided Abstraction Refinement

There are easily dozens of papers.

We will skim.

Key Terms

- Counterexample guided abstraction refinement (CEGAR)
  - A successful software model-checking approach. Sometimes called "Iterative Abstraction Refinement".
- SLAM = The first CEGAR project/tool.
  - Developed at MSR
- Lazy Abstraction = CEGAR optimization
  - Used in the BLAST tool from Berkeley.

What *is* Counterexample Guided Abstraction Refinement (CEGAR)?

Verification by …

Model Checking?

Theorem Proving?

Dataflow Analysis or Program Analysis?

Verification

```
Example ( ) {
1: do{
        lock();
        old = new;
        q = q->next;
2:      if (q != NULL){
3:         q->data = new;
           unlock();
           new ++;
        }
4: } while(new != old);
5: unlock();
   return;
}
```

Is this program correct?

What does correct mean?

How do we determine if a program is correct?

Verification by Model Checking

```
Example ( ) {
1: do{
        lock();
        old = new;
        q = q->next;
2:      if (q != NULL){
3:          q->data = new;
            unlock();
            new ++;
        }
4: } while(new != old);
5: unlock();
   return;
}
```

1. (Finite State) Program
2. State Transition Graph
3. Reachability

- Program→Finite state model
- State explosion
+ State exploration
+ Counterexamples

Precise [SPIN,SMV,Bandera,JPF]

Verification by Theorem Proving

```
Example ( ) {
1: do{
         lock();
         old = new;
         q = q->next;
2:       if (q != NULL){
3:            q->data = new;
              unlock();
              new ++;
         }
4: } while(new != old);
5: unlock();
   return;
}
```

1. Loop Invariants
2. Logical Formulas
3. Check Validity

Invariant:

$$\text{lock} \wedge \text{new} = \text{old}$$
$$\vee$$
$$\neg\text{lock} \wedge \text{new} \neq \text{old}$$

Verification by Theorem Proving

```
Example ( ) {
1: do{
        lock();
        old = new;
        q = q->next;
2:      if (q != NULL){
3:          q->data = new;
            unlock();
            new ++;
        }
4: } while(new != old);
5: unlock();
   return;
}
```

1. Loop Invariants
2. Logical Formulas
3. Check Validity

- Loop invariants
- Multithreaded programs

+ Behaviors encoded in logic
+ Decision procedures

Precise [ESC,PCC]

Verification by Program Analysis

```
Example ( ) {
1: do{
        lock();
        old = new;
        q = q->next;
2:      if (q != NULL){
3:          q->data = new;
            unlock();
            new ++;
        }
4: } while(new != old);
5: unlock();
   return;
}
```

1. Dataflow Facts
2. Constraint System
3. Solve Constraints

- Imprecision: fixed facts
+ Abstraction
+ Type/Flow analyses

Scalable [Cqual, ESP]

Combining Strengths

Theorem Proving

- Need loop invariants
(will find automatically)
+ Behaviors encoded in logic
(used to refine abstraction)
+ Theorem provers
(used to compute successors, refine abstraction)

Program Analysis

- Imprecise
(will be precise)
+ Abstraction
(will shrink the state space we must explore)

SLAM

Model Checking

- Finite-state model, state explosion
(will find small good model)
+ State space exploration
(used to get a path sensitive analysis)
+ Counterexamples
(used to find relevant facts, refine abstraction)

Software Model Checking via Counterexample Guided Abstraction Refinement

There are easily dozens of papers.

We will skim.

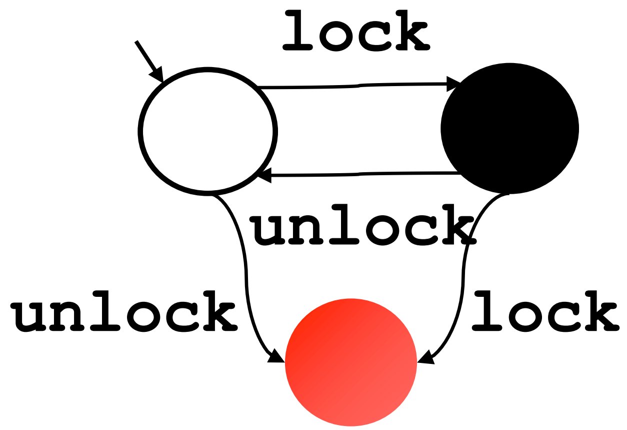
SLAM Overview

- Input: Program and Specification
  - Standard C Program (pointers, procedures)
  - Specification = Partial Correctness
    - Given as a finite state machine (typestate)
    - "I use locks correctly", not "I am a webserver"
- Output: Verified or Counterexample
  - Verified = program does not violate spec
    - Can come with proof!
  - Counterexample = concrete bug instance
    - A path through the program that violates the spec

Take-Home Message

- **SLAM** is a **software model checker**. It **abstracts** C programs to **boolean programs** and model-checks the boolean programs.
- No errors in the boolean program implies no errors in the original.
- An error in the boolean program **may** be a real bug. Or SLAM may **refine** the abstraction and start again.

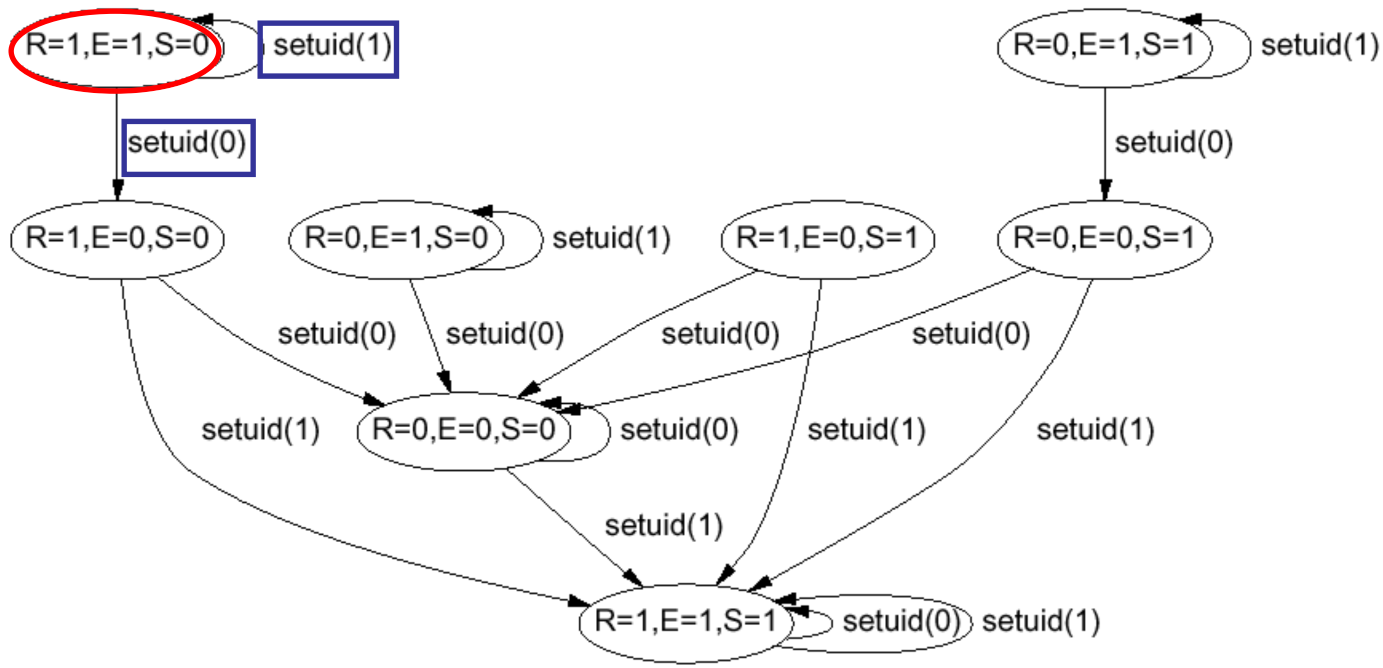
Property 1: Double Locking



"An attempt to re-acquire an acquired lock or release a released lock will cause a deadlock."

Calls to lock and unlock must alternate.

Property 2: Drop Root Privilege

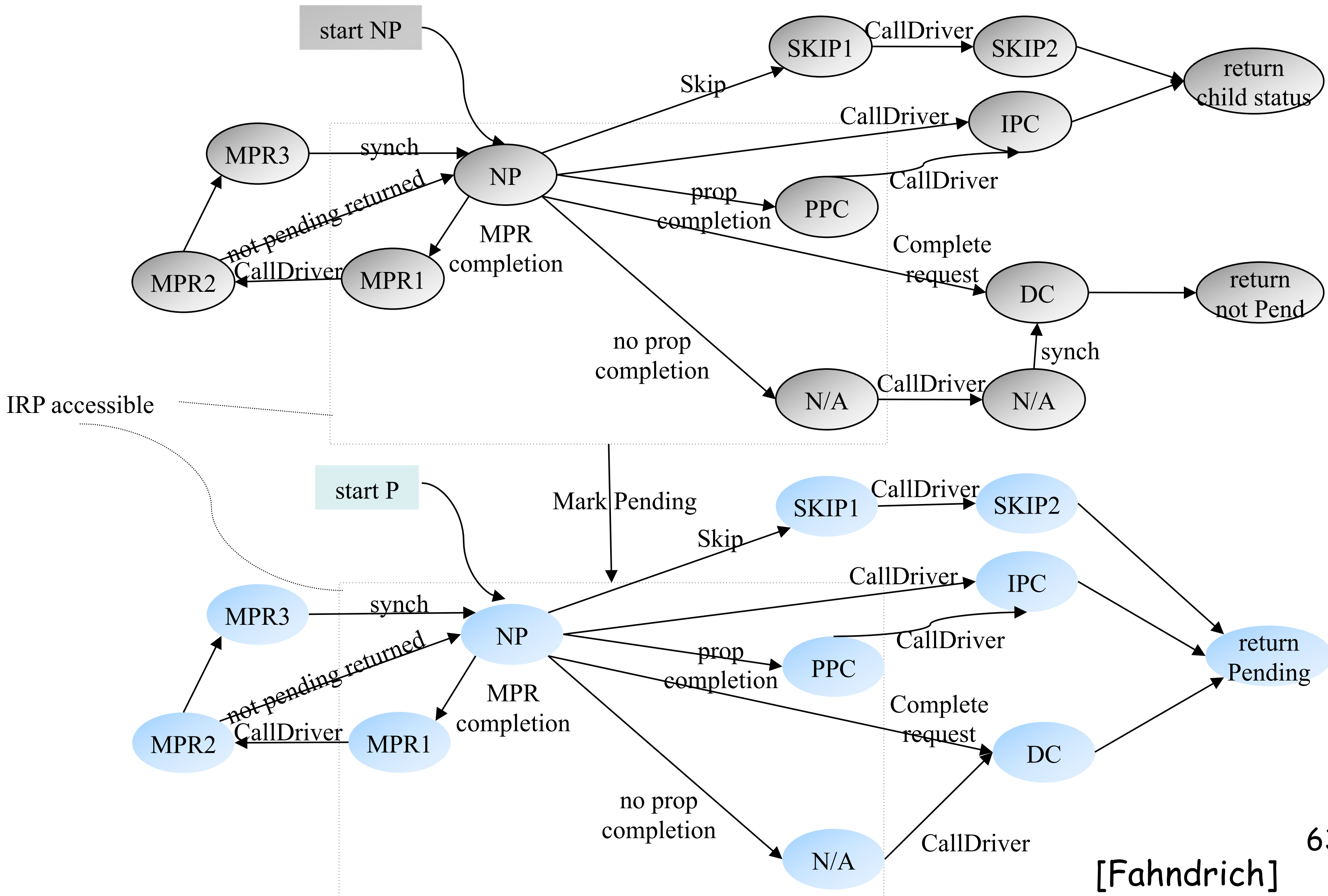


[Chen-Wagner-Dean '02]

"User applications must not run with root privilege"

When **execv** is called, must have **suid ≠ 0**

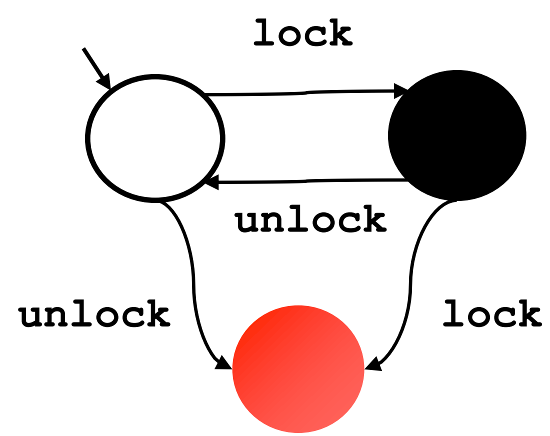
Property 3 : IRP Handler



[Fahndrich]

Example SLAM Input

```
Example ( ) {
1: do{
        lock();
        old = new;
q = q->next;
2:      if (q != NULL){
3:          q->data = new;

              unlock();
            new ++;
        }
4: } while(new != old);
5: unlock();
   return;
}
```



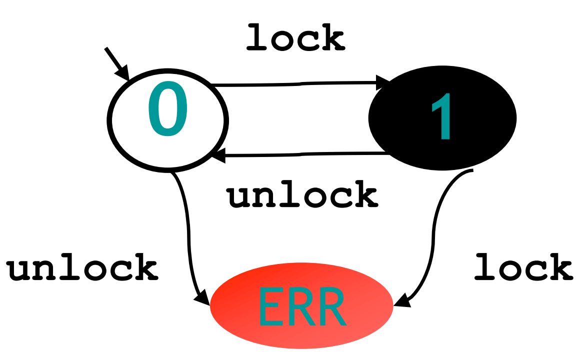
SLAM in a Nutshell

```
SLAM(Program p, Spec s) =
 Program q = incorporate_spec(p,s);              // slic
 PredicateSet abs = { };
 while true do
   BooleanProgram b = abstract(q,abs);           // c2bp
   match model_check(b) with                     // bebop
   | No_Error → print("no bug"); exit(0)
   | Counterexample(c) →
       if is_valid_path(c, p) then               // newton
         print("real bug"); exit(1)
       else
         abs ← abs ∪ new_preds(c)                // newton
 done
```

Incorporating Specs

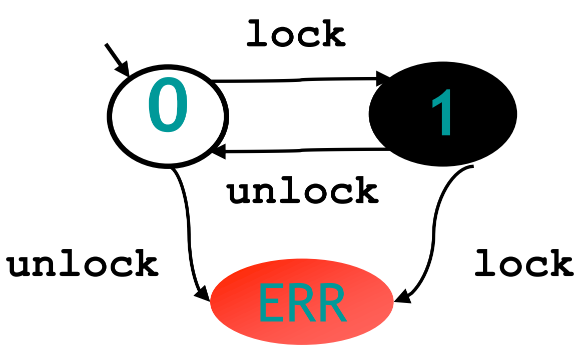
```
Example ( ) {
1: do{
       lock();
       old = new;
       q = q->next;
2:     if (q != NULL){
3:       q->data = new;
         unlock();
         new ++;
       }
4: } while(new != old);
5: unlock();
   return;
}
```

Ideas?



Incorporating Specs

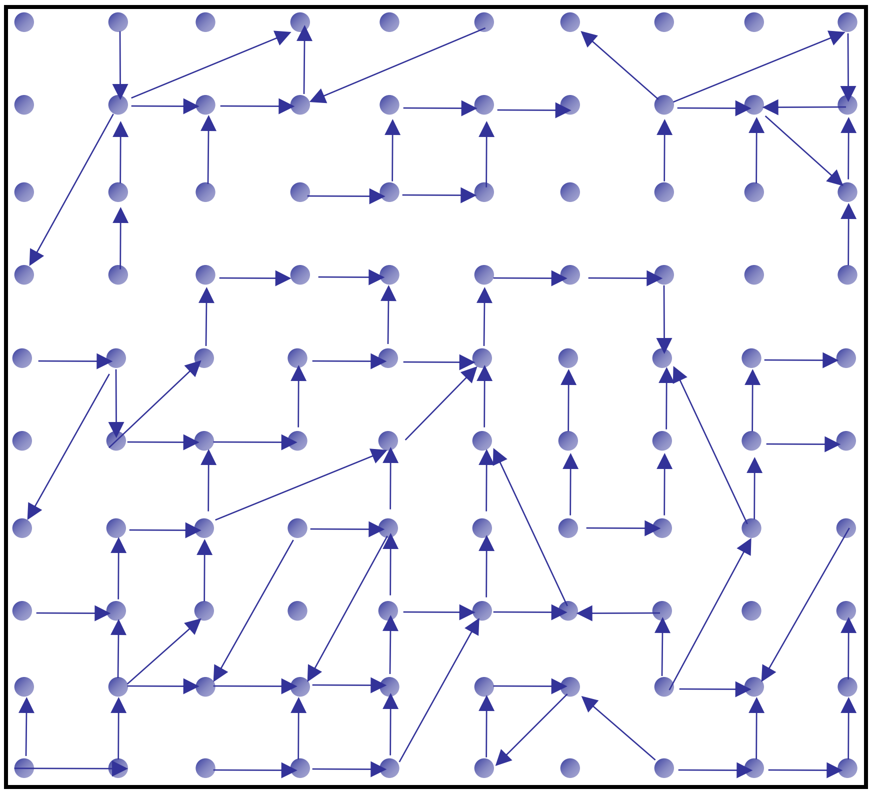
```
Example ( ) {
1: do{
       lock();
       old = new;
       q = q->next;
2:     if (q != NULL){
3:       q->data = new;
         unlock();
         new ++;
       }
4: } while(new != old);
5: unlock();
   return;
}
```

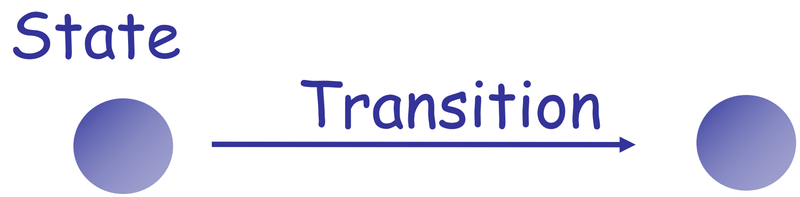



```
Example ( ) {
1: do{
       if L=1 goto ERR;
       else L=1;
       old = new;
       q = q->next;
2:     if (q != NULL){
3:         q->data = new;
           if L=0 goto ERR;
           else L=0;
           new ++;
       }
4: } while(new != old);
5: if L=0 goto ERR;
   else L=0;
   return;
ERR: abort();
}
```

Original program violates spec iff new program reaches ERR

Program As Labeled Transition System





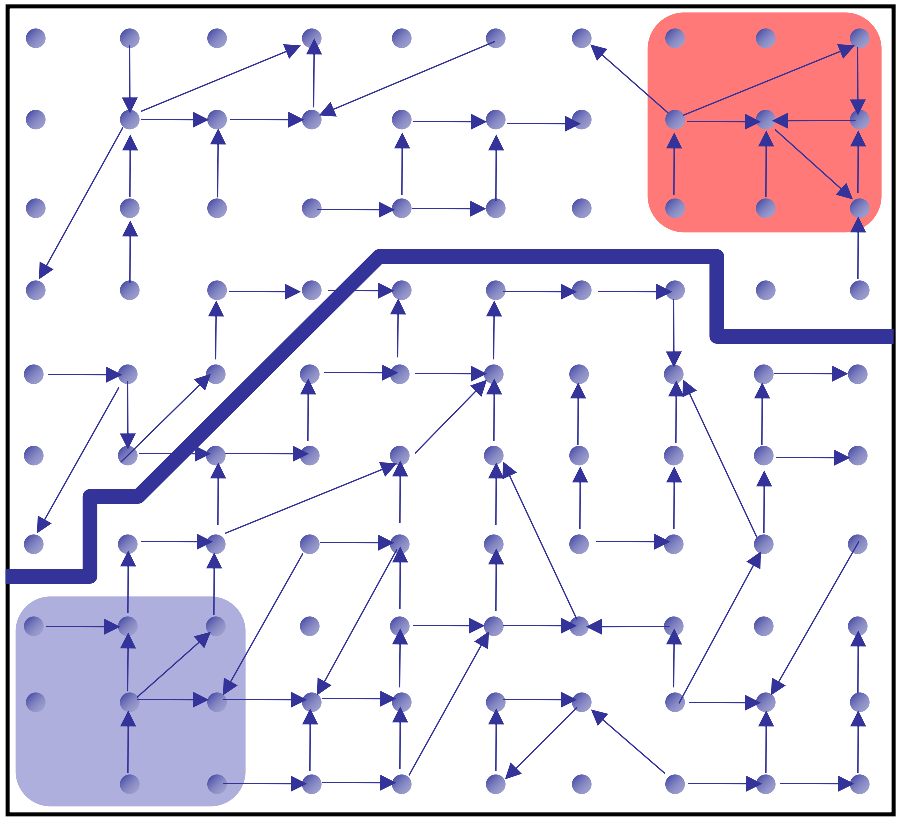
pc	$\mapsto$ 3
lock	$\mapsto$ ●
old	$\mapsto$ 5
new	$\mapsto$ 5
q	$\mapsto$ 0x133a

```
3:  unlock();
    new++;
4:}  …
```

pc	$\mapsto$ 4
lock	$\mapsto$ ○
old	$\mapsto$ 5
new	$\mapsto$ 6
q	$\mapsto$ 0x133a

```
Example ( ) {
1: do{
        lock();
        old = new;
        q = q->next;
2:      if (q != NULL){
3:           q->data = new;
             unlock();
             new ++;
        }
4: } while(new != old);
5: unlock();
   return;
}
```

The Safety Verification Problem



Error

(e.g., states with PC = Err)

Safe States

(never reach Error)

Initial

Is there a **path** from an **initial** to an **error** state ?

Problem? **Infinite** state graph (old=1, old=2, old=…)

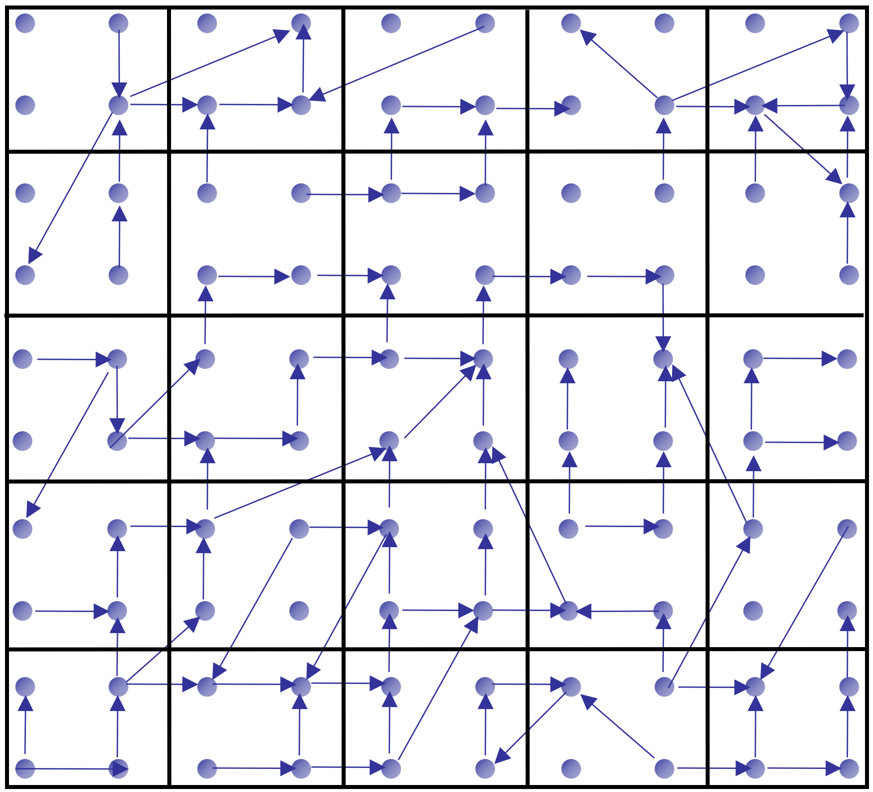
Solution? **Set** of states $\simeq$ logical **formula**

Representing [Sets of States] as Formulas

$[F]$ states satisfying F $\{s \mid s \models F\}$	F FO formula over program vars
$[F_1] \cap [F_2]$	$F_1 \wedge F_2$
$[F_1] \cup [F_2]$	
$\overline{[F]}$	
$[F_1] \subseteq [F_2]$	

i.e. $F_1 \wedge \neg F_2$ unsatisfiable

Idea 1: Predicate Abstraction

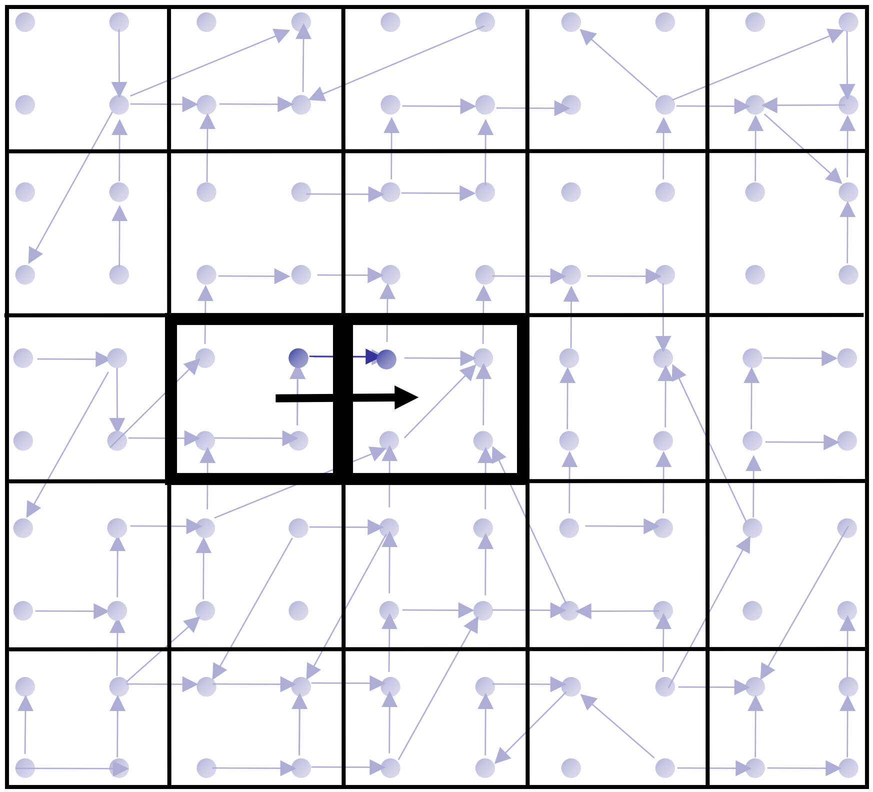


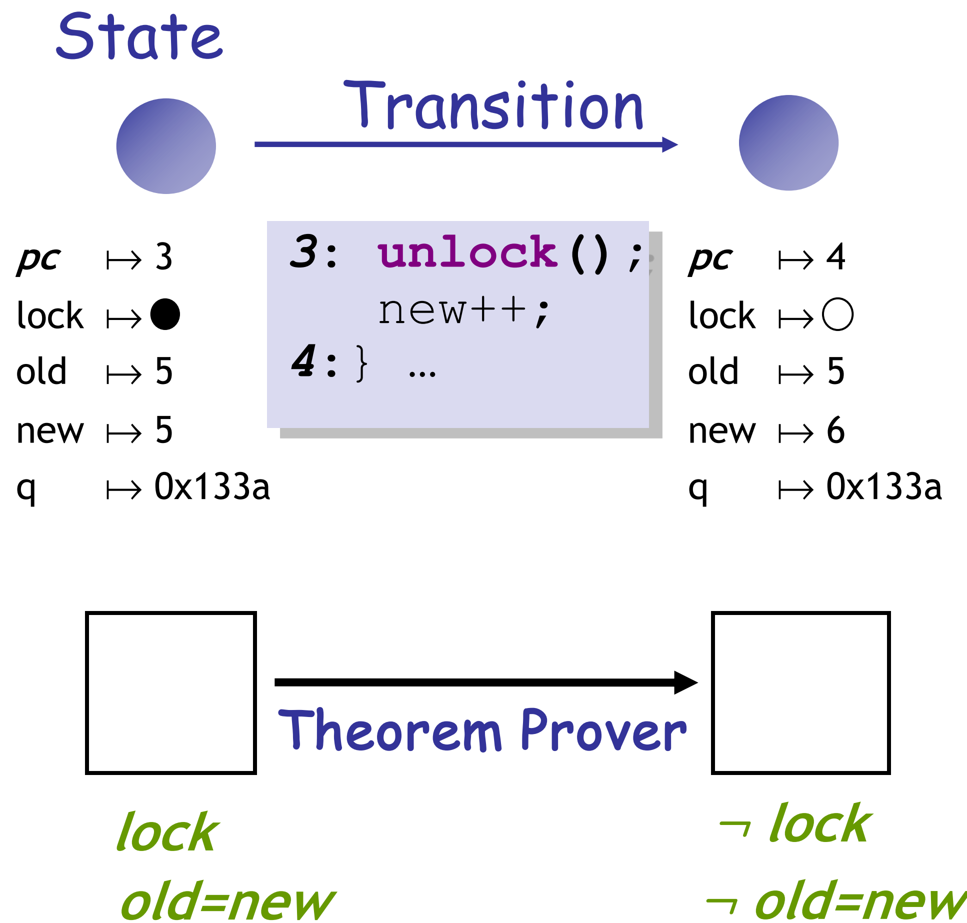
- Predicates on program state:
  lock (i.e., *lock=true*)
  old = new
- States satisfying same predicates are equivalent
  - Merged into one abstract state

Why?

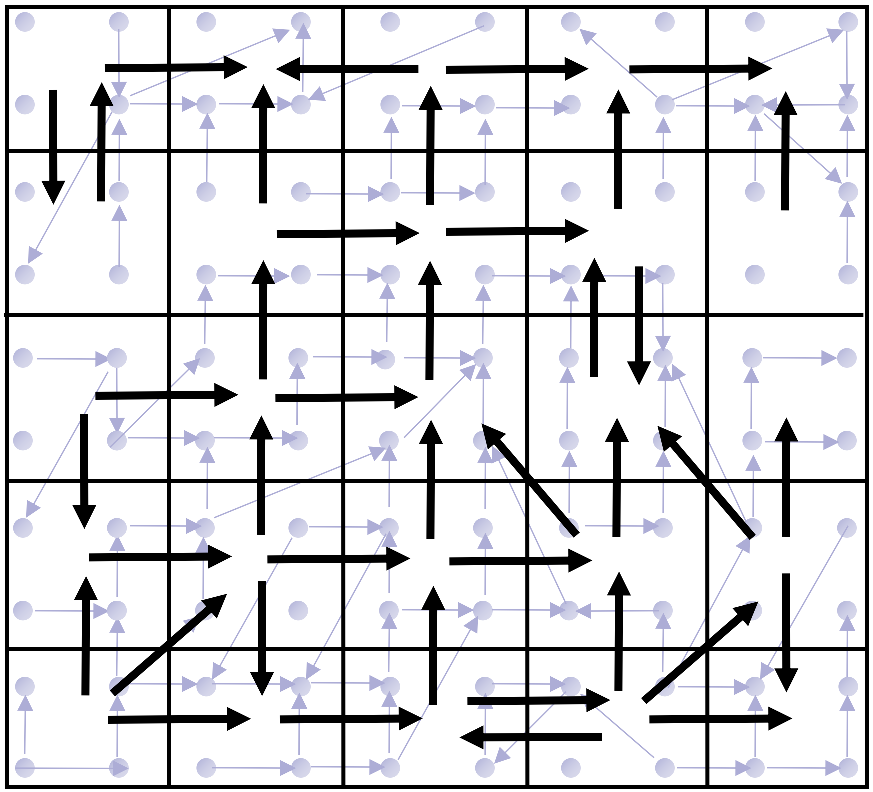
- Num of abstract states is finite
  - Thus model-checking the abstraction will be feasible!

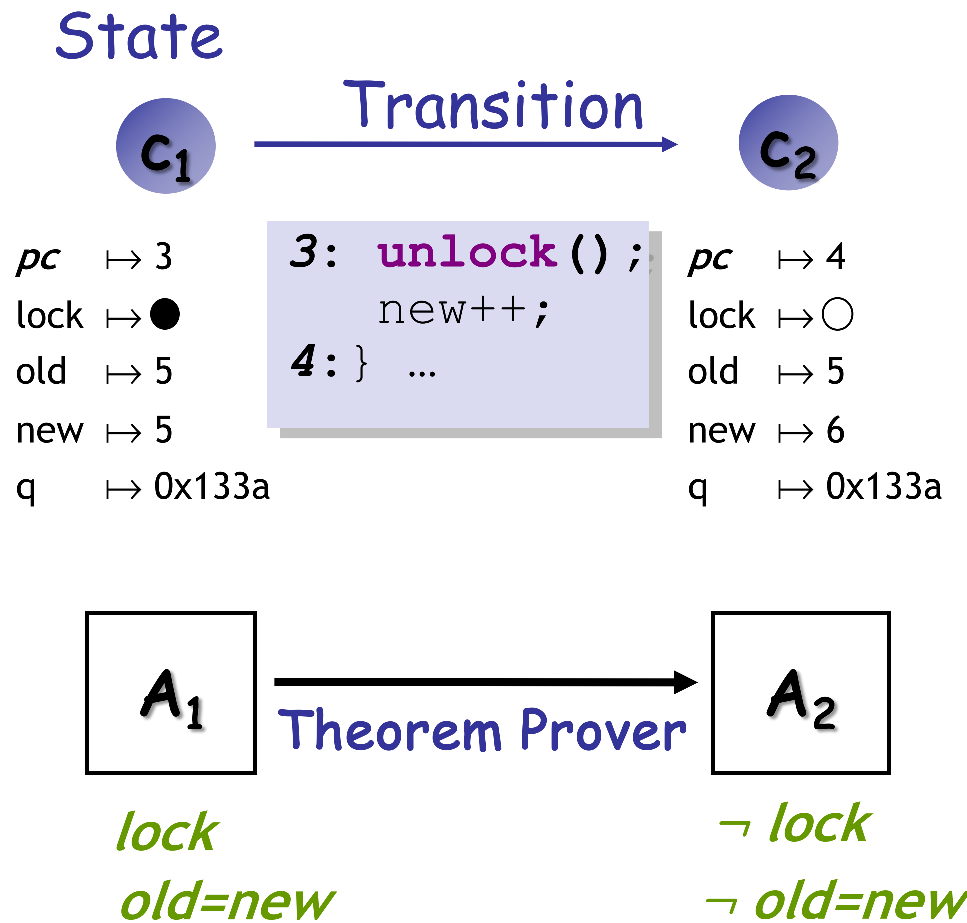
Abstract States and Transitions





Abstraction

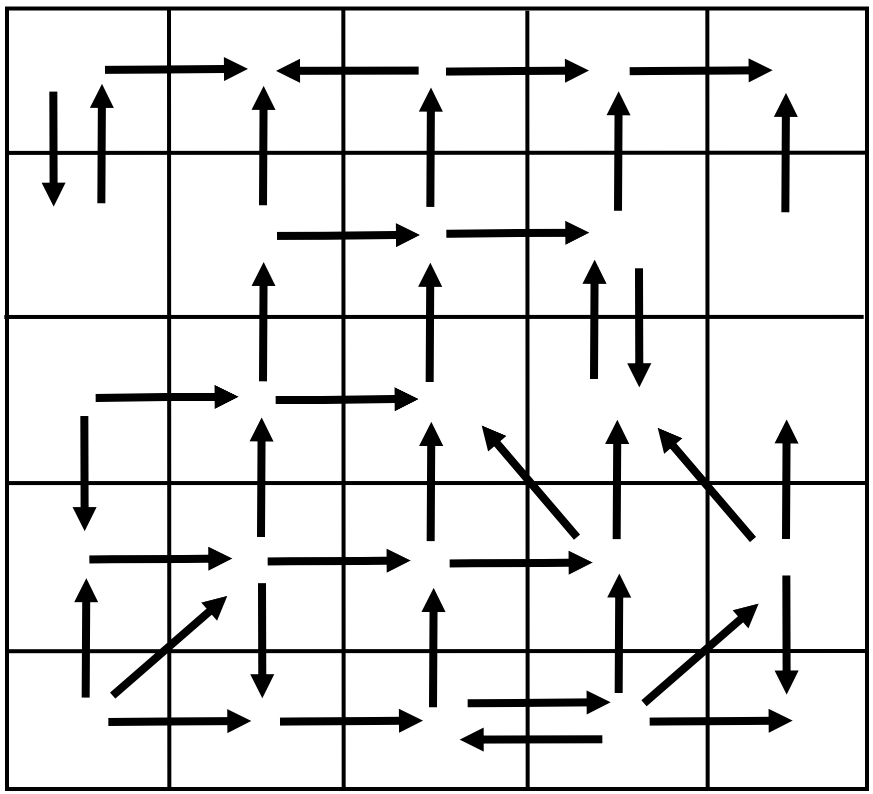


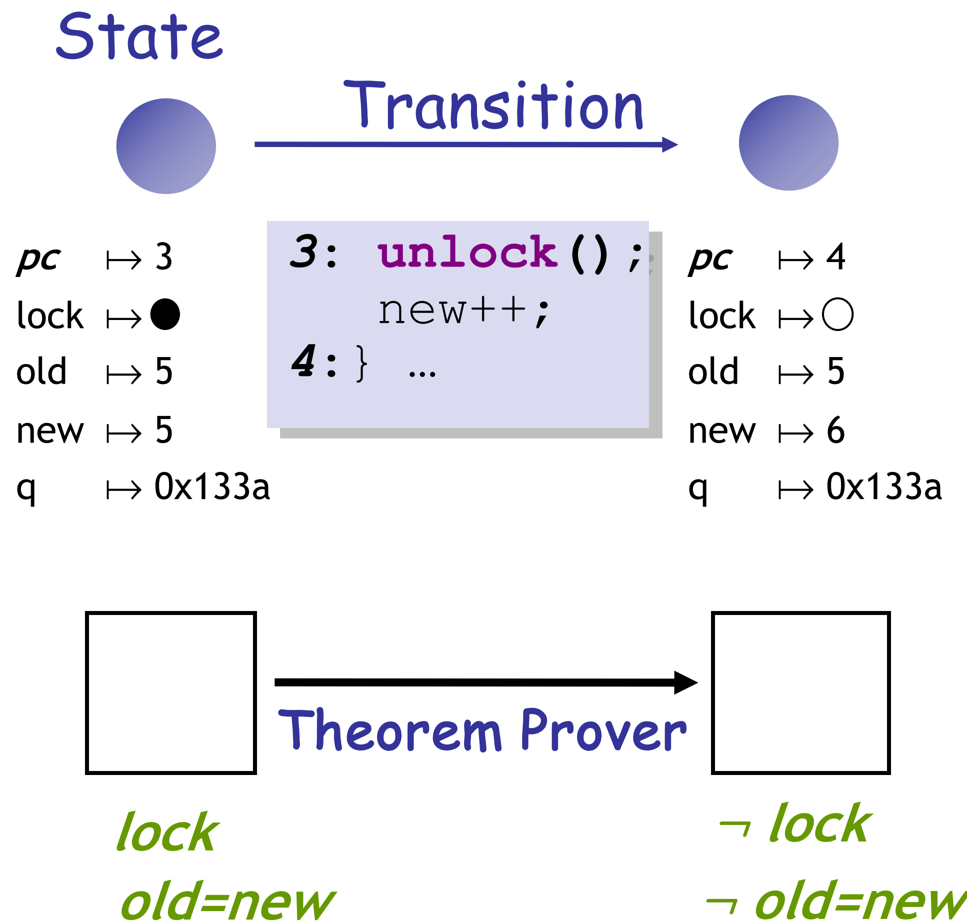


Existential Lifting

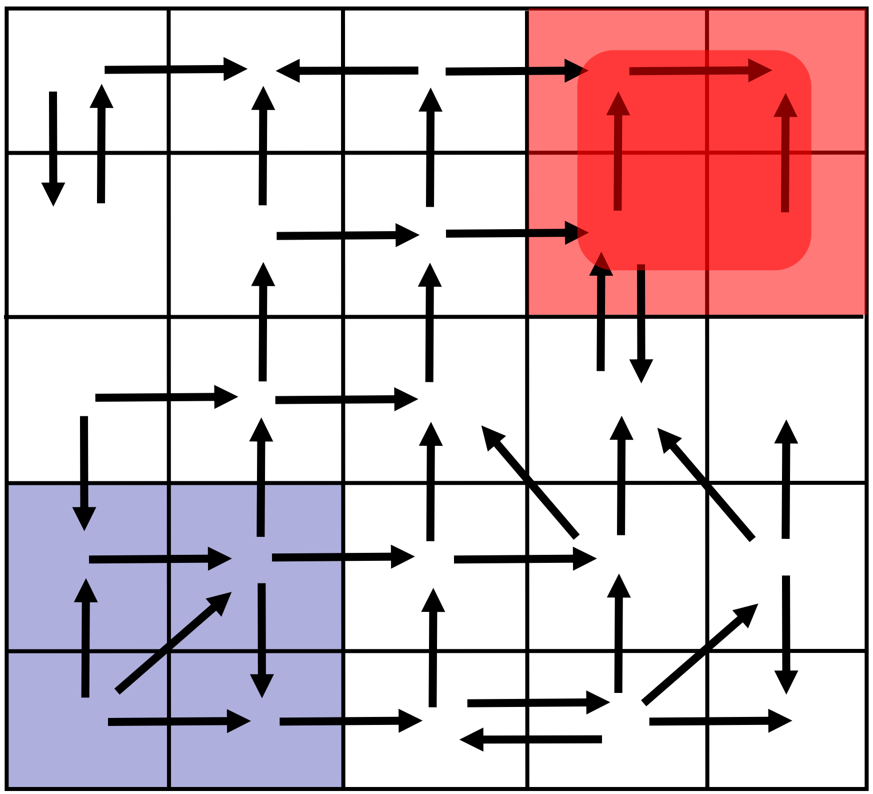
(i.e., $A_1 \rightarrow A_2$ iff $\exists c_1 \in A_1.\ \exists c_2 \in A_2.\ c_1 \rightarrow c_2$)

Abstraction





Analyze Abstraction



Analyze finite graph

Over Approximate

Safe ⇒ System Safe

No false negatives

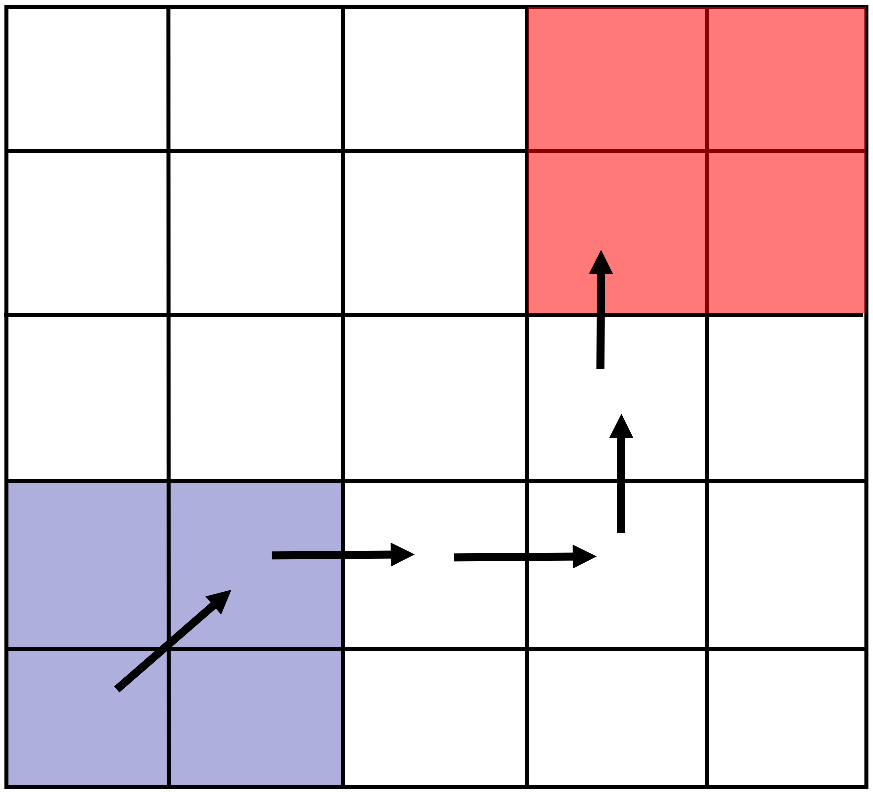
Problem

Spurious

co … s

false positives

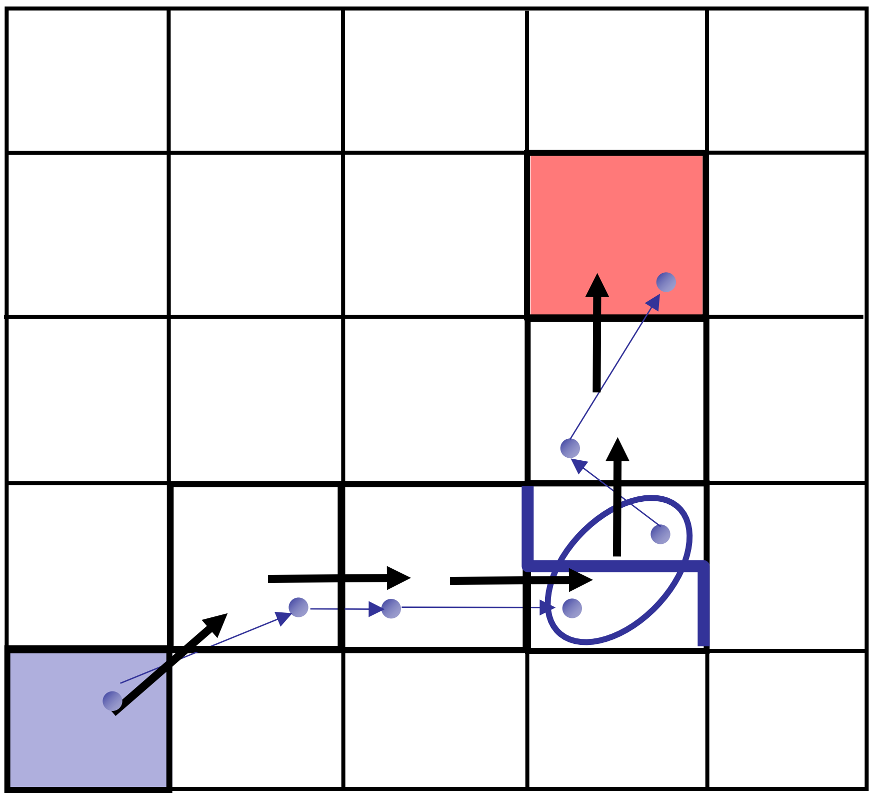
Idea 2: Counterexample-Guided Refinement



Solution

Use spurious **counterexamples** to **refine** abstraction!

Idea 2: Counterexample-Guided Refinement



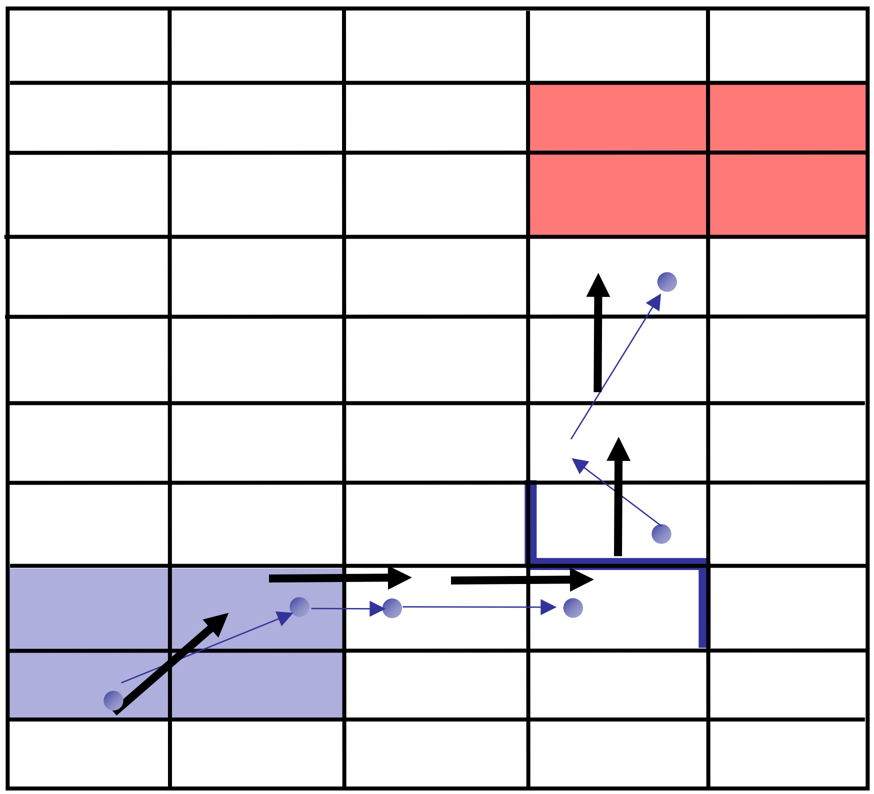
Solution

Use spurious **counterexamples** to **refine** abstraction!

1. **Add predicates** to distinguish states across **cut**
2. Build **refined** abstraction

Imprecision due to **merge**

Iterative Abstraction-Refinement



[Kurshan et al 93] [Clarke et al 00]
[Ball-Rajamani 01]

Solution

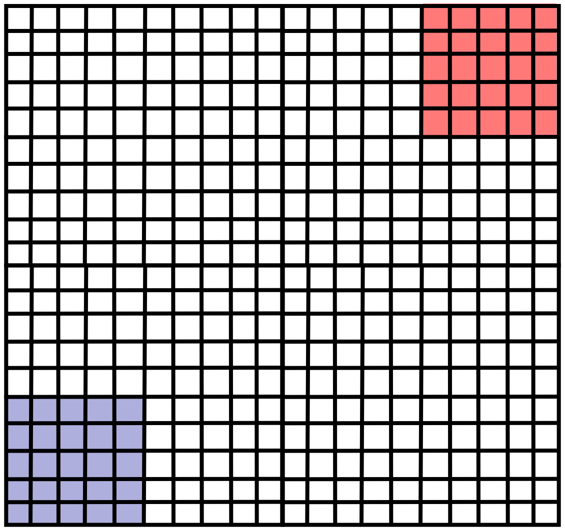
Use spurious **counterexamples** to **refine** abstraction!

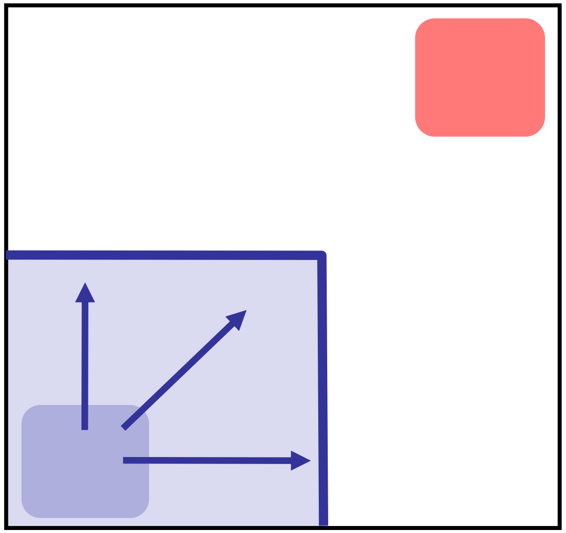
1. **Add predicates** to distinguish states across **cut**
2. Build **refined** abstraction
   - eliminates counterexample
3. **Repeat** search

   until real counterexample or system proved safe

Problem: Abstraction is Expensive

Why?





Reachable

Problem

#abstract states = $2^{\#predicates}$

Exponential Thm. Prover queries

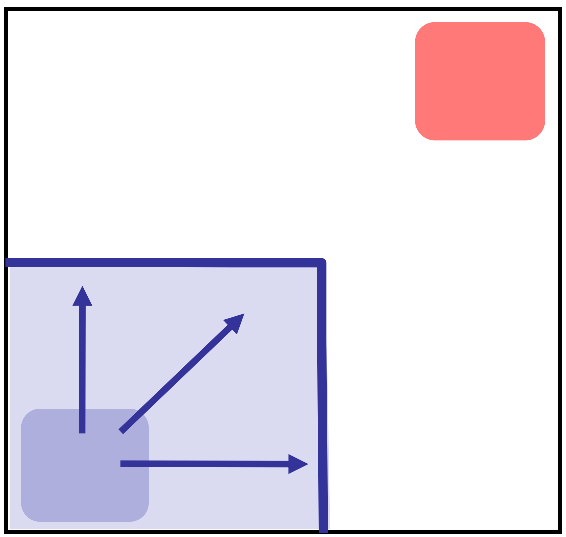
Observe

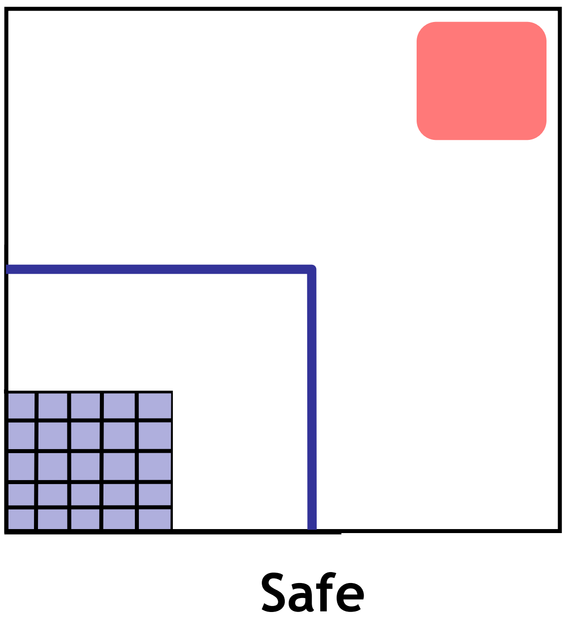
Fraction of state space reachable

#Preds ~ 100's, #States ~ 2^{100} ,

#Reach ~ 1000's

Solution1: Only Abstract Reachable States





Problem

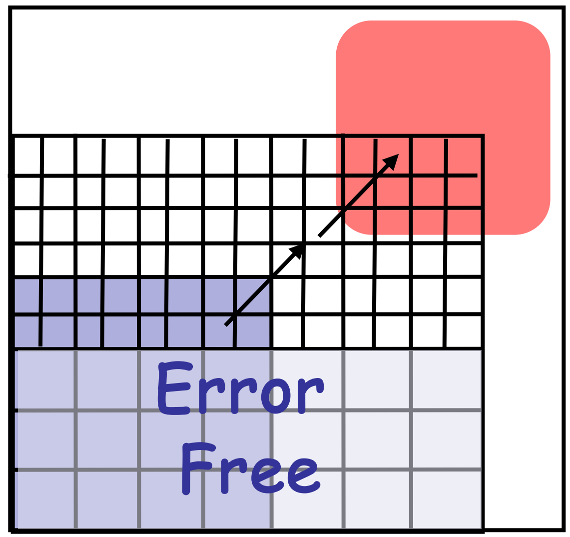
#abstract states = $2^{\#predicates}$

Exponential Thm. Prover queries

Solution

Build abstraction **during** search

Solution2: Don't Refine Error-Free Regions



Problem

#abstract states = $2^{\#predicates}$

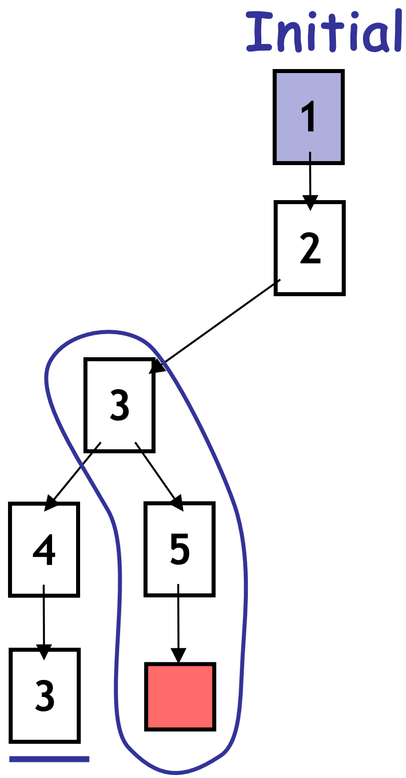
Exponential Thm. Prover queries

Solution

Don't refine error-free regions

Key Idea for Solutions?

Key Idea: Reachability Tree



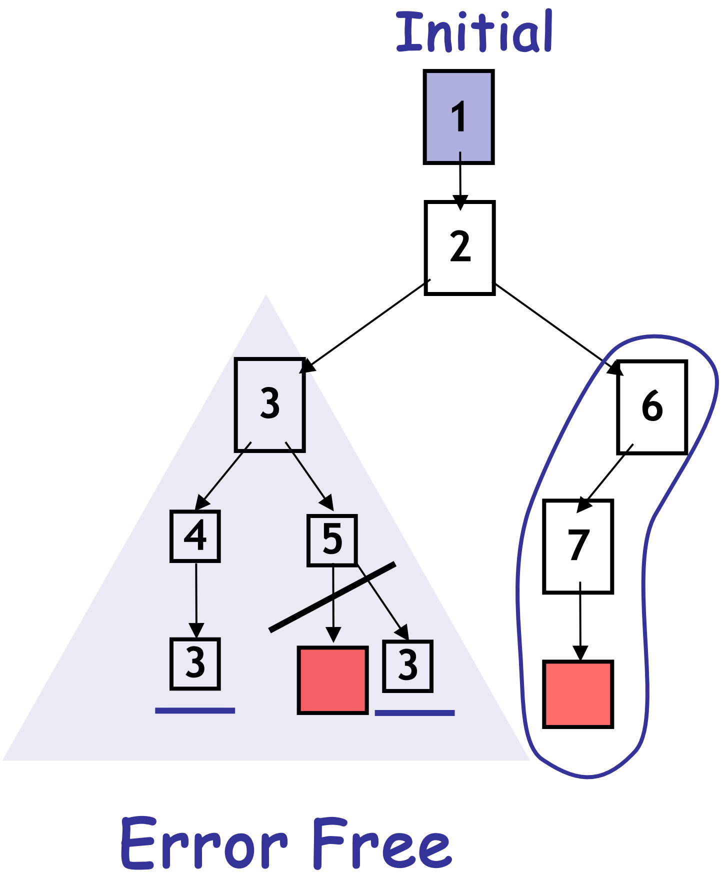
Unroll Abstraction

1. Pick tree-node (=abs. state)
2. Add children (=abs. successors)
3. On re-visiting abs. state, cut-off

Find min infeasible suffix

- Learn new predicates
- Rebuild subtree with new preds.

Key Idea: Reachability Tree



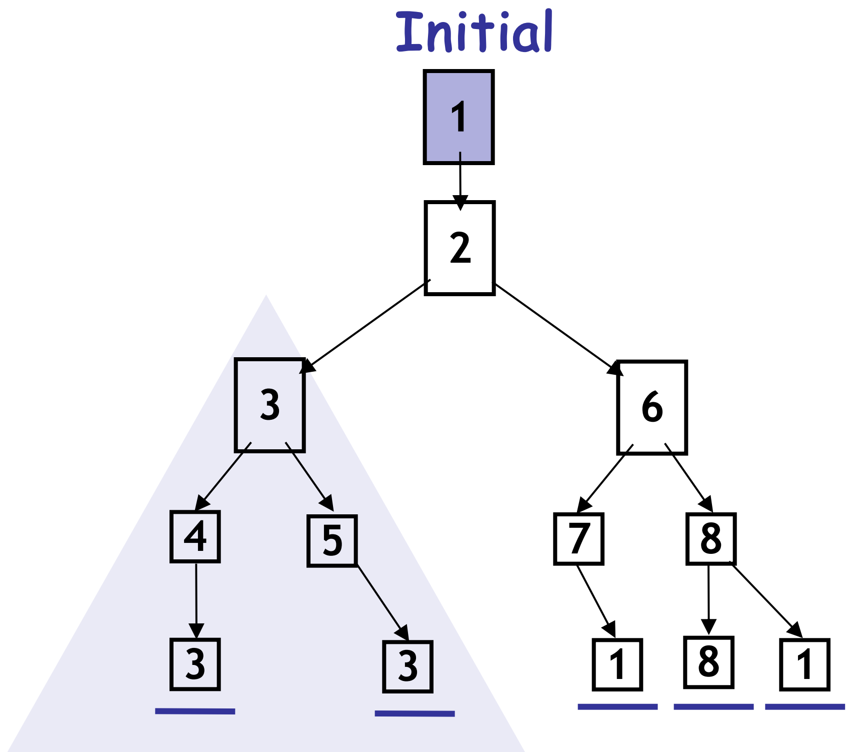
Unroll Abstraction

1. Pick tree-node (=abs. state)
2. Add children (=abs. successors)
3. On re-visiting abs. state, cut-off

Find min infeasible suffix

- Learn new predicates
- Rebuild subtree with new preds.

Key Idea: Reachability Tree



Unroll Abstraction

1. Pick tree-node (=abs. state)
2. Add children (=abs. successors)
3. On re-visiting abs. state, cut-off

Find min infeasible suffix

- Learn new predicates
- Rebuild subtree with new preds.

Error Free

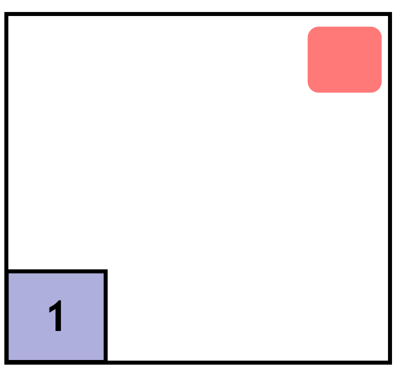
SAF
F

S1: Only Abstract Reachable States

S2: Don't refine error-free regions

Build-and-Search

```
Example ( ) {
1: do{
     lock();
     old = new;
     q = q->next;
2:   if (q != NULL){
3:      q->data = new;
        unlock();
        new ++;
     }
4:}while(new != old);
5:unlock();
}
```

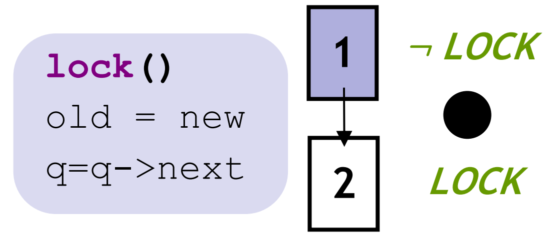


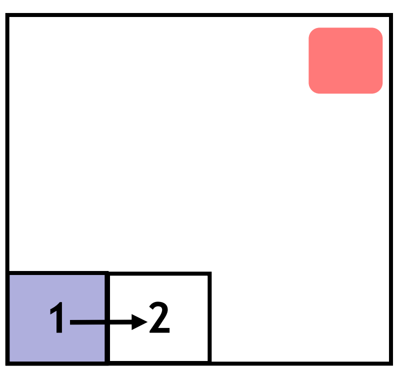
Predicates: *LOCK*

Reachability Tree

Build-and-Search

```
Example ( ) {
1: do{
      lock();
      old = new;
      q = q->next;
2:    if (q != NULL){
3:       q->data = new;
         unlock();
         new ++;
      }
4:}while(new != old);
5:unlock();
}
```



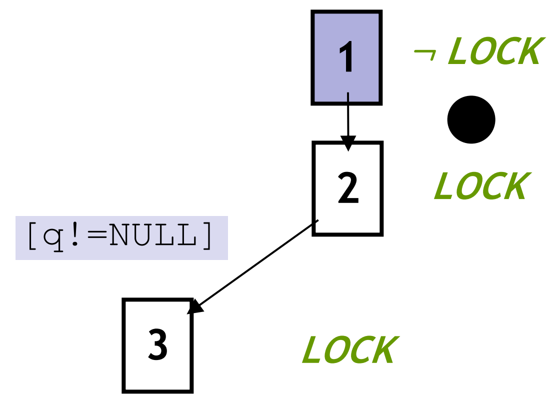


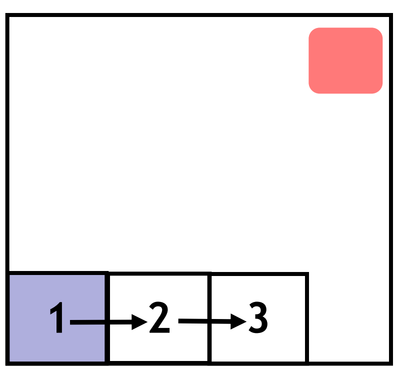
Predicates: *LOCK*

Reachability Tree

Build-and-Search

```
Example ( ) {
1: do{
      lock();
      old = new;
      q = q->next;
2:    if (q != NULL){
3:       q->data = new;
         unlock();
         new ++;
      }
4:}while(new != old);
5:unlock();
}
```



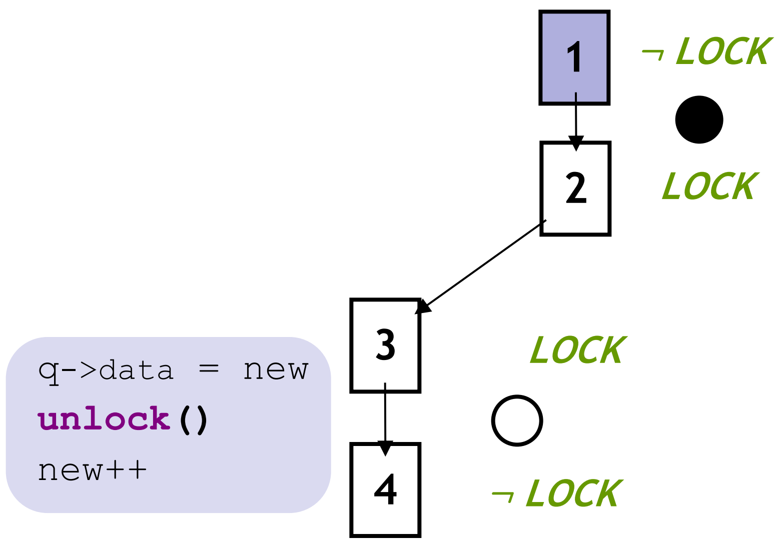


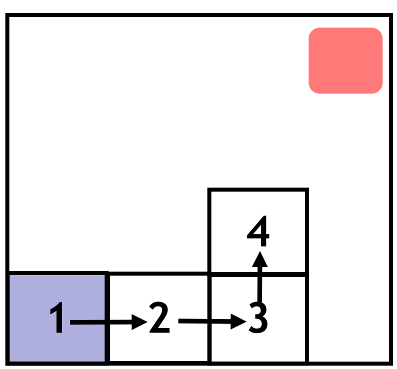
Predicates: *LOCK*

Reachability Tree

Build-and-Search

```
Example ( ) {
1: do{
      lock();
      old = new;
      q = q->next;
2:    if (q != NULL){
3:       q->data = new;
         unlock();
         new ++;
      }
4:}while(new != old);
5:unlock ();
}
```



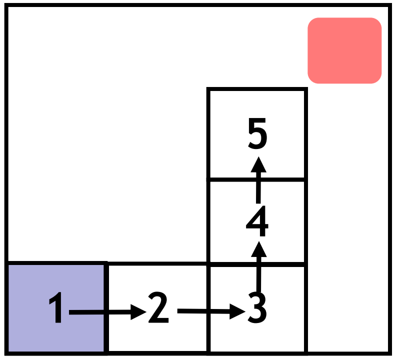


Predicates: LOCK

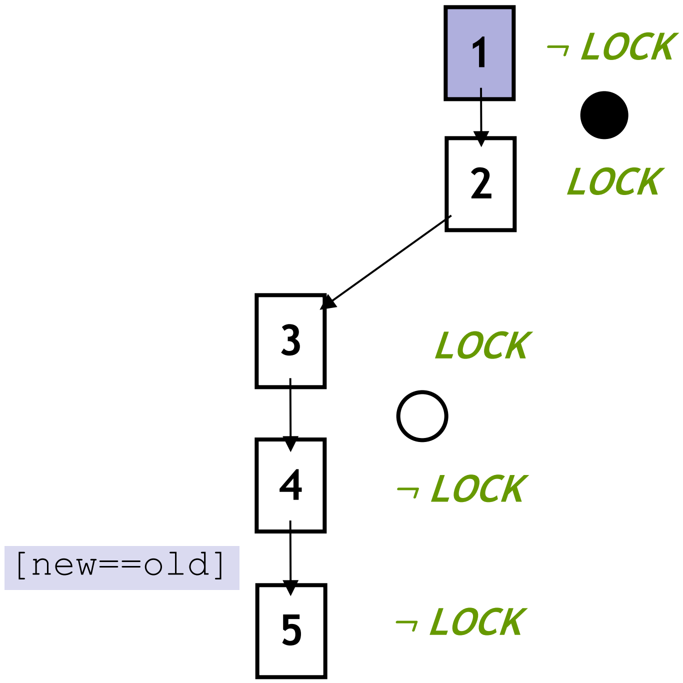
Reachability Tree

Build-and-Search

```
Example ( ) {
1: do{
     lock();
     old = new;
     q = q->next;
2:   if (q != NULL){
3:      q->data = new;
        unlock();
        new ++;
     }
4:}while(new != old);
5:unlock();
}
```



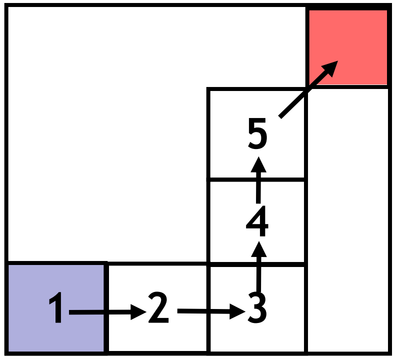
Predicates: *LOCK*



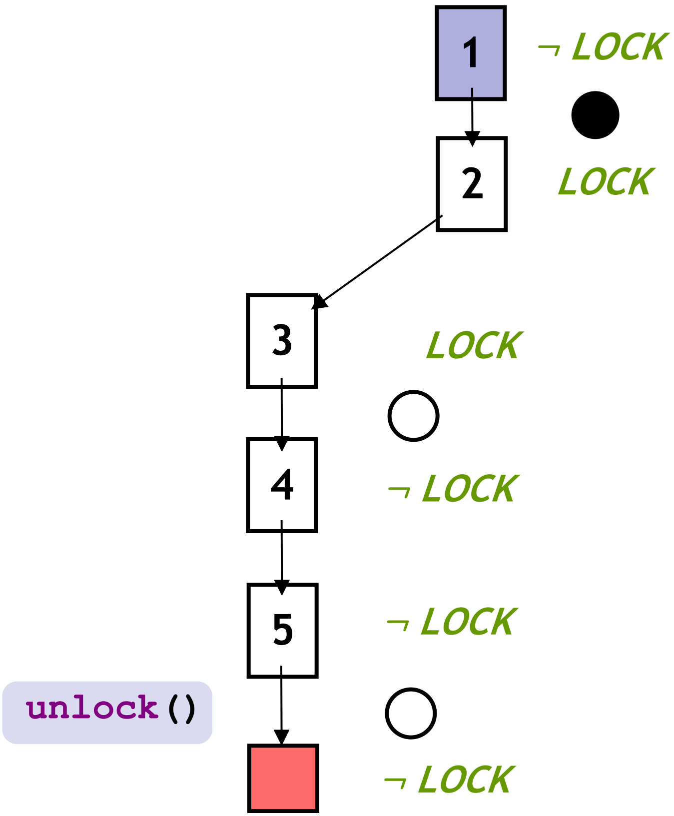
Reachability Tree

Build-and-Search

```
Example ( ) {
1: do{
      lock();
      old = new;
      q = q->next;
2:    if (q != NULL){
3:       q->data = new;
         unlock();
         new ++;
      }
4:}while(new != old);
5:unlock();
}
```



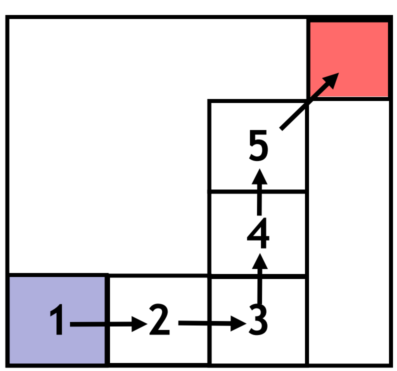
Predicates: *LOCK*



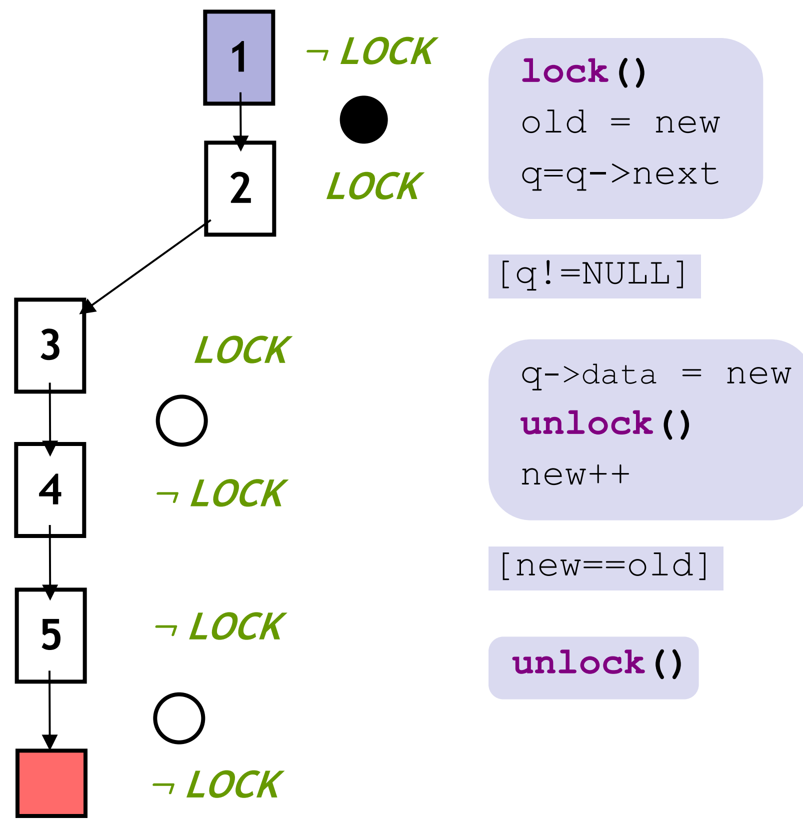
Reachability Tree

Analyze Counterexample

```
Example ( ) {
1:  do{
      lock();
      old = new;
      q = q->next;
2:    if (q != NULL){
3:       q->data = new;
         unlock();
         new ++;
      }
4:}while(new != old);
5:unlock();
}
```



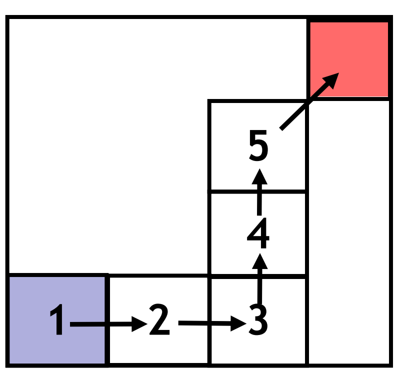
Predicates: *LOCK*



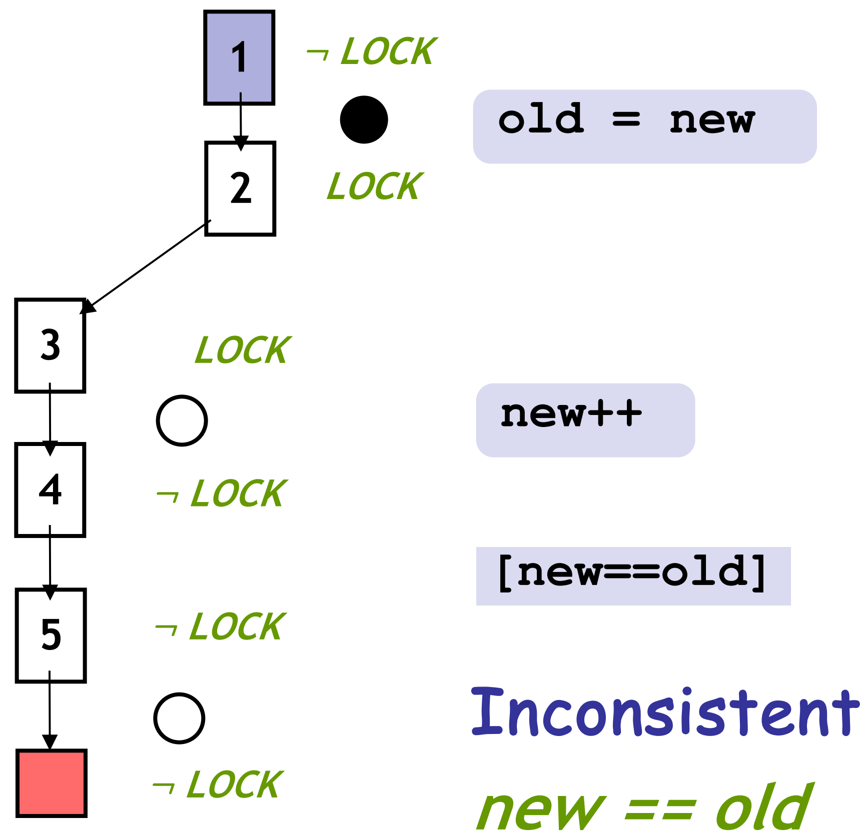
Reachability Tree

Analyze Counterexample

```
Example ( ) {
1: do{
       lock();
       old = new;
       q = q->next;
2:     if (q != NULL){
3:        q->data = new;
          unlock();
          new ++;
       }
4:}while(new != old);
5:unlock();
}
```



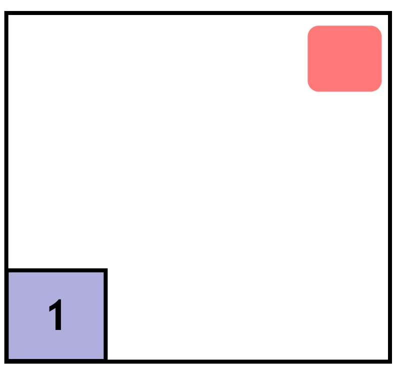
Predicates: *LOCK*



Reachability Tree

Repeat Build-and-Search

```
Example ( ) {
1: do{
      lock();
      old = new;
      q = q->next;
2:    if (q != NULL){
3:       q->data = new;
         unlock();
         new ++;
      }
4:}while(new != old);
5:unlock();
}
```

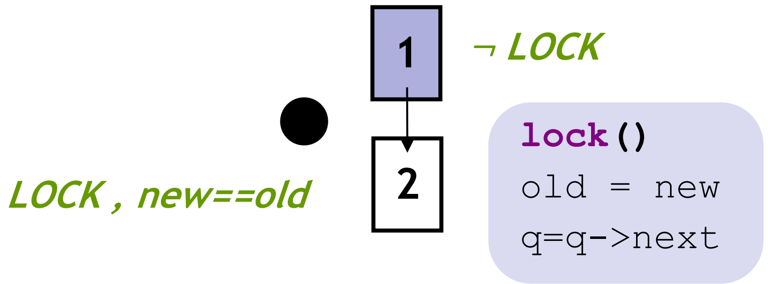


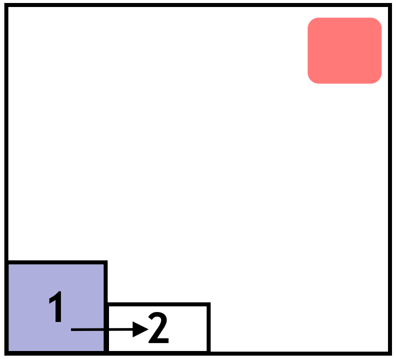
Predicates: *LOCK, new == old*

Reachability Tree

Repeat Build-and-Search

```
Example ( ) {
1: do{
      lock();
      old = new;
      q = q->next;
2:    if (q != NULL){
3:       q->data = new;
         unlock();
         new ++;
      }
4:}while(new != old);
5:unlock();
}
```



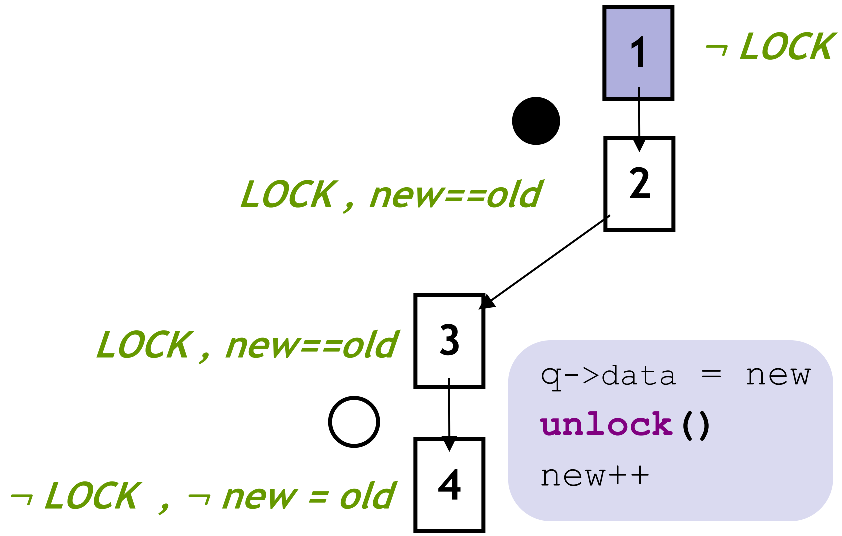


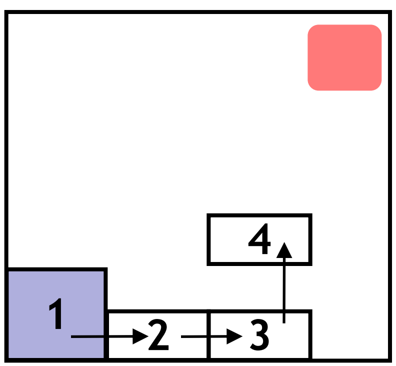
Predicates: LOCK, new == old

Reachability Tree

Repeat Build-and-Search

```
Example ( ) {
1: do{
      lock();
      old = new;
      q = q->next;
2:    if (q != NULL){
3:       q->data = new;
         unlock();
         new ++;
      }
4:}while(new != old);
5:unlock ();
}
```



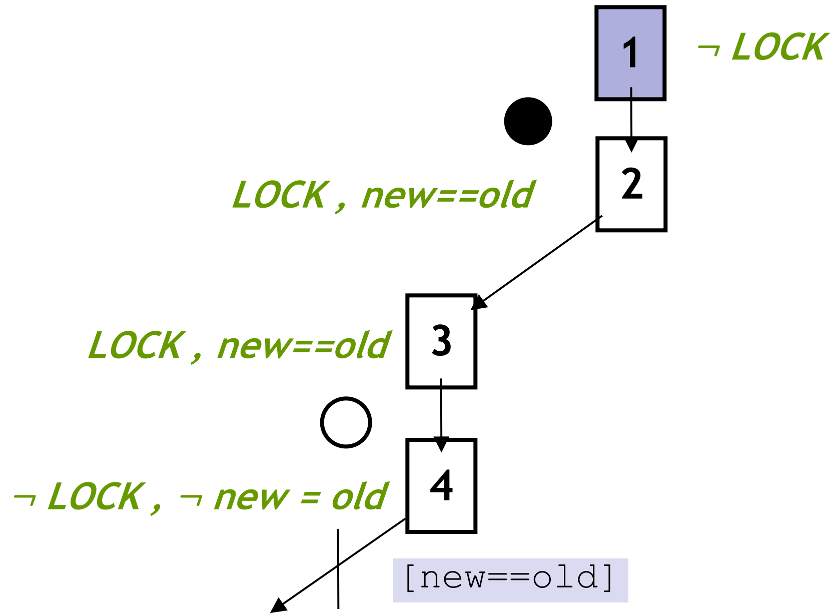


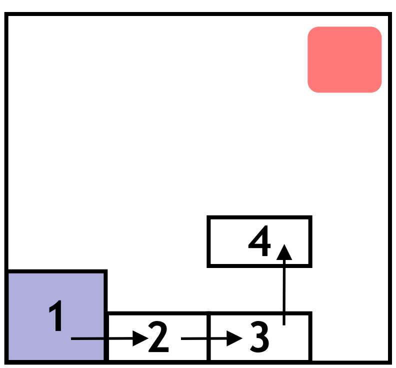
Predicates: LOCK, new == old

Reachability Tree

Repeat Build-and-Search

```
Example ( ) {
1: do{
     lock();
     old = new;
     q = q->next;
2:   if (q != NULL){
3:     q->data = new;
       unlock();
       new ++;
     }
4:}while(new != old);
5:unlock();
}
```



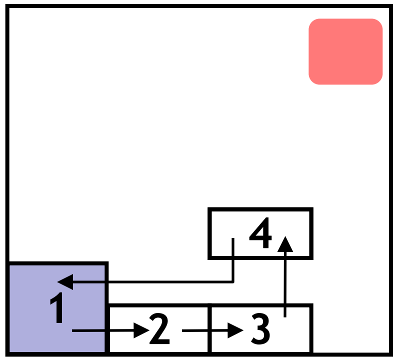


Predicates: LOCK, new == old

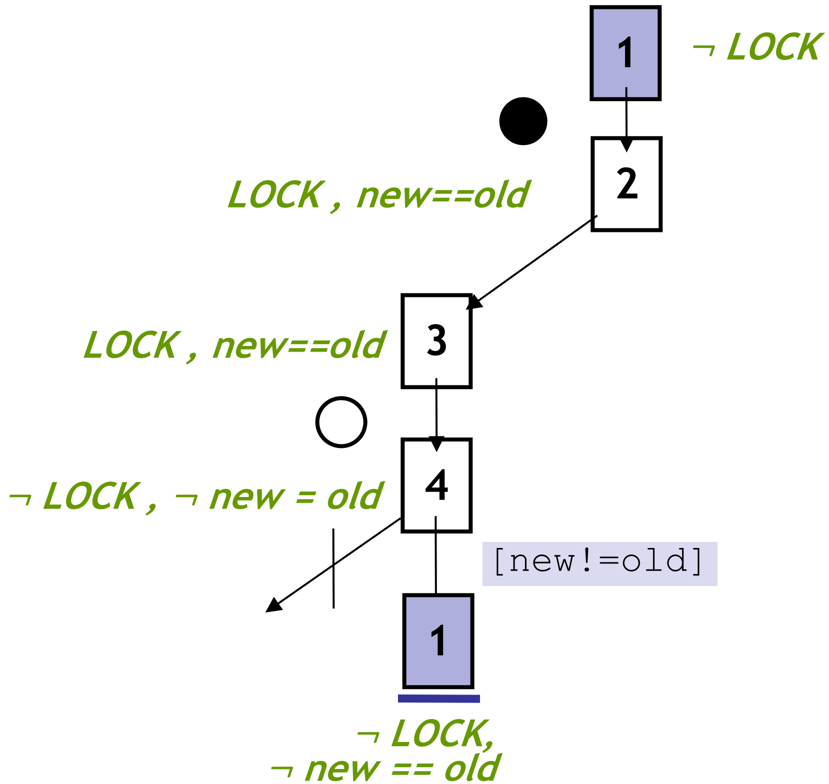
Reachability Tree

Repeat Build-and-Search

```
Example ( ) {
1:  do{
      lock();
      old = new;
      q = q->next;
2:    if (q != NULL){
3:      q->data = new;
        unlock();
        new ++;
      }
4:}while(new != old);
5:unlock();
}
```



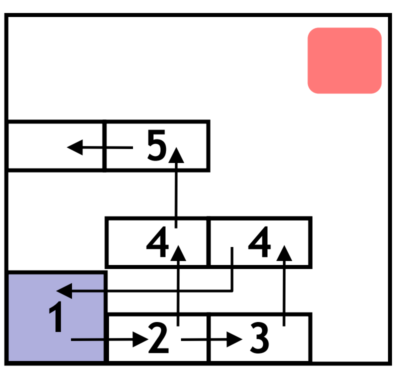
Predicates: *LOCK, new == old*



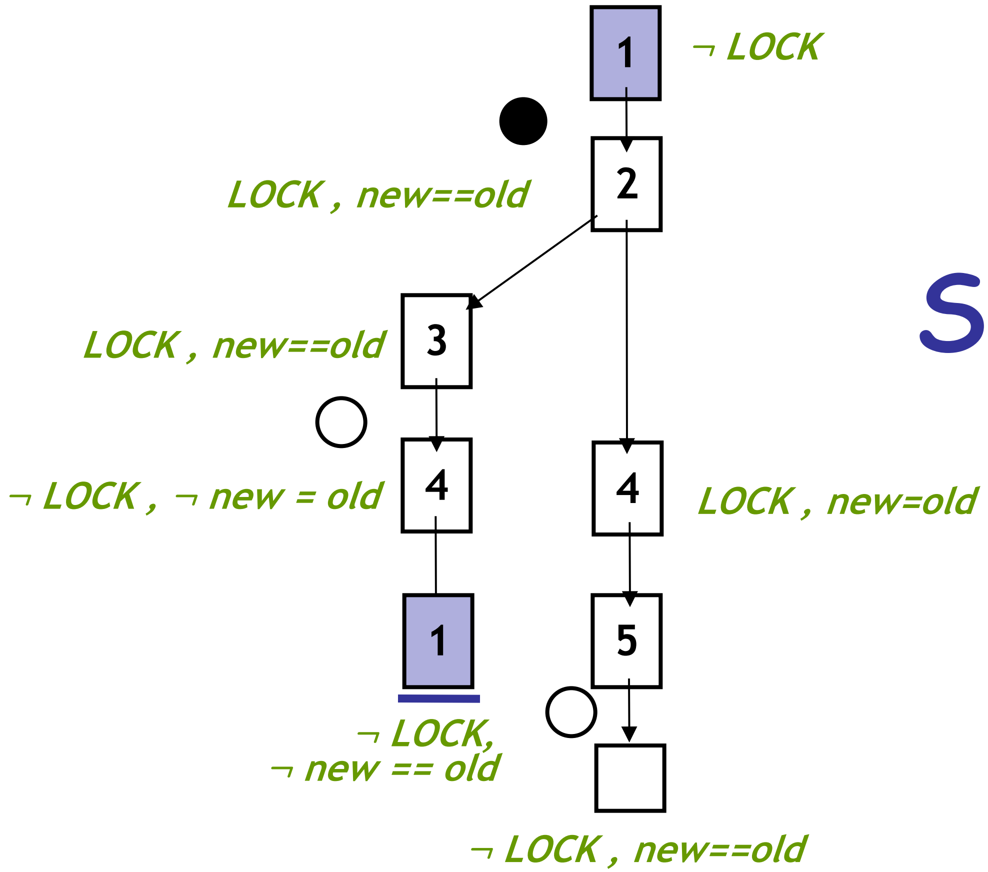
Reachability Tree

Repeat Build-and-Search

```
Example ( ) {
1: do{
     lock();
     old = new;
     q = q->next;
2:   if (q != NULL){
3:      q->data = new;
        unlock();
        new ++;
     }
4:}while(new != old);
5:unlock();
}
```



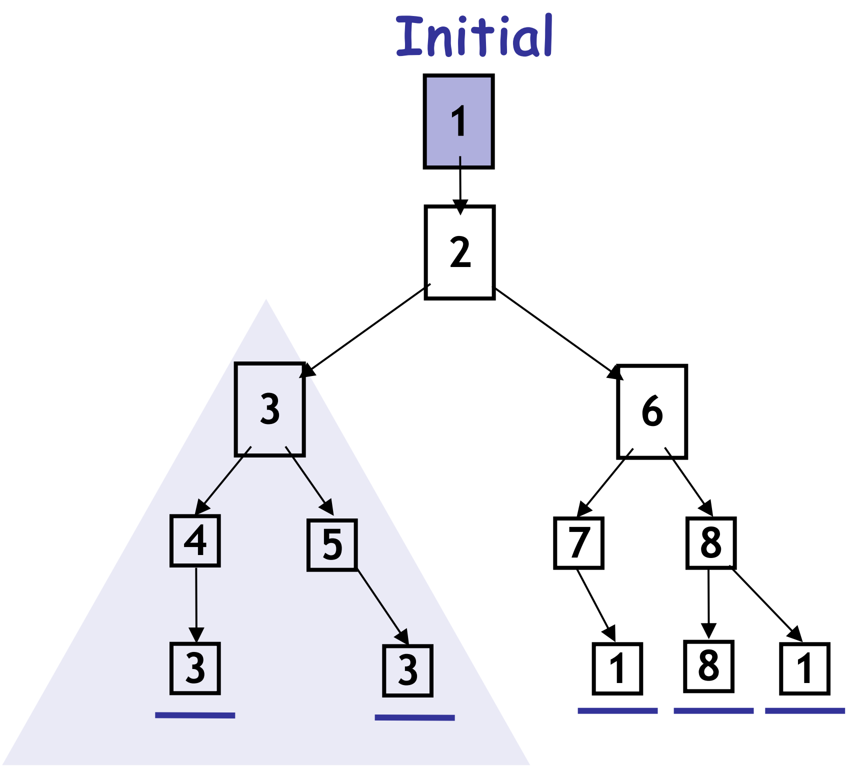
Predicates: *LOCK, new == old*



SAFE

Reachability Tree

Key Idea: Reachability Tree



Unroll Abstraction

1. Pick tree-node (=abs. state)
2. Add children (=abs. successors)
3. On re-visiting abs. state, cut-off

Find min infeasible suffix

- Learn new predicates
- Rebuild subtree with new preds.

Error Free

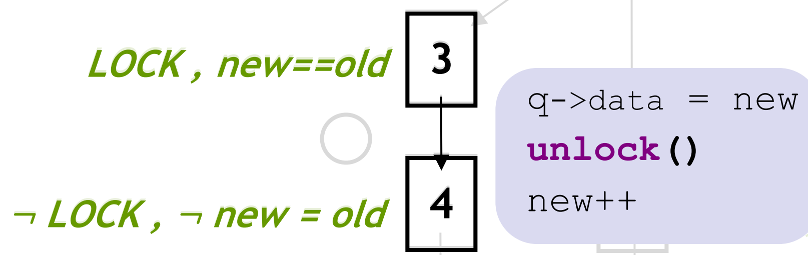
SAFE

S1: Only Abstract Reachable States

S2: Don't refine error-free regions

Two Handwaves

Q. How to compute "successors" ?



Q. How to find predicates ?

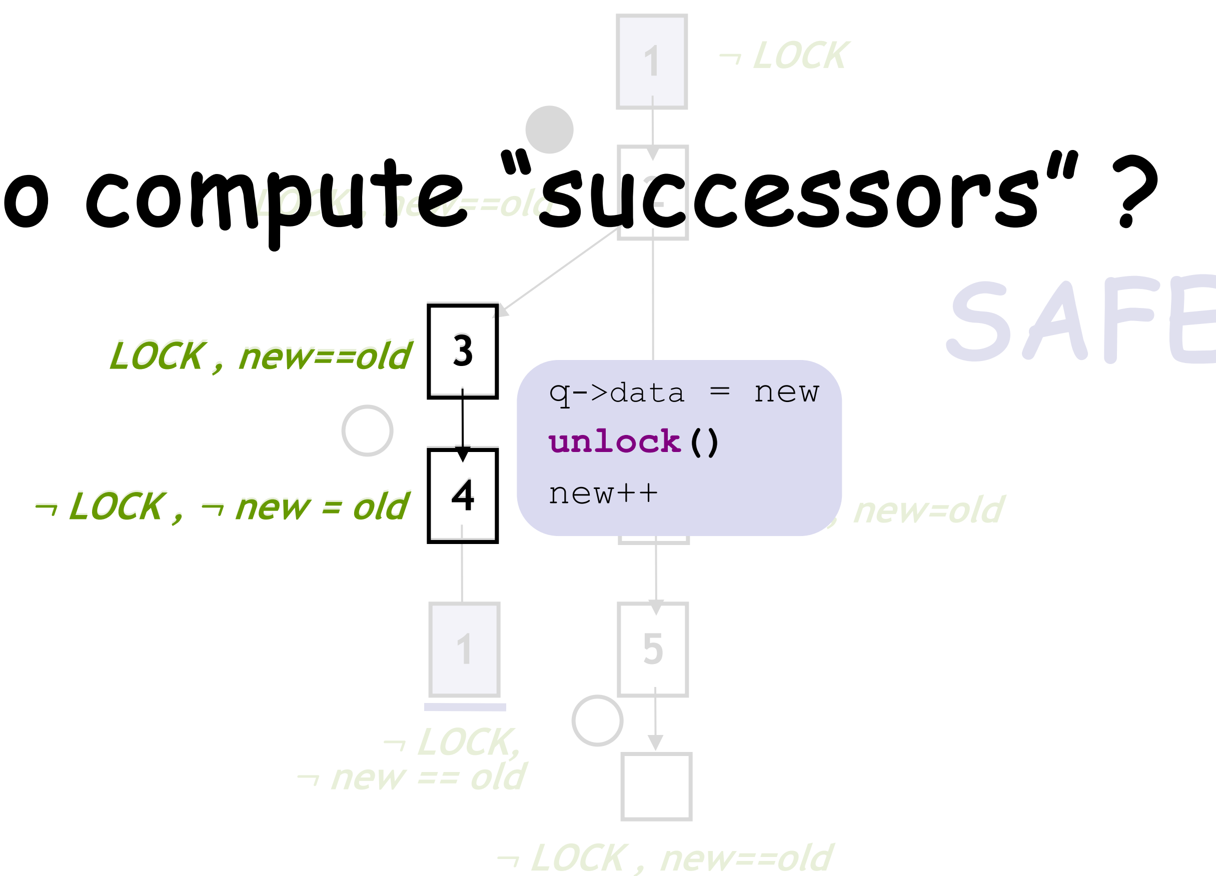
Refinement

Predicates: *LOCK, new==old*

Two Handwaves

```
Example ( ) {
1: do{
      lock();
      old = new;
      q = q->next;
2:    if (q != NULL){
3:       q->data = new;
         unlock();
         new ++;
      }
4:}while(new != old);
5:unlock();
}
```

Q. How to compute "successors" ?



Reachability Tree

Predicates: LOCK, new == old

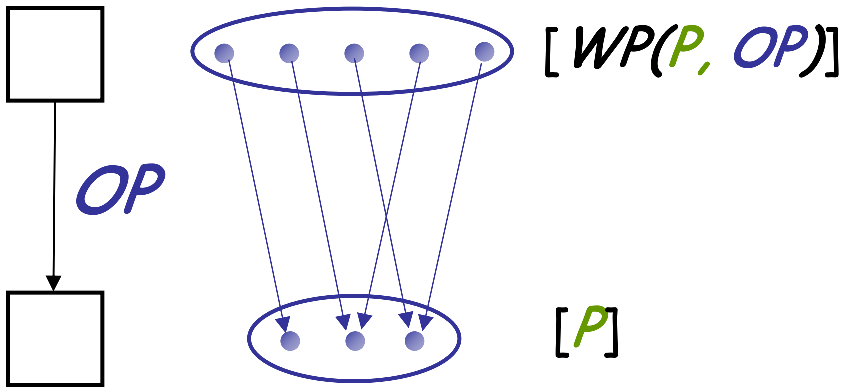
Weakest Preconditions

WP(P,OP)

Weakest formula *P'* s.t.

if *P'* is true <u>before</u> *OP*

then *P* is true <u>after</u> *OP*



Weakest Preconditions

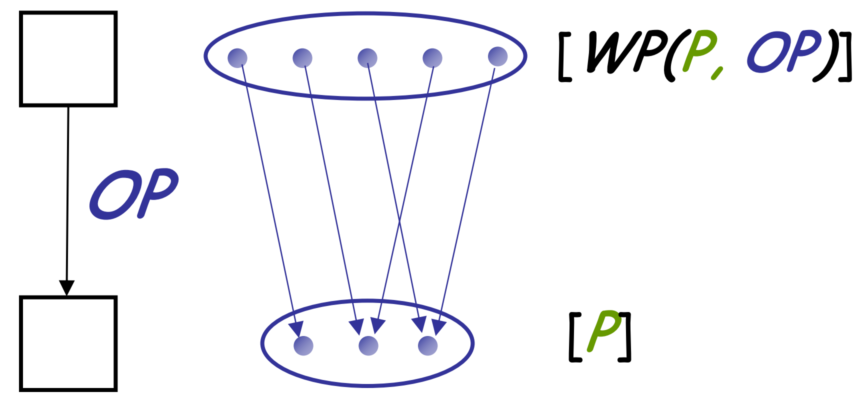
More on this later in the semester!

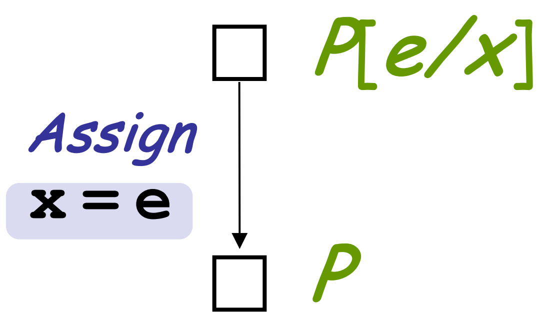
WP(P,OP)

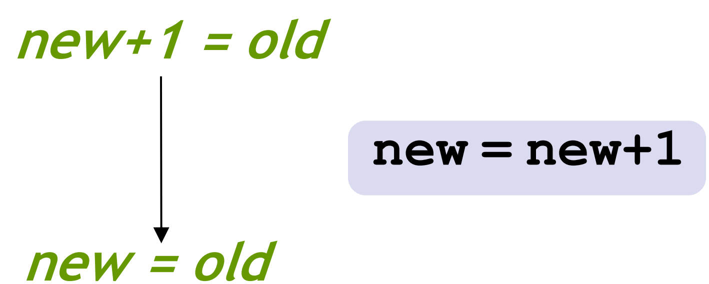
Weakest formula *P'* s.t.

if *P'* is true before *OP*

then *P* is true after *OP*

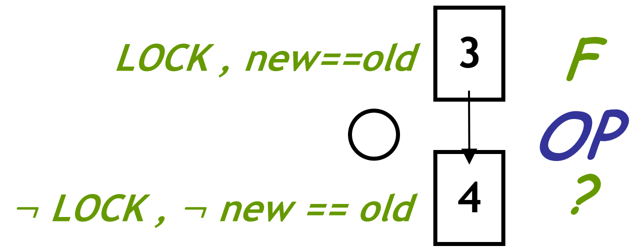






How to compute successor?

```
Example ( ) {
1: do{
     lock();
     old = new;
     q = q->next;
2:   if (q != NULL){
3:     q->data = new;
       unlock();
       new ++;
     }
4:}while(new != old);
5:unlock();
}
```



For each *p*

- Check if *p* is true (or false) after *OP*

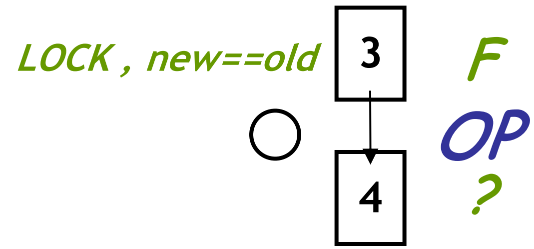
Q: When is *p* true <u>after</u> *OP* ?

- If *WP(p, OP)* is true <u>before</u> *OP* !
- We know *F* is true <u>before</u> *OP*
- Thm. Pvr. Query: $F \Rightarrow WP(p, OP)$

Predicates: *LOCK, new == old*

How to compute successor?

```
Example ( ) {
1: do{
      lock();
      old = new;
      q = q->next;
2:    if (q != NULL){
3:       q->data = new;
         unlock();
         new ++;
      }
4:}while(new != old);
5:unlock();
}
```



For each *p*

- Check if *p* is true (or false) after *OP*

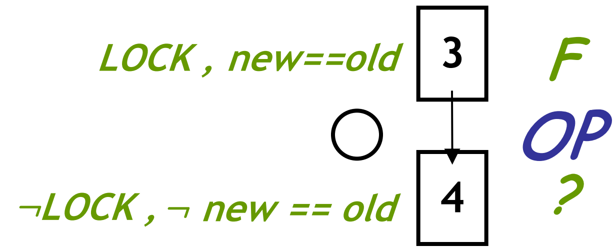
Q: When is *p* false after *OP* ?

- If *WP(¬p, OP)* is true before *OP* !
- We know *F* is true before *OP*
- Thm. Pvr. Query: $F \Rightarrow WP(\neg p, OP)$

Predicates: *LOCK, new == old*

How to compute successor?

```
Example ( ) {
1: do{
     lock();
     old = new;
     q = q->next;
2:   if (q != NULL){
3:     q->data = new;
       unlock();
       new ++;
     }
4:}while(new != old);
5:unlock();
}
```



For each *p*

- Check if *p* is true (or false) after *OP*

Q: When is *p* false after *OP* ?

- If *WP(¬p, OP)* is true before *OP* !
- We know *F* is true before *OP*
- Thm. Pvr. Query: $F \Rightarrow WP(\neg p, OP)$

Predicate: *new == old*

True? *(LOCK , new==old)* $\Rightarrow$ *(new + 1 = old)* **NO**

False? *(LOCK , new==old)* $\Rightarrow$ *(new + 1 ≠ old)* **YES**

Advanced SLAM/BLAST

Too Many Predicates
- Use Predicates Locally

Counter-Examples
- Craig Interpolants

Procedures
- Summaries

Concurrency
- Thread-Context Reasoning

SLAM Summary

1) Instrument Program With Safety Policy
2) Predicates = { }
3) Abstract Program With Predicates
   - Use Weakest Preconditions and Theorem Prover Calls
4) Model-Check Resulting Boolean Program
   - Use Symbolic Model Checking
5) Error State Not Reachable?
   - Original Program Has No Errors: Done!
6) Check Counterexample Feasibility
   - Use Symbolic Execution
7) Counterexample Is Feasible?
   - Real Bug: Done!
8) Counterexample Is Not Feasible?
   1) Find New Predicates (Refine Abstraction)
   2) Goto Line 3

Bonus: SLAM/BLAST Weakness

```
1: F() {
2:   int x=0;
3:   lock();
4:   x++;
5:   while (x ≠ 88);
6:   if (x < 77)
7:     lock();
8: }
```

- Preds = {}, Path = 234567
- $[x=0, \neg x+1 \neq 88, x+1 < 77]$
- Preds = {x=0}, Path = 234567
- $[x=0, \neg x+1 \neq 88, x+1 < 77]$
- Preds = {x=0, x+1=88}
- Path = 23454567
- $[x=0, \neg x+2 \neq 88, x+2 < 77]$
- Preds = {x=0,x+1=88,x+2=88}
- Path = 2345454567
- …
- Result: the predicates "count" the loop iterations