Software Model Checking and Counter-example Guided Abstraction Refinement

Claire Le Goues

Motivation: How should we analyze this?

```
2:  do {
          lock();
          old = new;
3:        if (*){
             unlock();
             new++;
          }
5:  } while (new != old);
6:  unlock();
    return;
```

- * means something we can't analyze (user input, random value)
- Line 5: the lock is held if and only if old = new

Motivation: How should we analyze this?

```
 Example() {
1:   if (*){
7:      do {
             got_lock = 0;
8:           if (*){
9:              lock();
                got_lock++;
             }
10:          if (got_lock){
11:             unlock();
             }
12:     } while (*)
    }
```

- * means something we can't analyze (user input, random value)
- Line 10: the lock is held if and only if got_lock = 1

Tradeoffs...

Dataflow analysis requires fixed abstractions, e.g., zero/non-zero, locked/unlocked

```
 Example() {
1:   if (*){
7:      do {
            got_lock = 0;
8:          if (*){
9:             lock();
               got_lock++;
            }
10:         if (got_lock){
11:            unlock();
            }
12:     } while (*)
    }
2:   do {
        lock();
        old = new;
3:      if (*){
4:         unlock();
           new++;
        }
5:   } while (new != old);
6:   unlock();
     return;
```

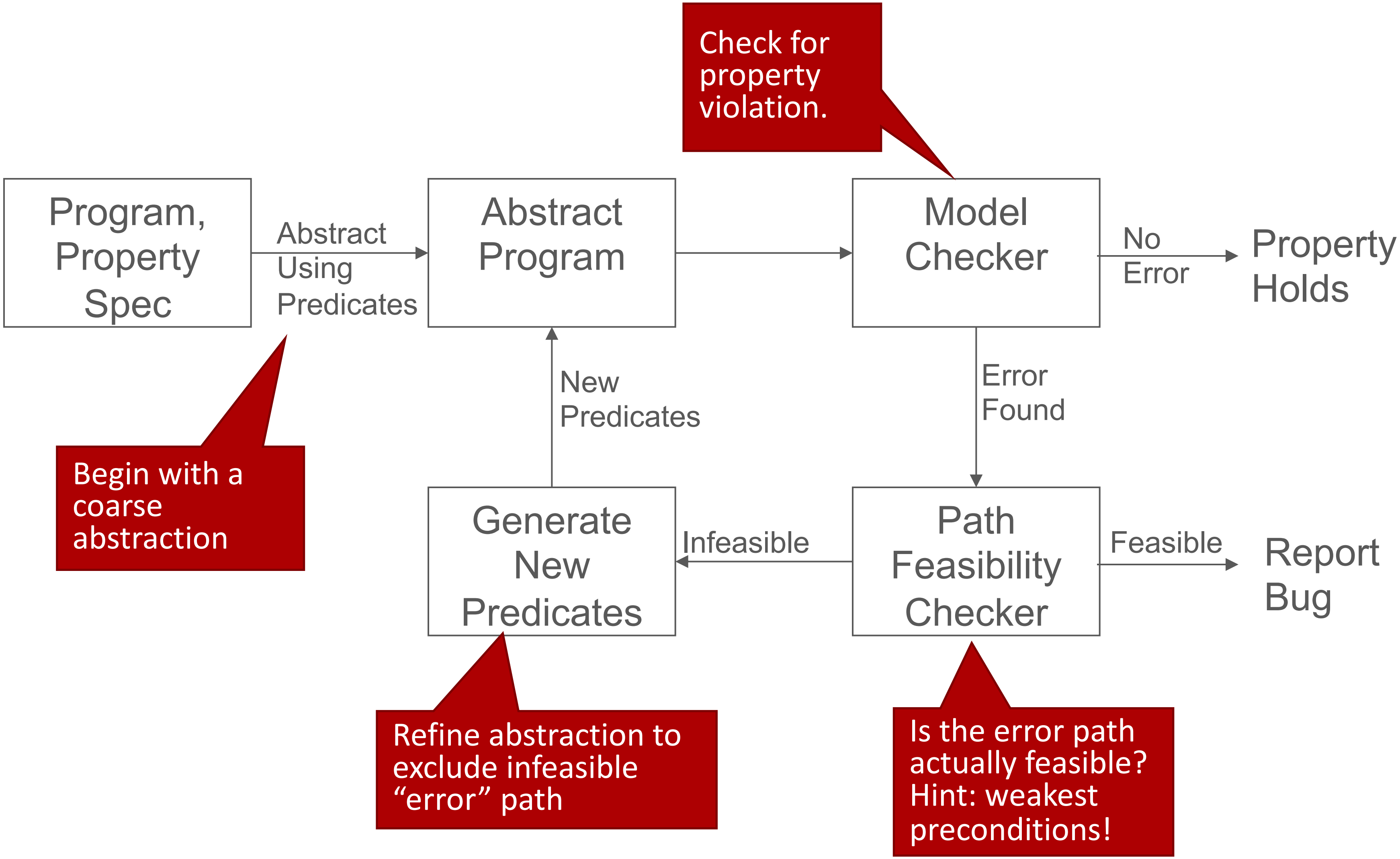
Symbolic execution shows need to eliminate infeasible paths, see lock/unlock on correlated branches (more complicated logic!).

Explicit-state Model Checking needs programs to be represented as a finite state model…state explosion??

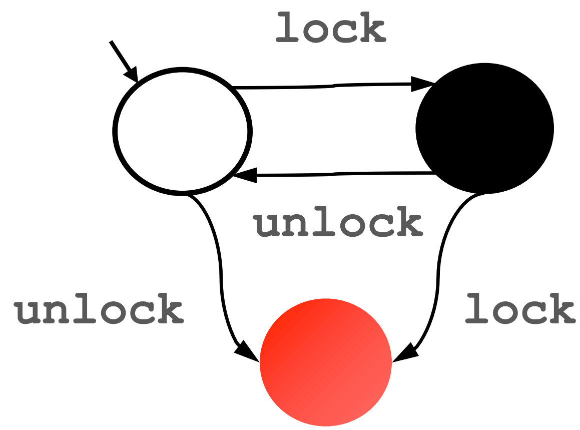
Enter: Abstraction Refinement

- Can we get both soundness and the precision to eliminate infeasible paths?
  - In general: of course not! That's undecidable.
  - But in many situations we can solve it with *abstraction refinement.*
- *…what will we lose?*
  - Answer: Termination guarantees. OH WELL.

CEGAR: Counterexample Guided Abstraction Refinement



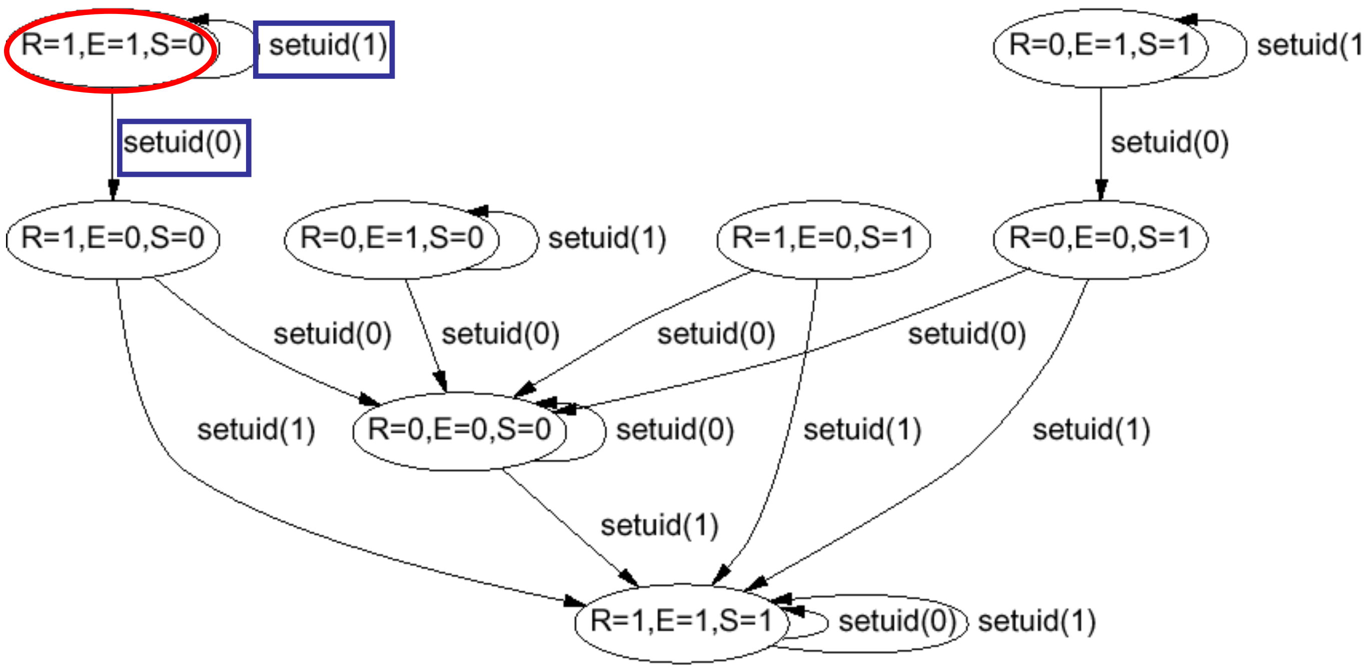
Property 1: Double Locking



"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

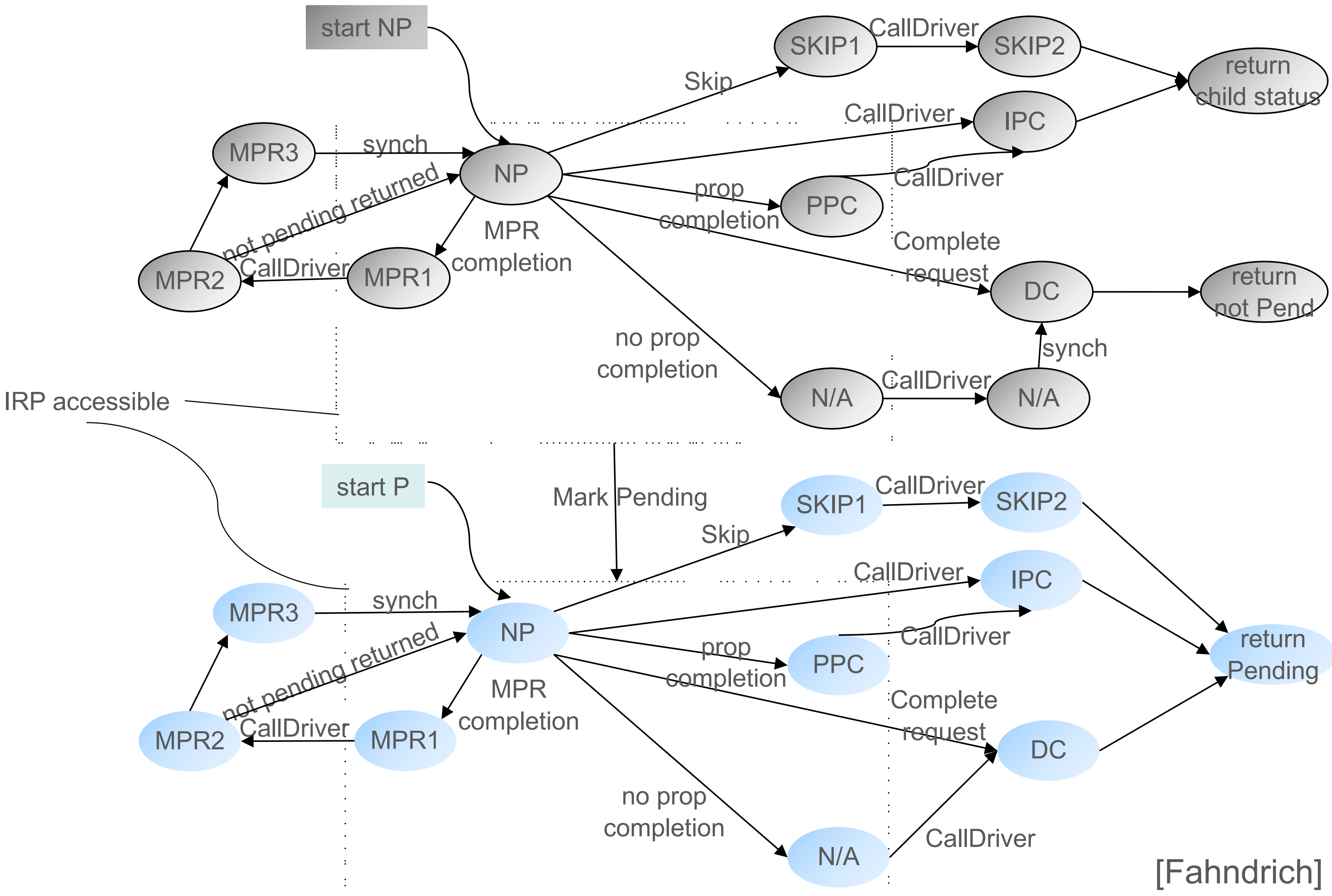


[Chen-Dean-Wagner '02]

"User applications must not run with root privilege"

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

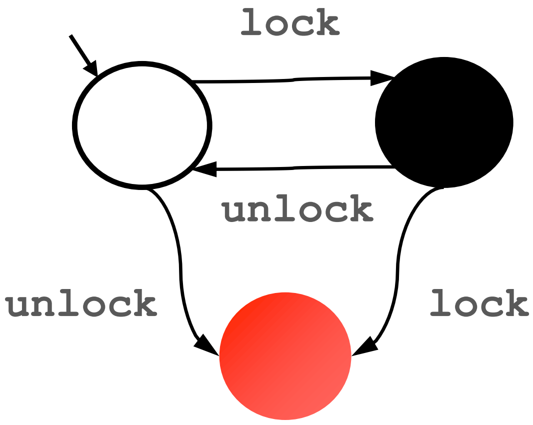
Property 3 : IRP Handler



[Fahndrich]

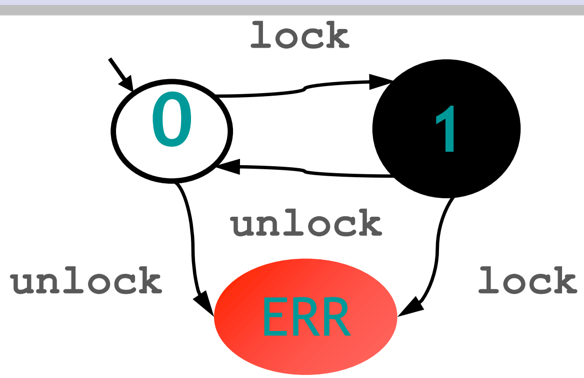
Example SLAM Input

```
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;
}
```



Incorporating Specs

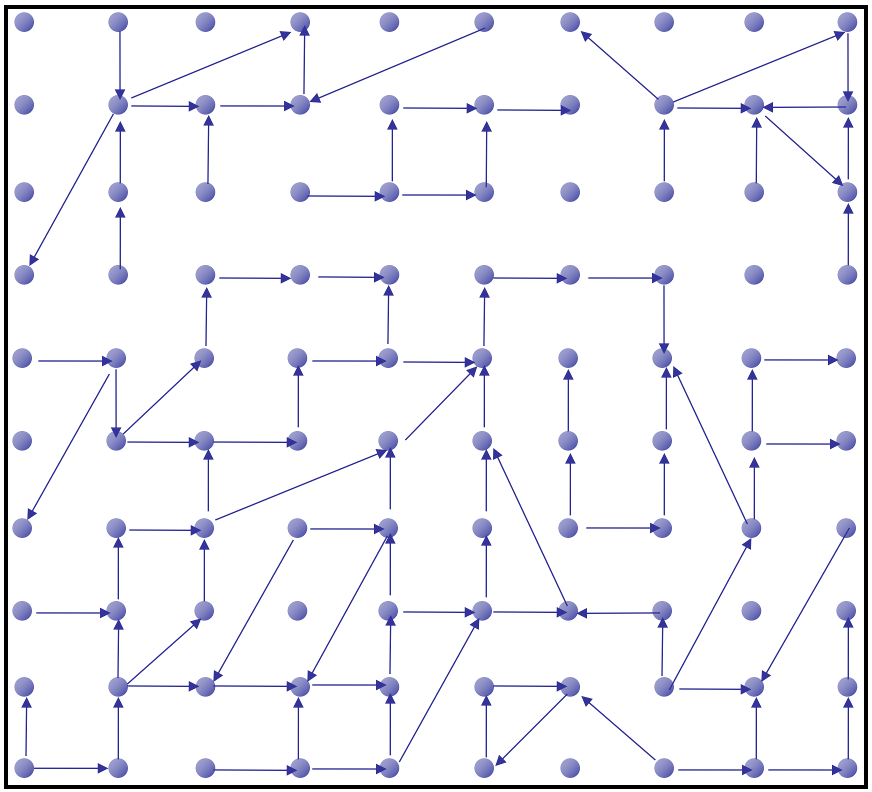
```
Example ( ) {
1: do{
       lock();
       old = new;
       q = q->next;
2:     if (q != NULL){
3:       q->data = new;
         unlock();
         new ++;
       }
4: } while(new != old);
5:  unlock ();
    return;
}
```

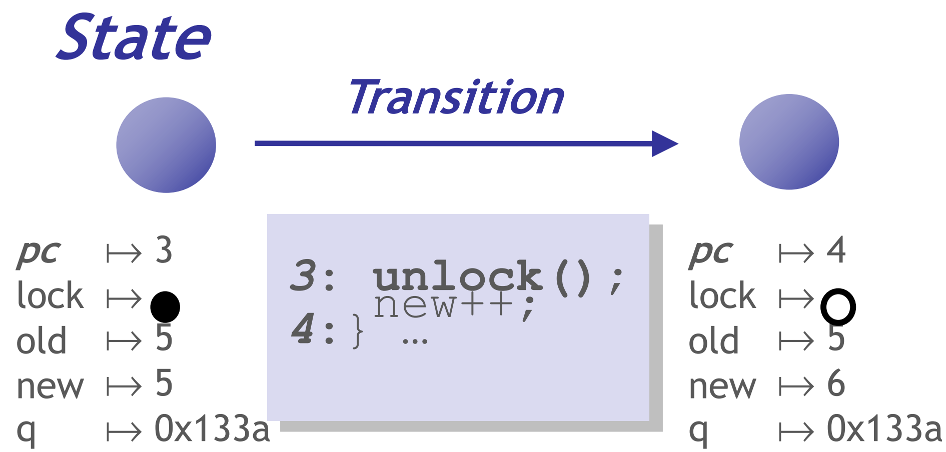



```
Example ( ) {
1: do{
       if L=1 goto ERR;
       else L=1;
       old = new;
       q = q->next;
2:     if (q != NULL){
3:          q->data = new;
            if L=0 goto ERR;
            else L=0;
            new ++;
       }
4: } while(new != old);
5:   if L=0 goto ERR;
     else L=0;
     return;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

Original program violates spec iff new program reaches ERR

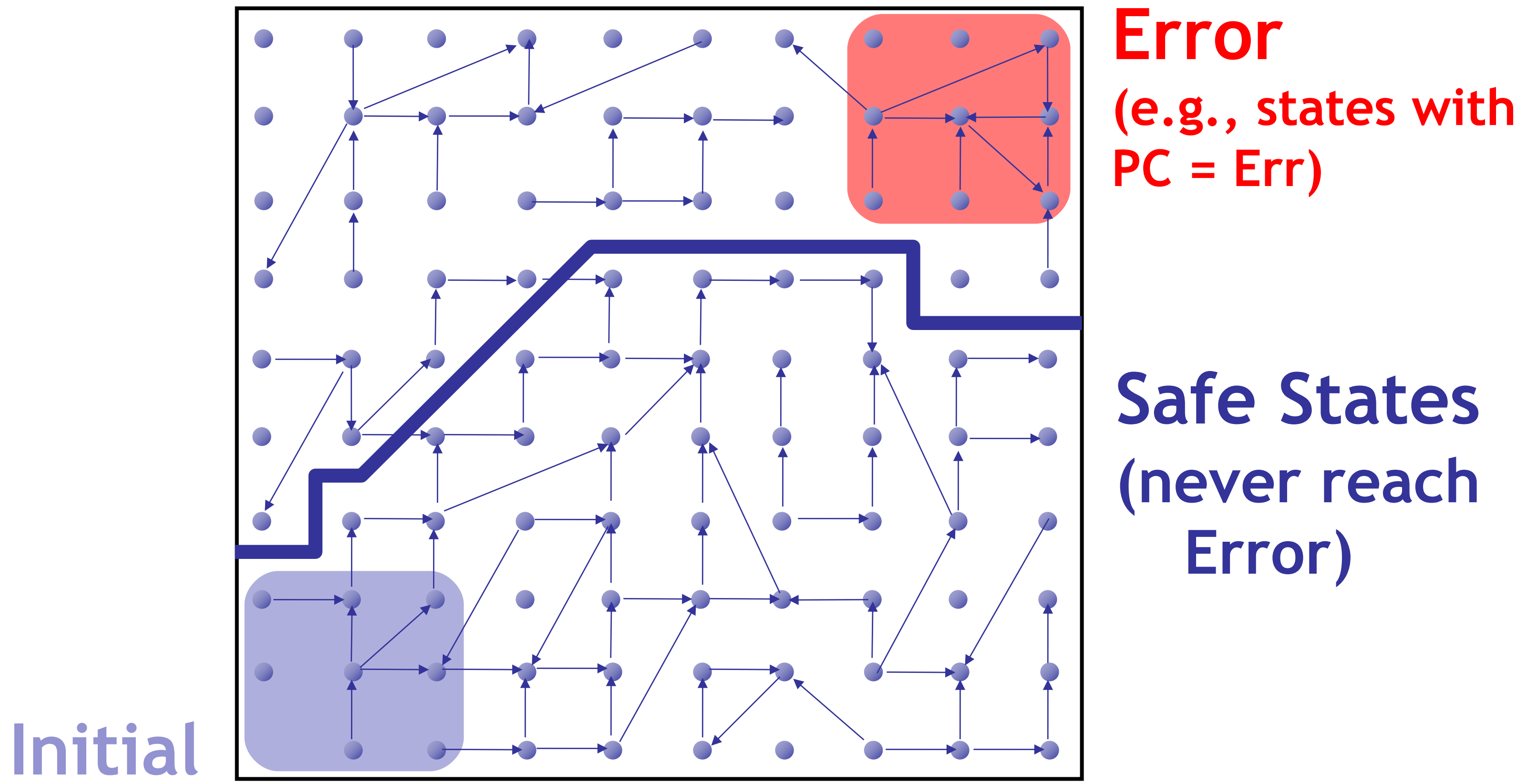
Program As
Labeled Transition System






```
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



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

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

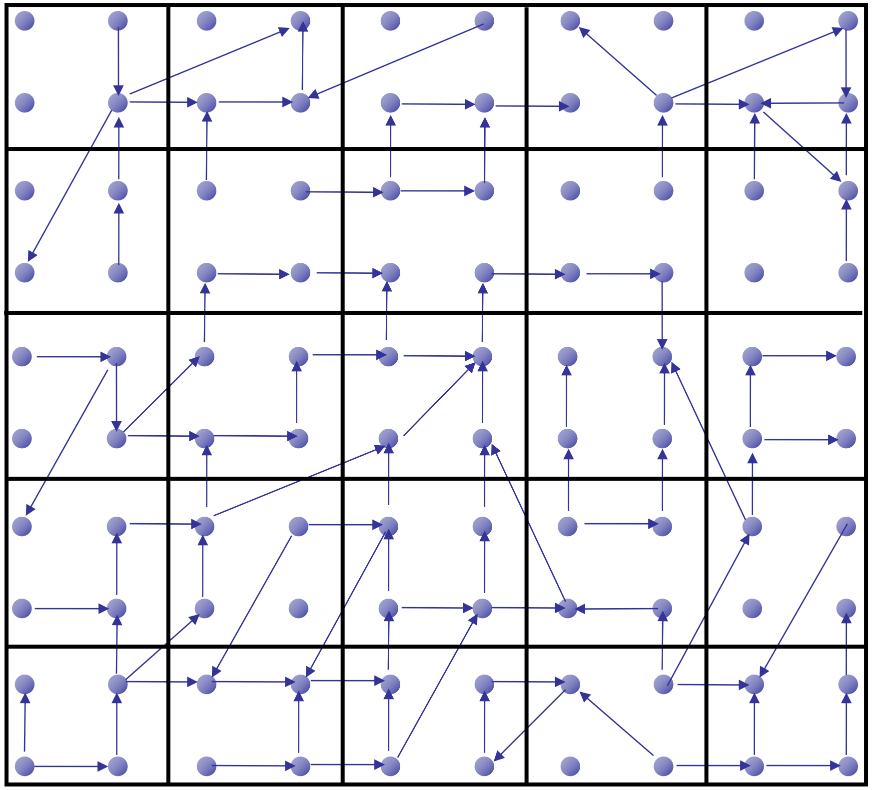
Solution : **Set** of states ≃ logical **formula**

Representing [Sets of States] as *Formulas*

$[F]$ states satisfying F $\{s \mid s \models F\}$	F FO fmla over prog. vars
$[F_1] \cap [F_2]$	$F_1 \wedge F_2$
$[F_1] \cup [F_2]$	$F_1 \vee F_2$
$\overline{[F]}$	$\neg F$
$[F_1] \subseteq [F_2]$	$F_1 \Rightarrow F_2$

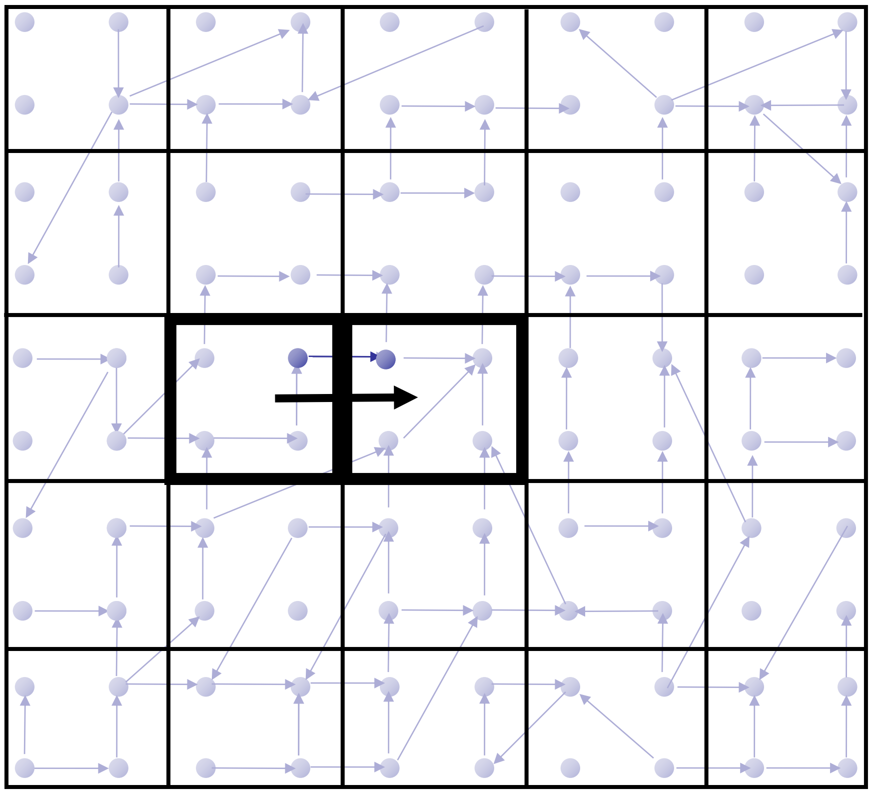
i.e. $F_1 \wedge \neg F_2$ unsatisfiable

Idea 1: Predicate Abstraction

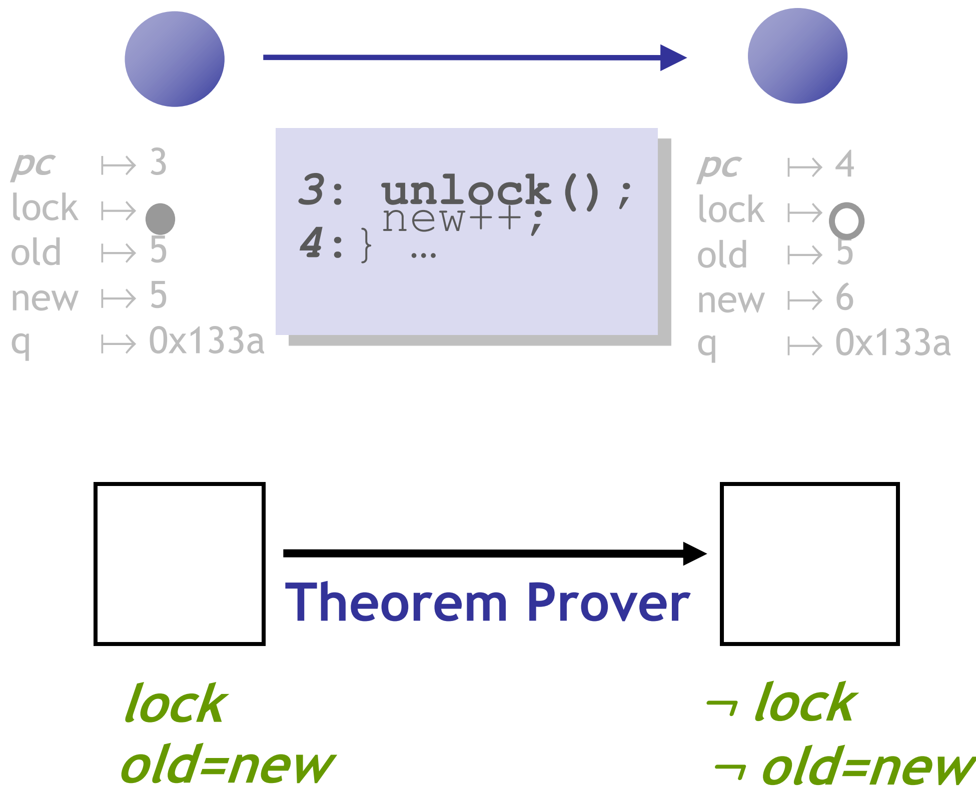


- **Predicates** on program state:
  lock *(i.e., lock=true)*
  old = new
- States satisfying **same** predicates are **equivalent**
  - **Merged** into one **abstract state**
- #abstract states is **finite**
  - **Thus model-checking the abstraction will be feasible!**

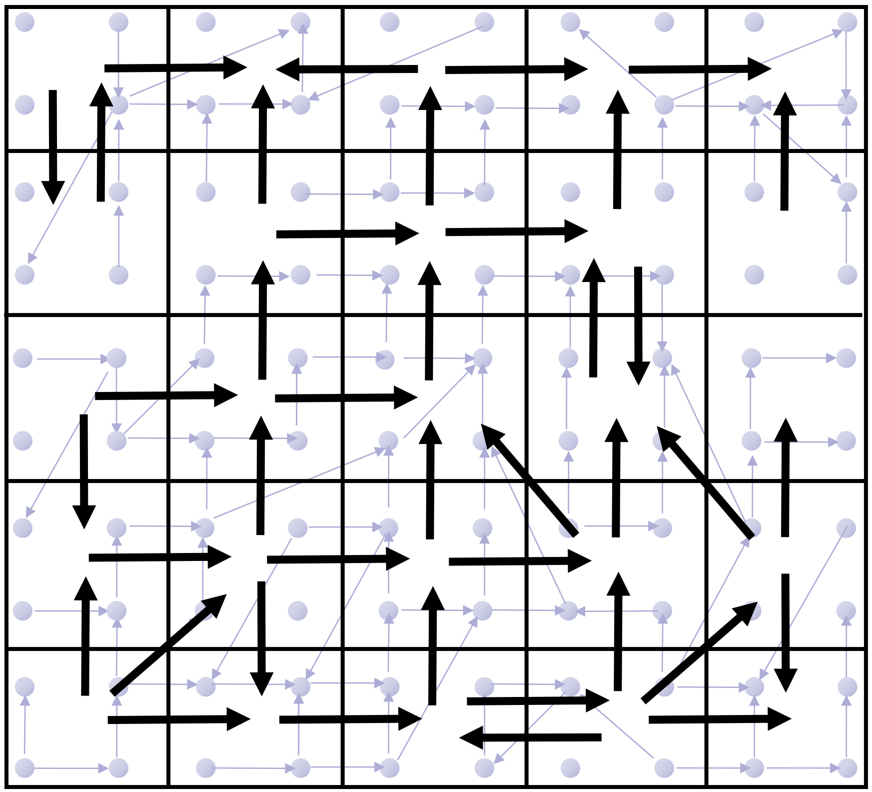
Abstract States and Transitions

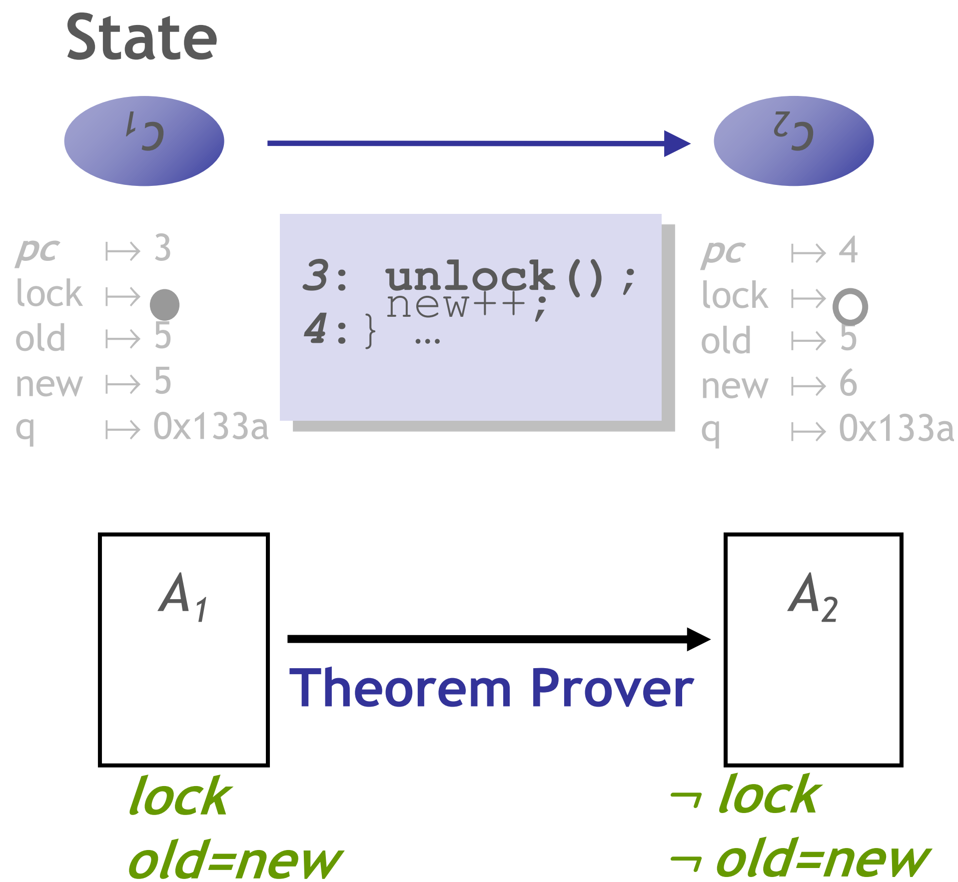


State



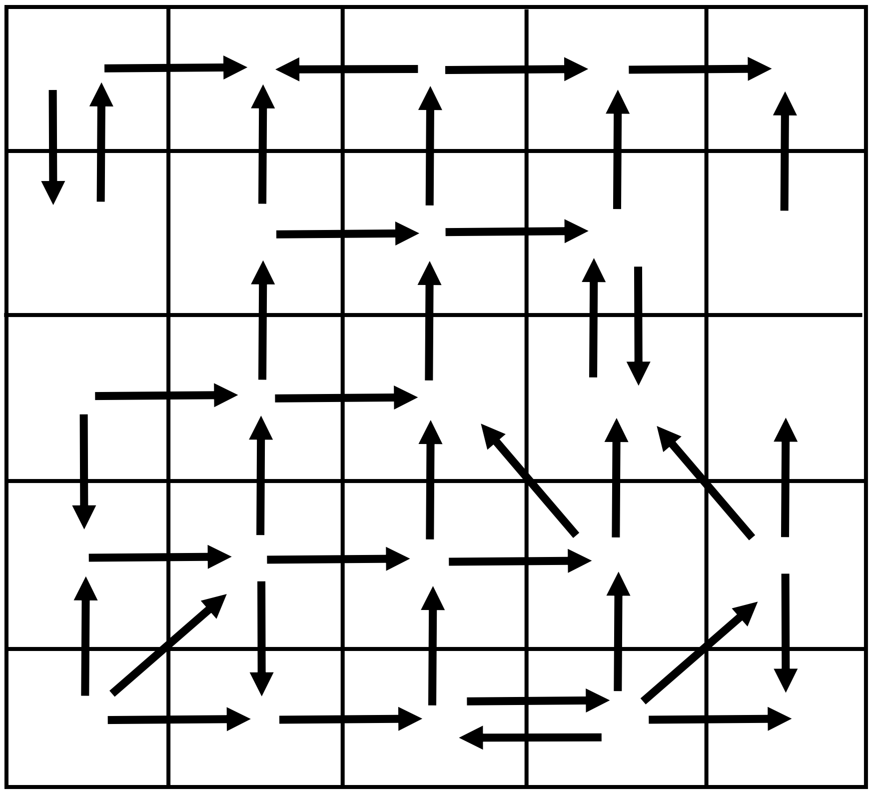
Abstraction

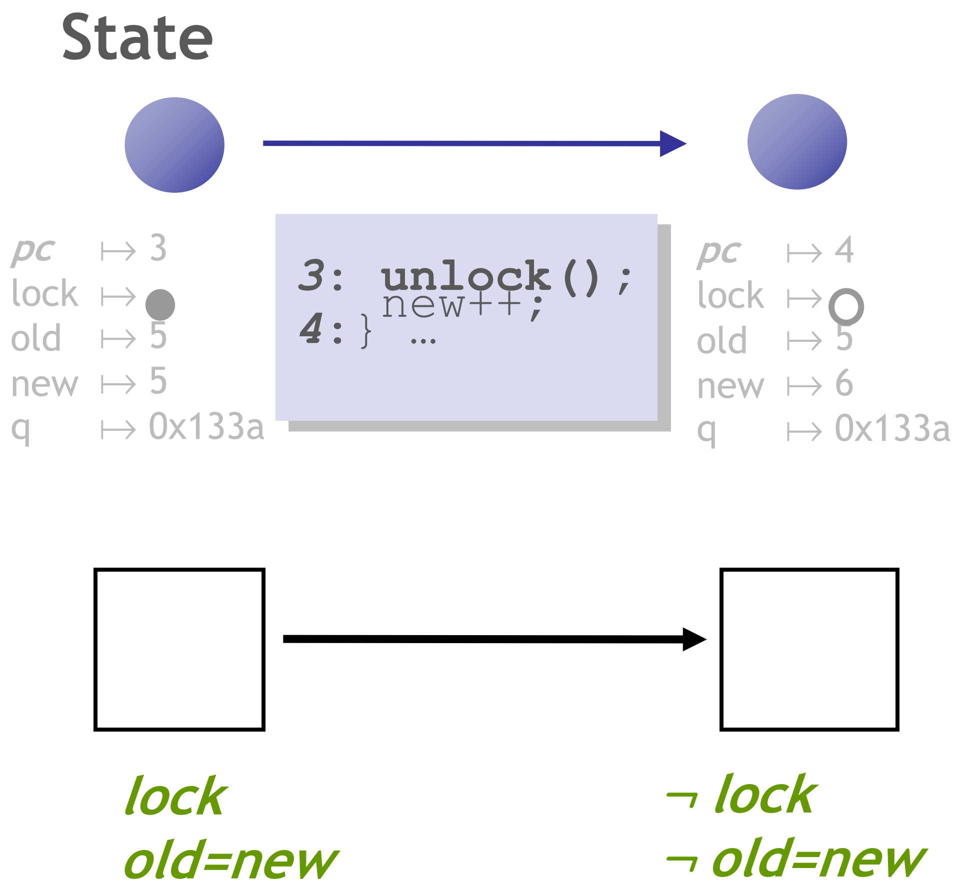




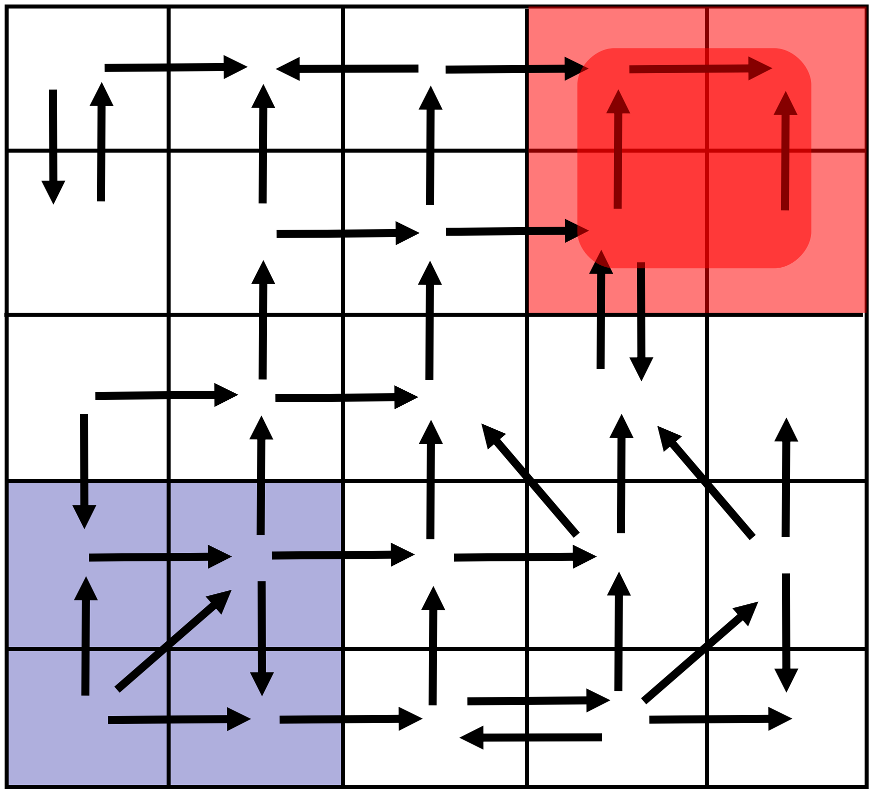
Existential Lifting
(i.e., $A_1 \rightarrow A_2$ iff $\exists c_1 \in A_1.\ \exists c_2 \in A_2.\ c_1 \rightarrow c_2$)

Abstraction





Analyze Abstraction

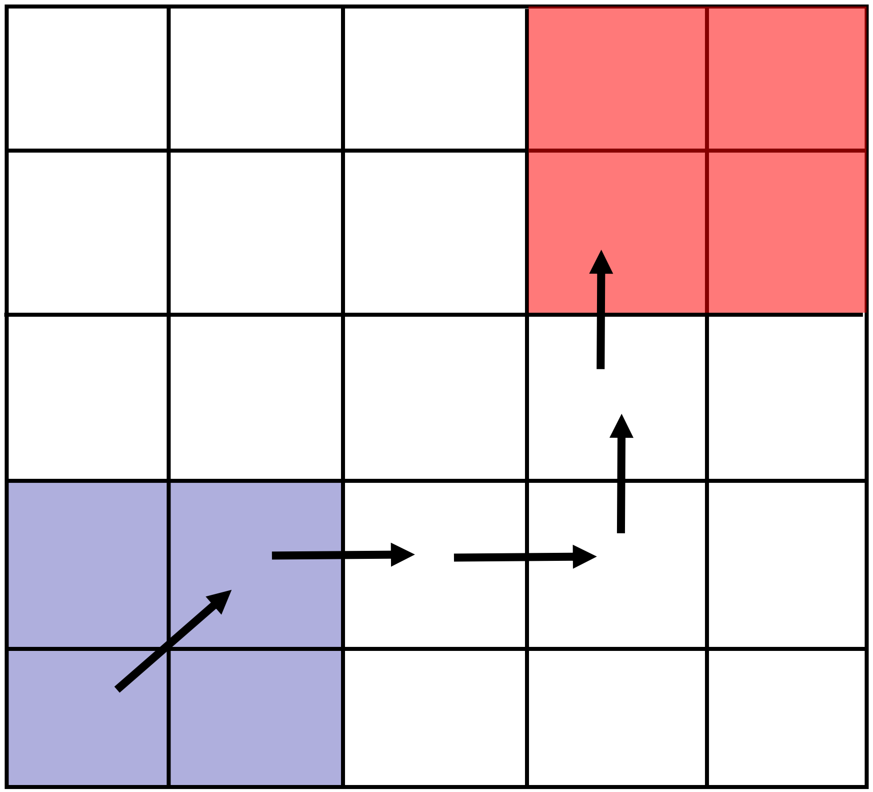


Analyze finite graph

Over Approximate:
Safe $\Rightarrow$ System Safe
No **false negatives**

Problem
Spurious **counterexamples**

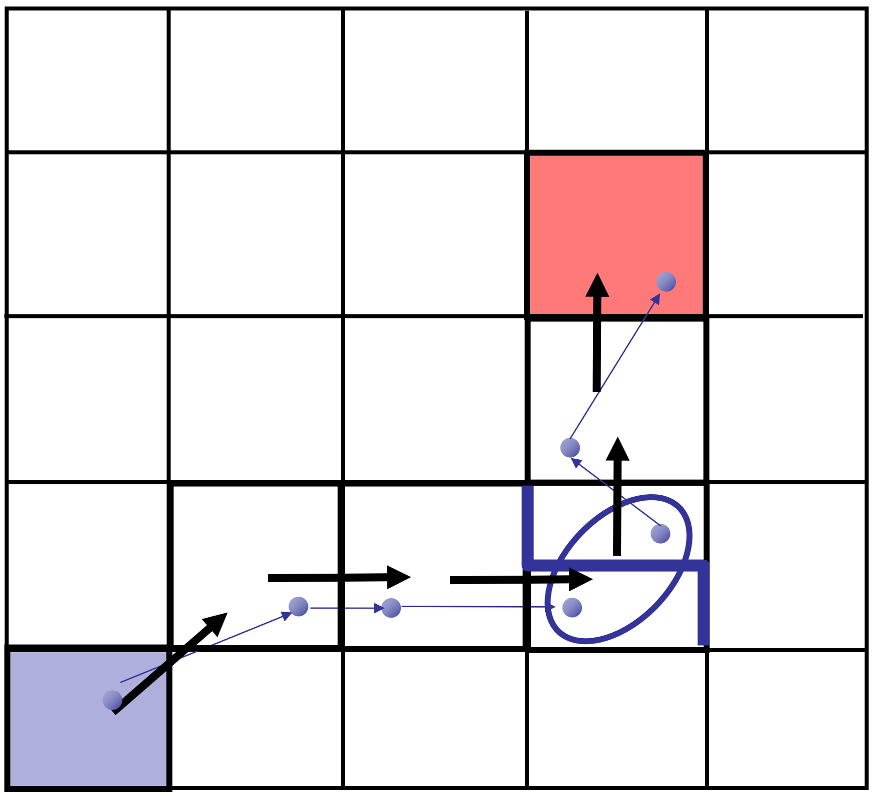
Idea 2: Counterex.-Guided Refinement



Solution

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

Idea 2: Counterex.-Guided Refinement



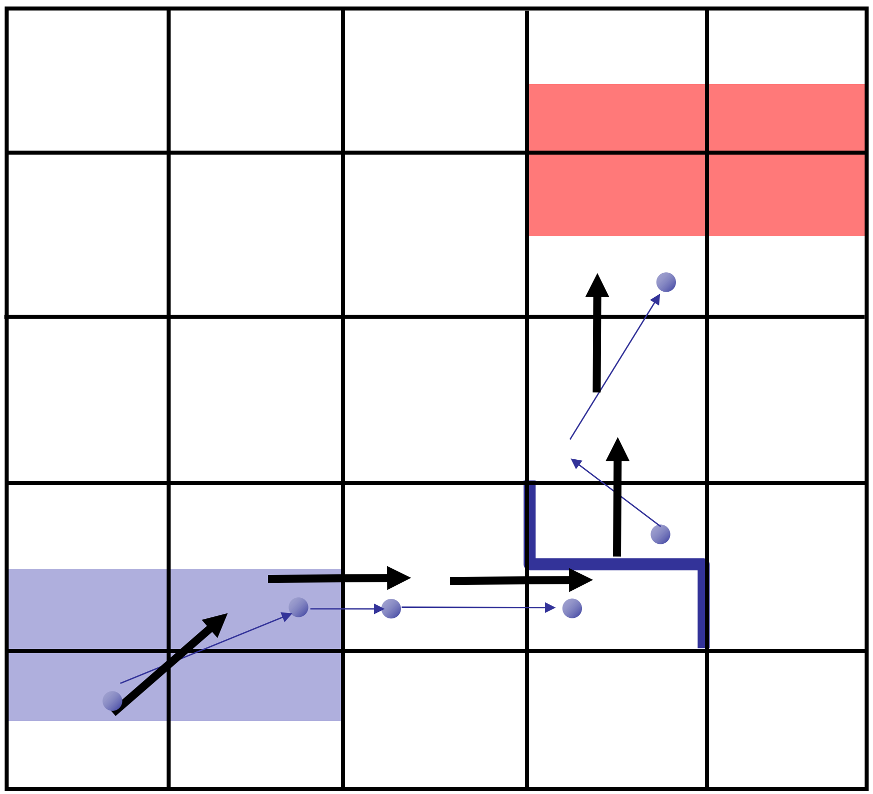
Solution

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



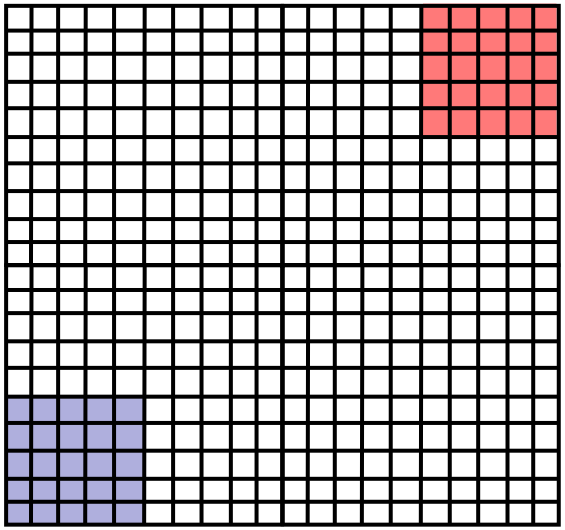
[Kurshan et al 93] [Clarke et al 00]
[Ball-Rajamani 01]

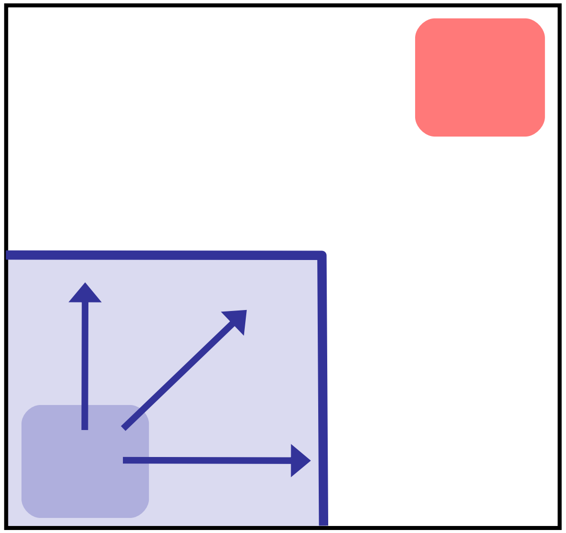
Solution

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

1. Add predicates to distinguish states across **cut**
2. Build **refined** abstraction
   -eliminates counterexample
3. **Repeat** search
   Untill real counterexample or system proved safe

Problem: Abstraction is Expensive





Reachable

Problem

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

Exponential Thm. Prover queries

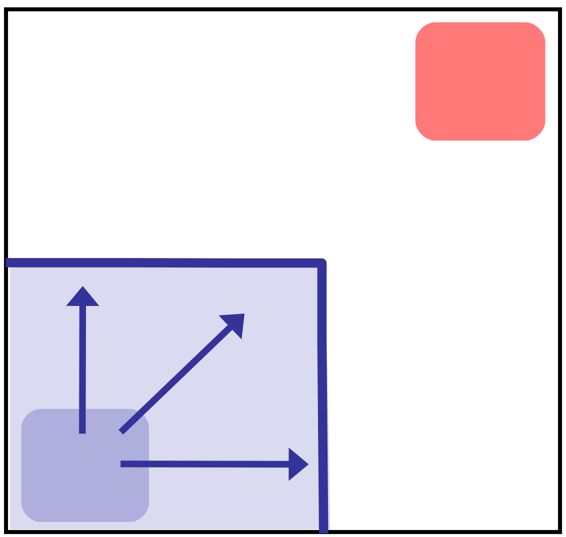
Observe

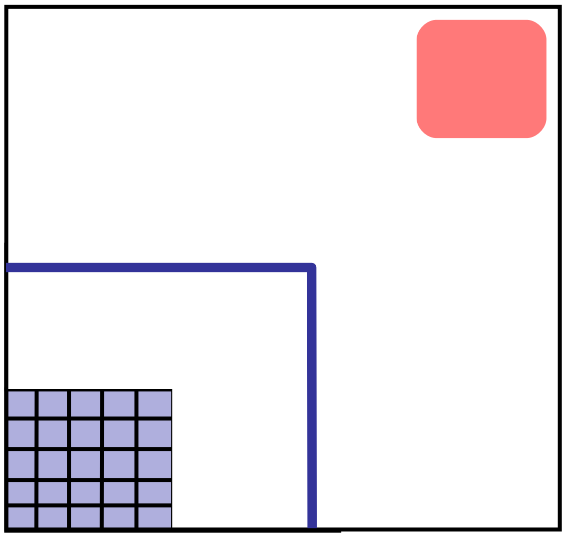
Fraction of state space reachable

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

#Reach ~ 1000's

Solution1: Only Abstract Reachable States





Problem

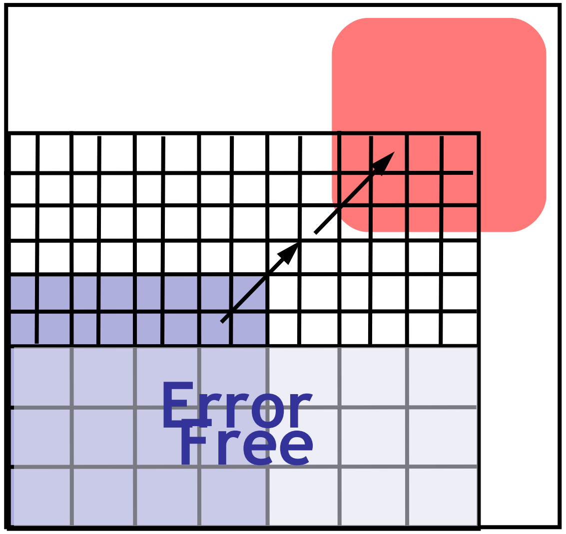
#abstract states = $2^{\#predicates}$

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

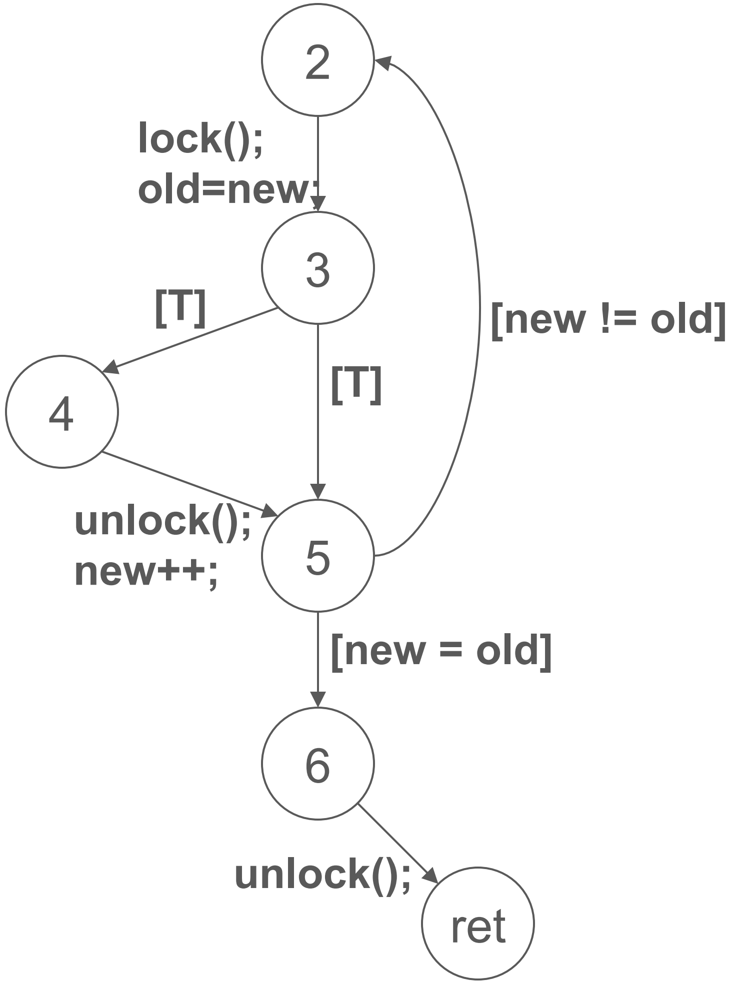
Don’t refine error-free regions

Build reachability tree.

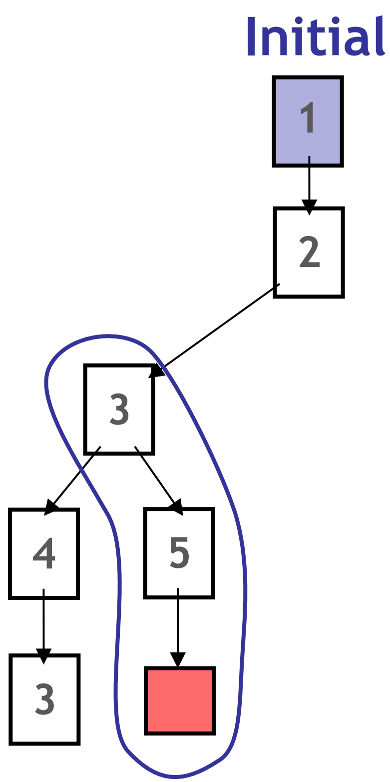
- Generate Abstract Reachability Tree
  - Contains all reachable nodes
  - Annotates each node with state
    - Initially LOCK = 0 or LOCK = 1
    - Cross product of CFA and data flow abstraction
- Algorithm: depth-first search
  - Generate nodes one by one
  - If you come to a node that's already in the tree, stop
    - This state has already been explored through a different control flow path
  - If you come to an error node, stop

Less abstractly: build reachability tree

```
2:  do {
          lock();
          old = new;
3:        if (*){
4:           unlock();
             new++;
          }
5:  } while (new != old);
6:  unlock();
    return;
```



Key Idea: Reachability Tree



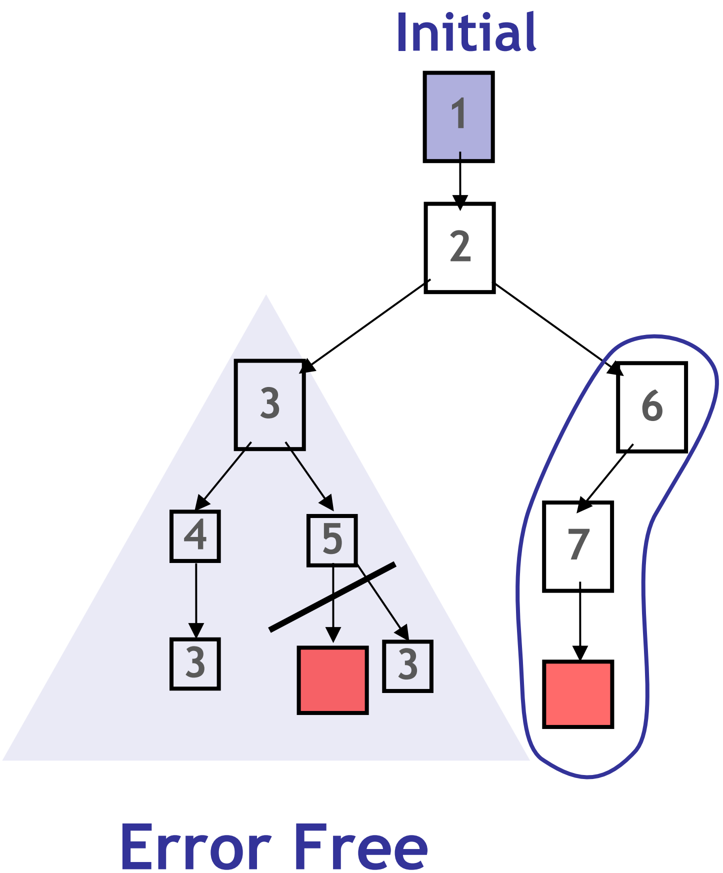
Unroll Abstraction

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



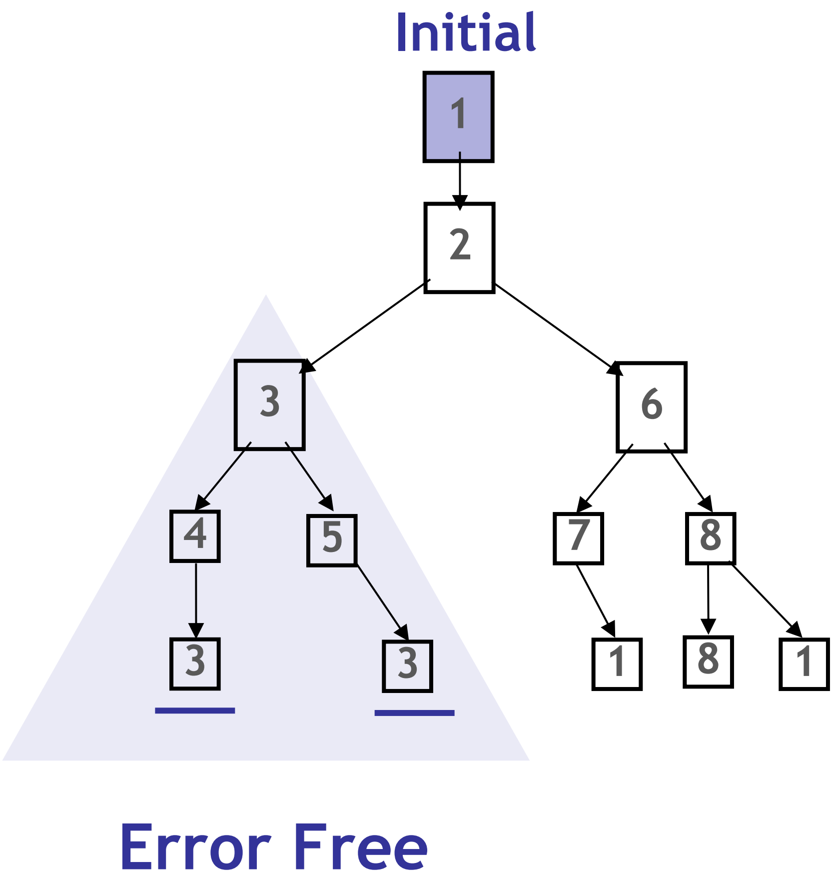
Unroll Abstraction

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

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

Find min spurious suffix

- Learn new predicates
- Rebuild subtree with new preds.

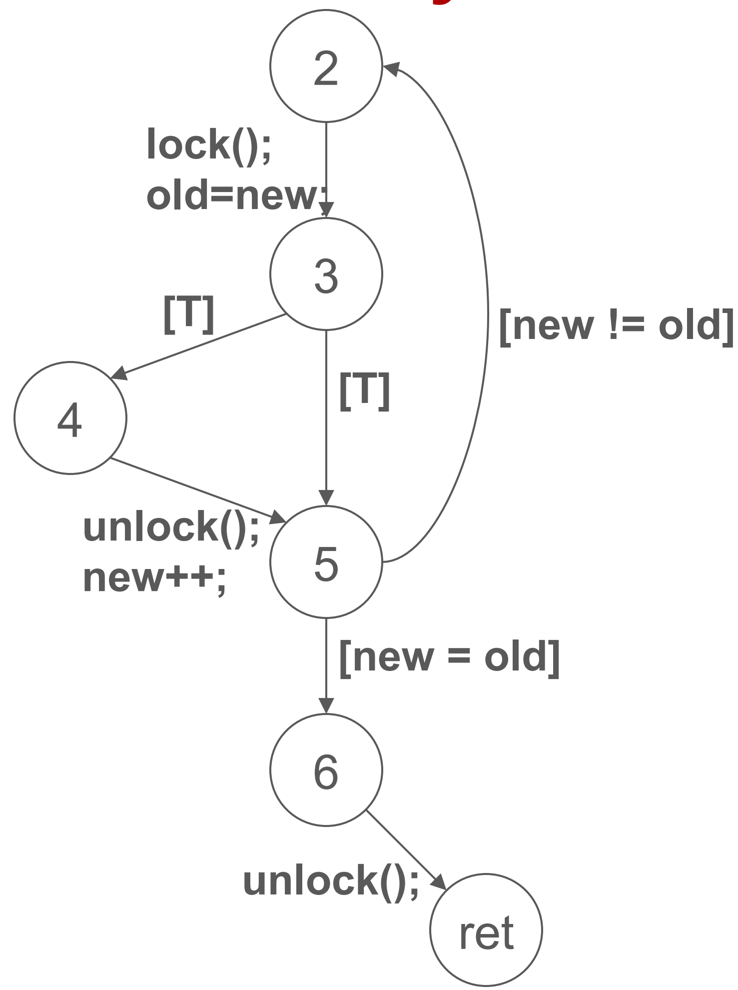
SAFE

S1: Only Abstract Reachable States

S2: Don't refine error-free regions

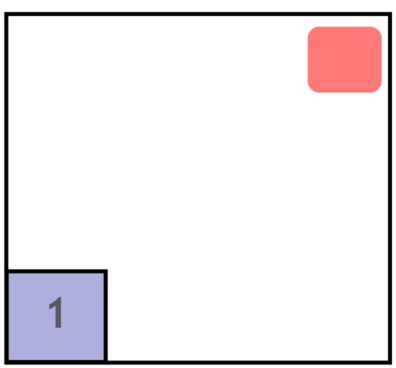
Less abstractly: build reachability tree

```
2:   do {
            lock();
            old = new;
3:          if (*){
4:              unlock();
                new++;
            }
5:   } while (new != old);
6:   unlock();
     return;
```



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 ();
}
```

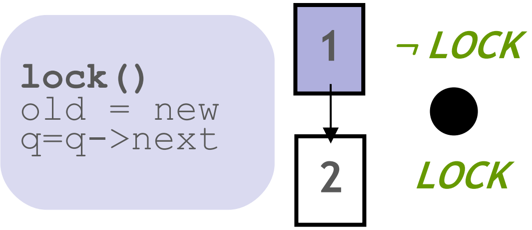


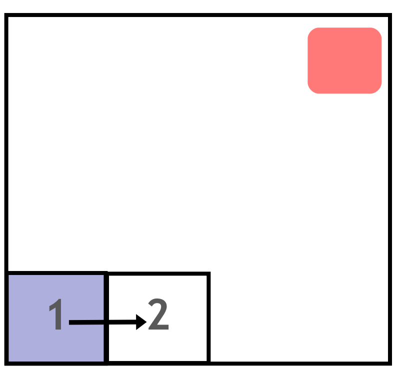
Predicates: *LOCK*

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 ();
}
```



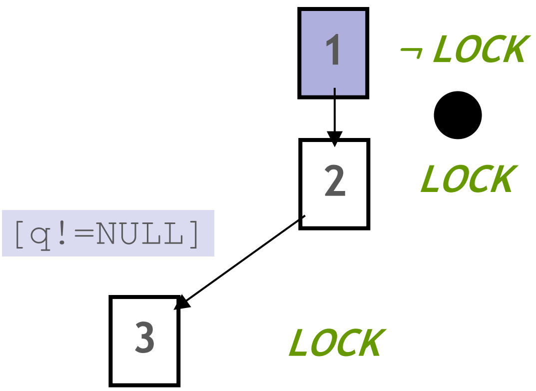


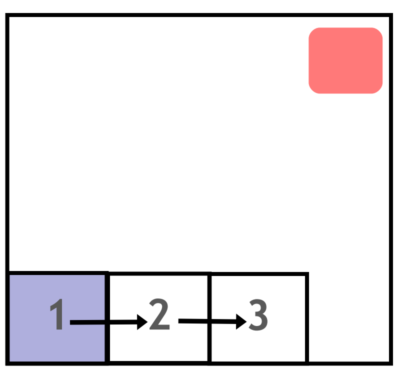
Predicates: *LOCK*

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 ();
}
```





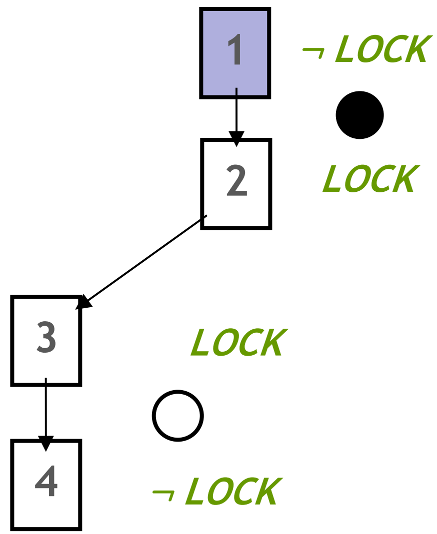
Predicates: *LOCK*

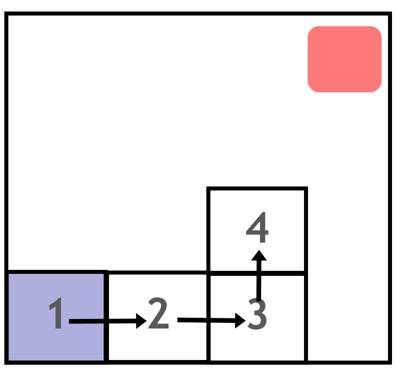
Reachability Tree

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 ();
}
```

```
q->data = new
unlock()
new++
```



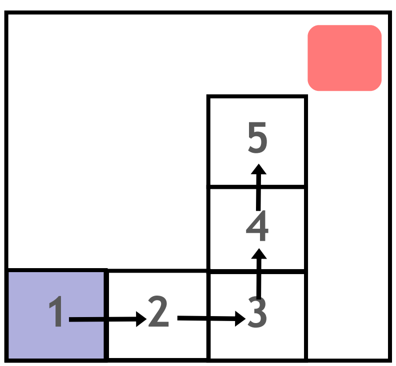


Predicates: *LOCK*

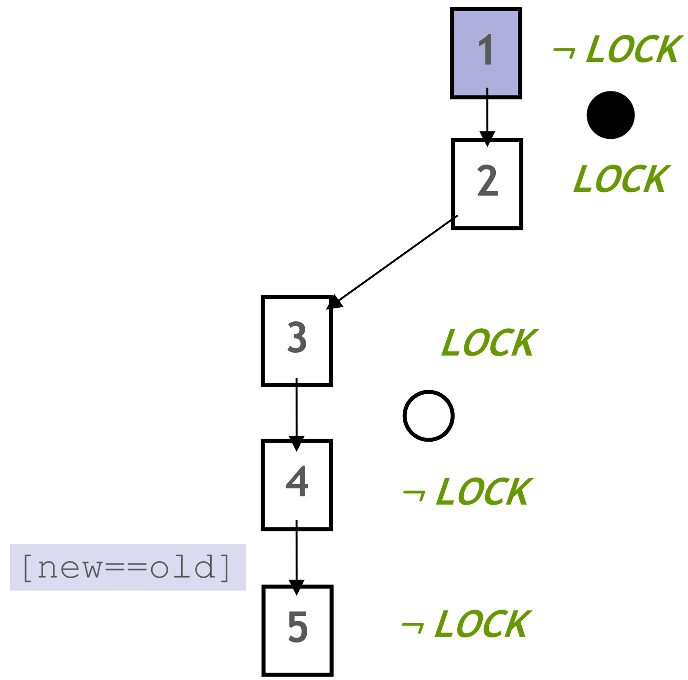
Reachability Tree

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 ();
}
```



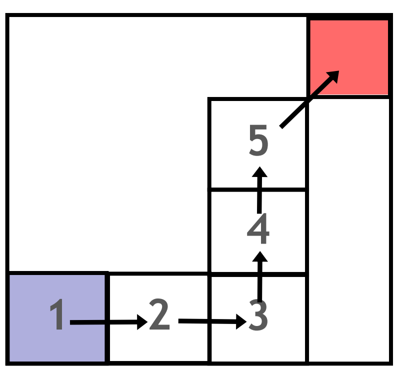
Predicates: *LOCK*



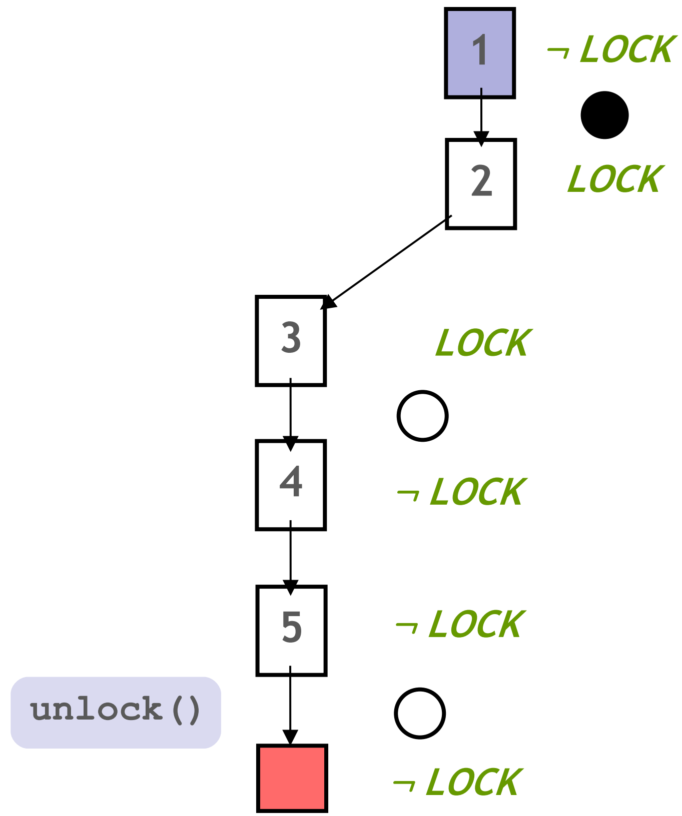
Reachability Tree

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 ();
}
```

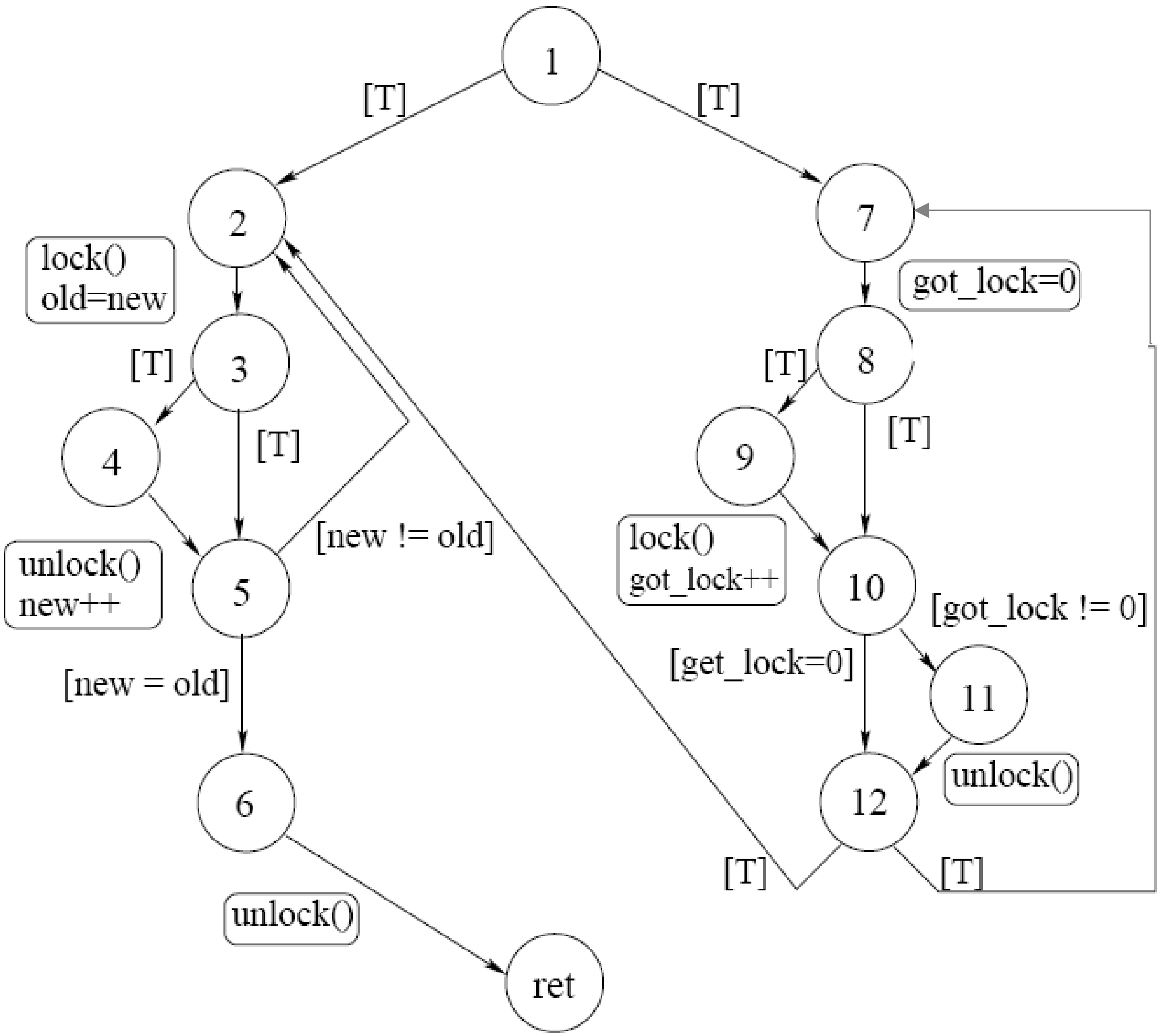


Predicates: *LOCK*



Reachability Tree

Depth First Search Example

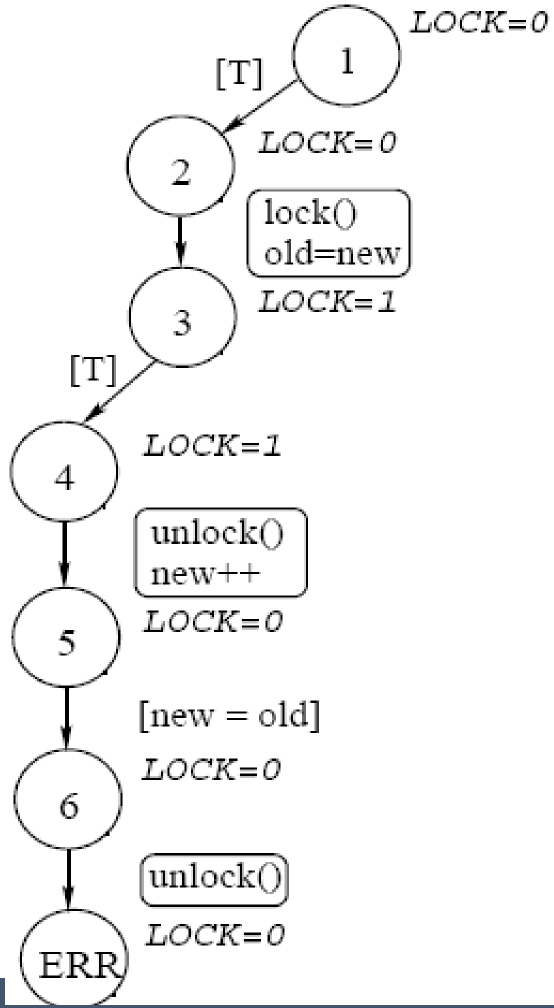


Is the Error Real?

- Use weakest preconditions to find out the weakest precondition that leads to the error
  - If the weakest precondition is false, there is no initial program condition that can lead to the error
  - Therefore the error is spurious
- Blast uses a variant of weakest preconditions
  - creates a new variable for each assignment before using weakest preconditions
  - Instead of substituting on assignment, adds new constraint
  - Helps isolate the reason for the spurious error more effectively

Is the Error Real?

- assume True;
- lock();
- old = new;
- assume True;
- unlock();
- new++;
- assume new==old
- error (lock==0)



Model Locking as Assignment

- assume True;
- lock = 1;
- old = new;
- assume True;
- lock = 0;
- new = new + 1;
- assume new==old
- error (lock==0)

Index the Variables

- assume True;
- lock1 = 1
- old1 = new1;
- assume True;
- lock2 = 0
- new2 = new1 + 1
- assume new2==old1
- error (lock2==0)

Generate Weakest Preconditions

- assume True;
- lock1 = 1
- old1 = new1;
- assume True;
- lock2 = 0
- new2 = new1 + 1
- assume new2==old1
- error (lock2==0)

$\wedge$ True
$\wedge$ lock1==1
$\wedge$ old1==new1
$\wedge$ True
$\wedge$ lock2==0
$\wedge$ new2==new1+1
$\wedge$ new2==old1
lock2==0

Contradictory!

Relevant Sidebar: Craig Interpolation

- Given an unsatisfiable formula A ∧ B, the Craig Interpolant I is a formula such that:
  - A → I
  - I ∧ B is unsatisfiable
  - I only refers to variables mentioned in both A and B
- It is guaranteed to exist, proof elided.

- ∧ True
- ∧ lock1==1
- ∧ old1==new1
- ∧ True
- ∧ lock2==0
- ∧ new2==new1+1
- ∧ new2==old1
- lock2==0

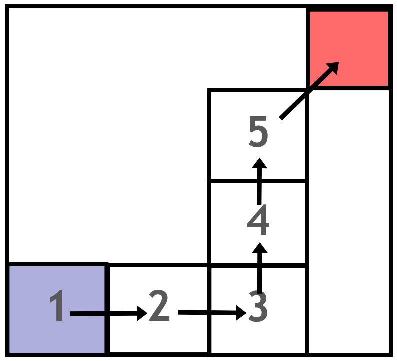
Why is the Error Spurious?

- More precisely, what predicate could we track that would eliminate the spurious error message?
- Consider, for each node, the constraints generated before that node (c1) and after that node (c2)
- Find a condition I such that
  - c1 => I
    - I is true at the node
  - I only contains variables mentioned in both c1 and c2
    - I mentions only variables in scope (not old or future copies)
  - I ∧ c2 = false
    - I is enough to show that the rest of the path is infeasible
  - I is guaranteed to exist
    - See Craig Interpolation

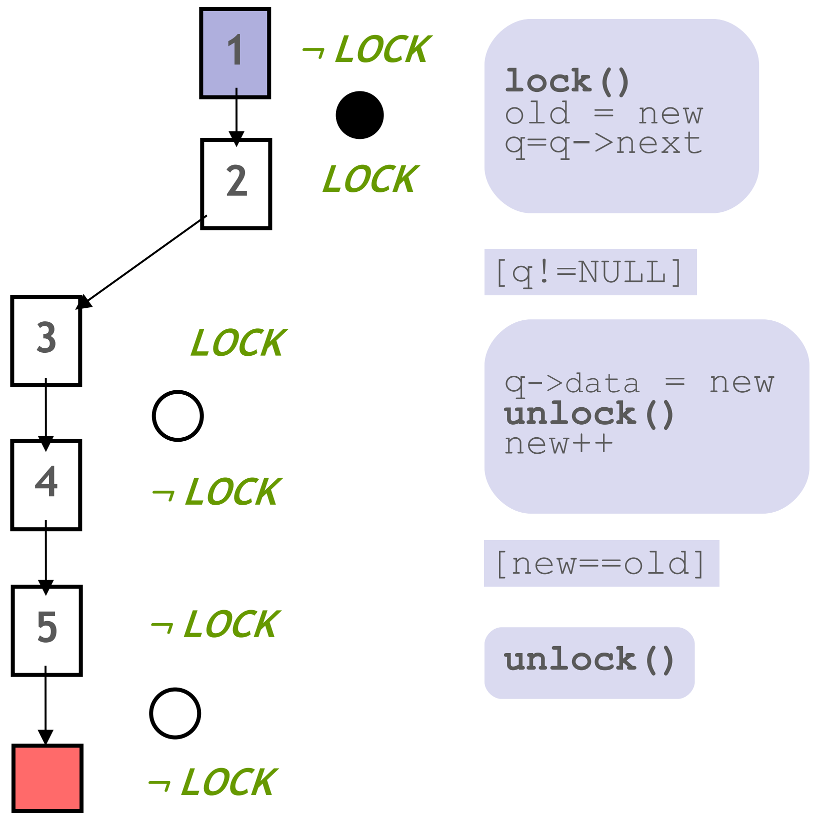
- ∧ True
- ∧ lock1==1
- ∧ old1==new1 ← Interpolant: old == new
- ∧ True
- ∧ lock2==0
- ∧ new2==new1+1
- ∧ new2==old1
- lock2==0

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 ();
}
```



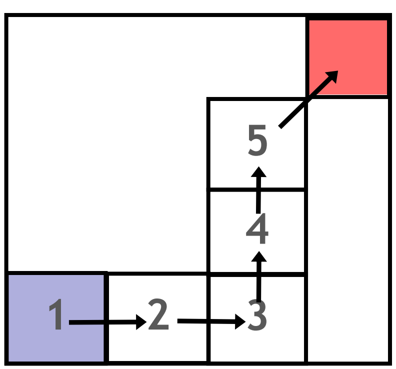
Predicates: *LOCK*



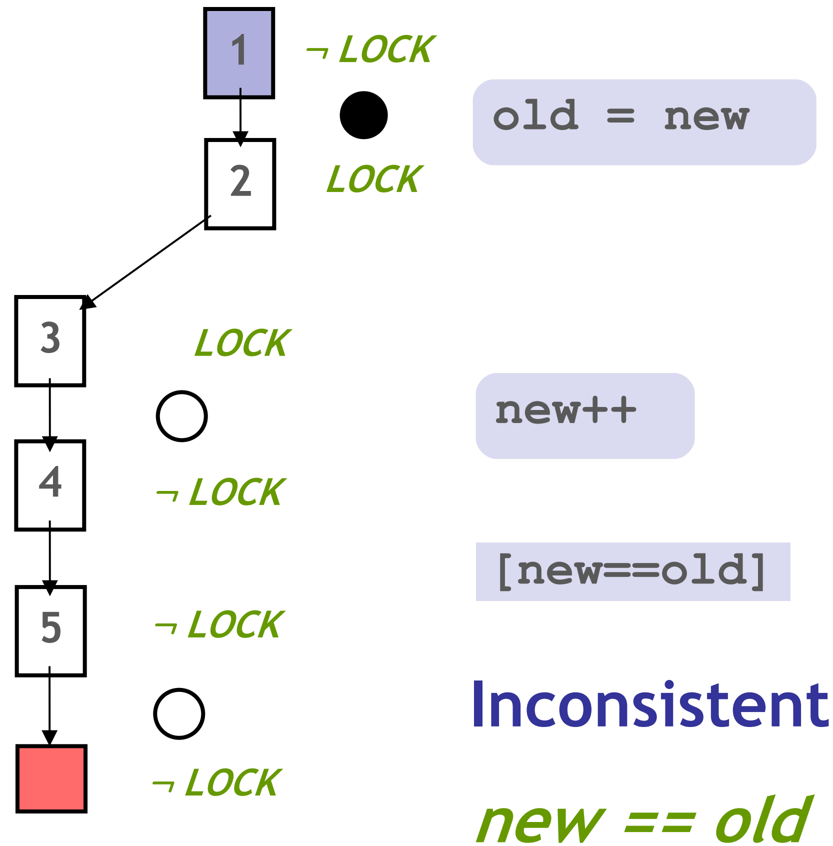
Reachability Tree

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*

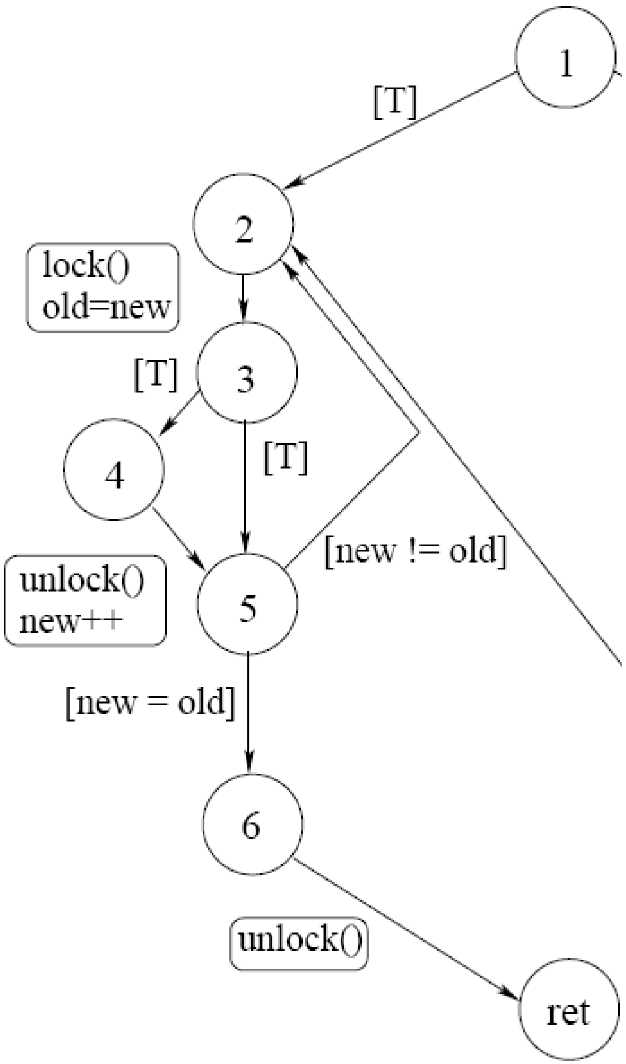


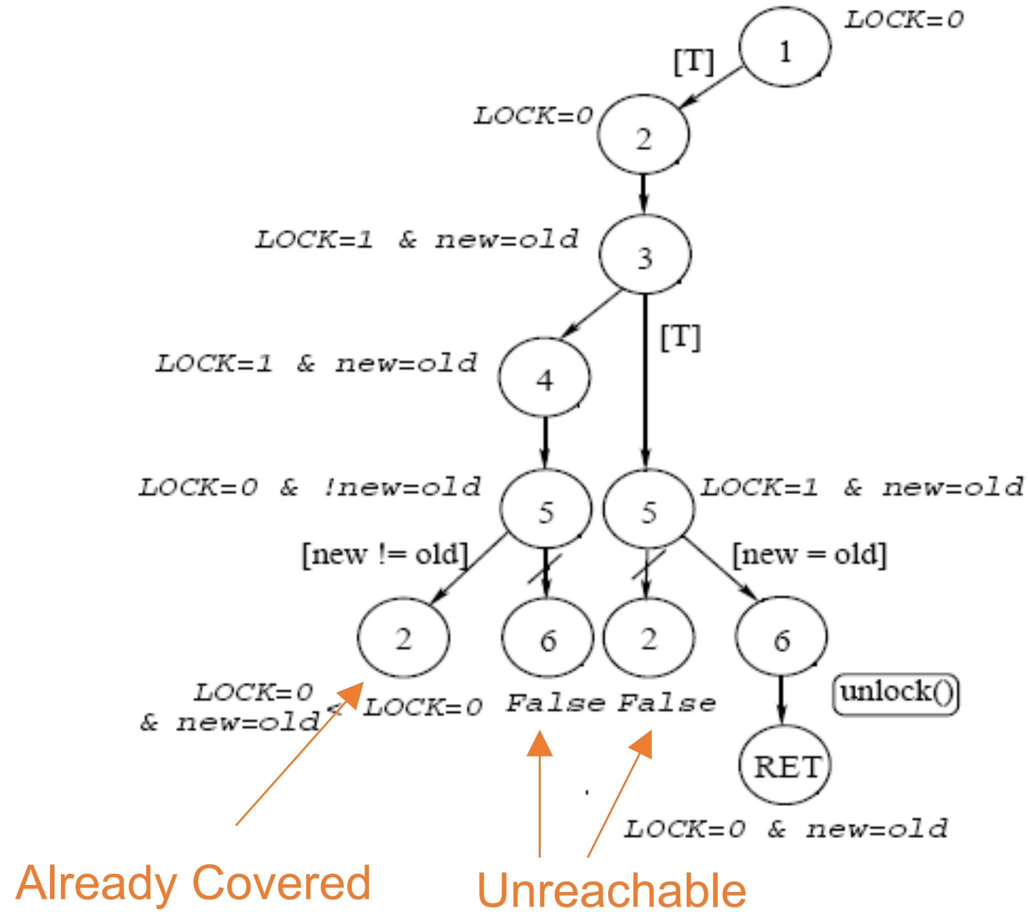
Reachability Tree

Reanalyzing the Program

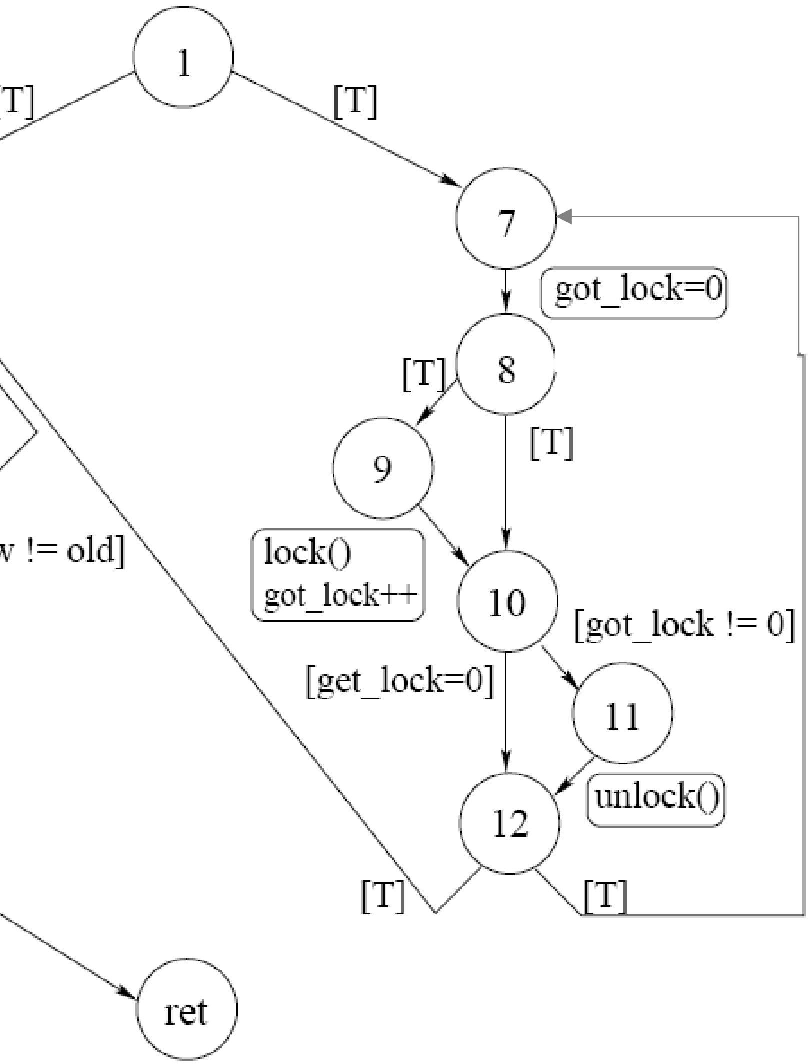
- Explore a subtree again
  - Start where new predicates were discovered
  - This time, track the new predicates
  - If the conjunction of the predicates on a node is false, stop exploring—this node is unreachable

Reanalysis Example





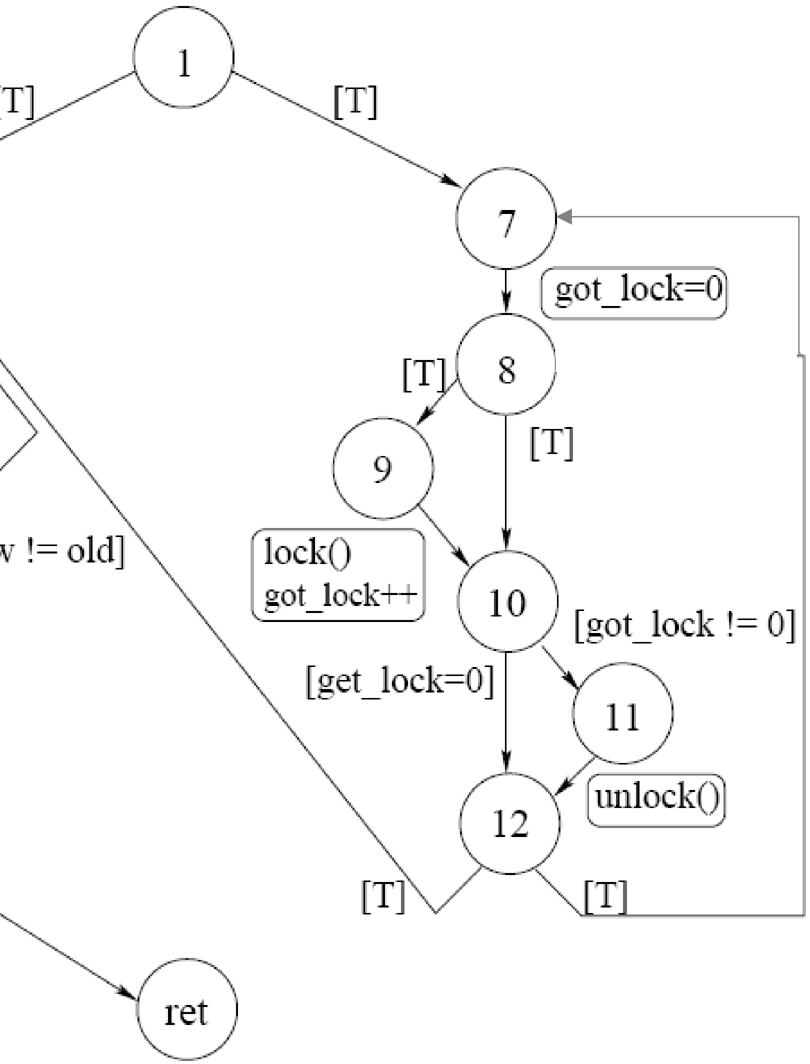
Analyzing the Right Hand Side



Exercise: run weakest preconditions from the unlock() at the end of the path 1-7-8-10-11-12.

Recall that we model locking with a variable *lock*, so unlock() is an error if *lock*==0

Reanalysis




```
 Example() {
1:   if (*){
7:      do {
                got_lock = 0;
8:              if (*){
9:                 lock();
                   got_lock++;
                }
10:             if (got_lock){
11:                unlock();
                }
12:      } while (*)
     }
```

Key: L = locked=1
Z = got_lock=0

Generate Weakest Preconditions

- assume True;
- got_lock = 0;
- assume True;
- assume got_lock != 0;
- error (lock==0)

Why is the Error Spurious?

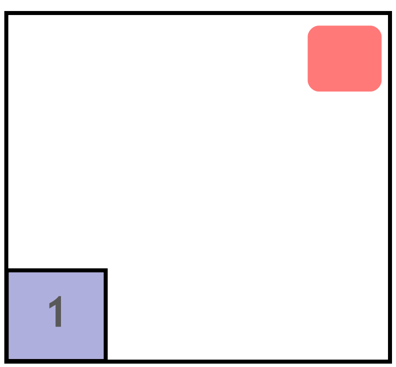
- More precisely, what predicate could we track that would eliminate the spurious error message?
- Consider, for each node, the constraints generated before that node (c1) and after that node (c2)
- Find a condition I such that
  - c1 => I
    - I is true at the node
  - I only contains variables mentioned in both c1 and c2
    - I mentions only variables in scope (not old or future copies)
  - I $\wedge$ c2 = false
    - I is enough to show that the rest of the path is infeasible
  - I is guaranteed to exist
    - See Craig Interpolation

- $\wedge$ True
- $\wedge$ got_lock==0
- $\wedge$ True
- $\wedge$ got_lock!=0
- lock==0

Exercise: now find the Craig interpolant

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 ();
}
```



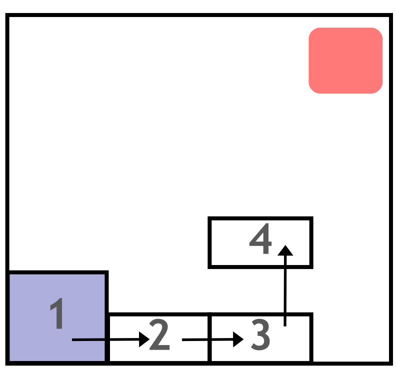
…but only at the minimum suffix!

Reachability Tree

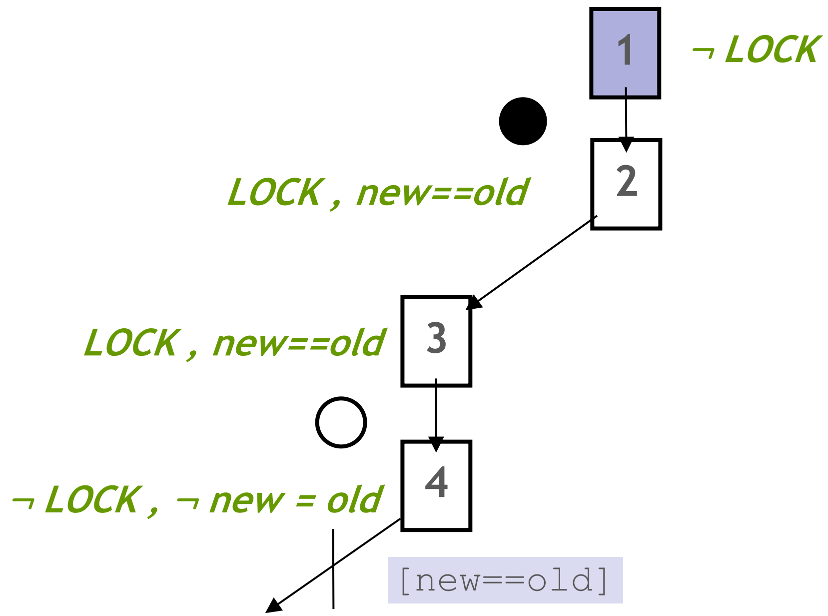
Predicates: *LOCK, new==old*

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 ();
}
```



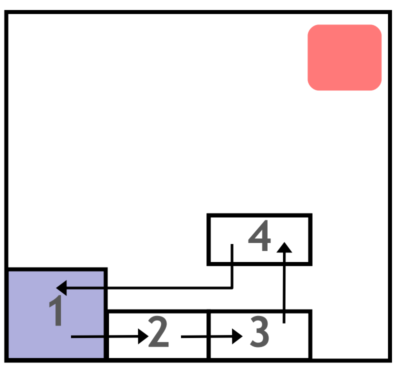
Predicates: *LOCK, new==old*



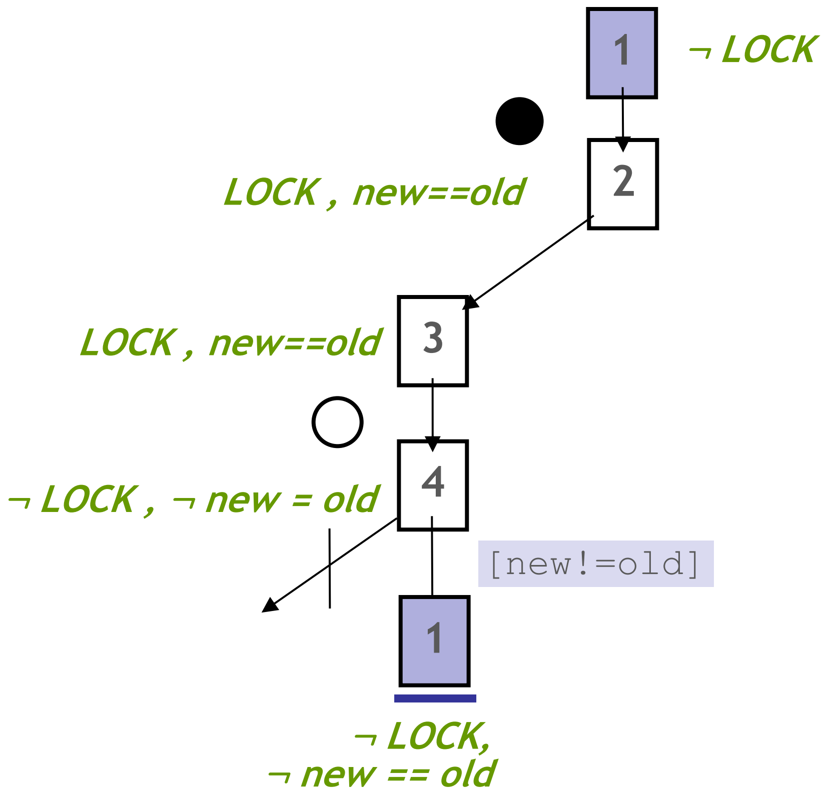
Reachability Tree

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 ();
}
```



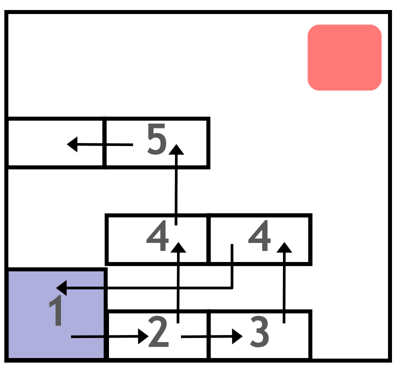
Predicates: *LOCK, new==old*



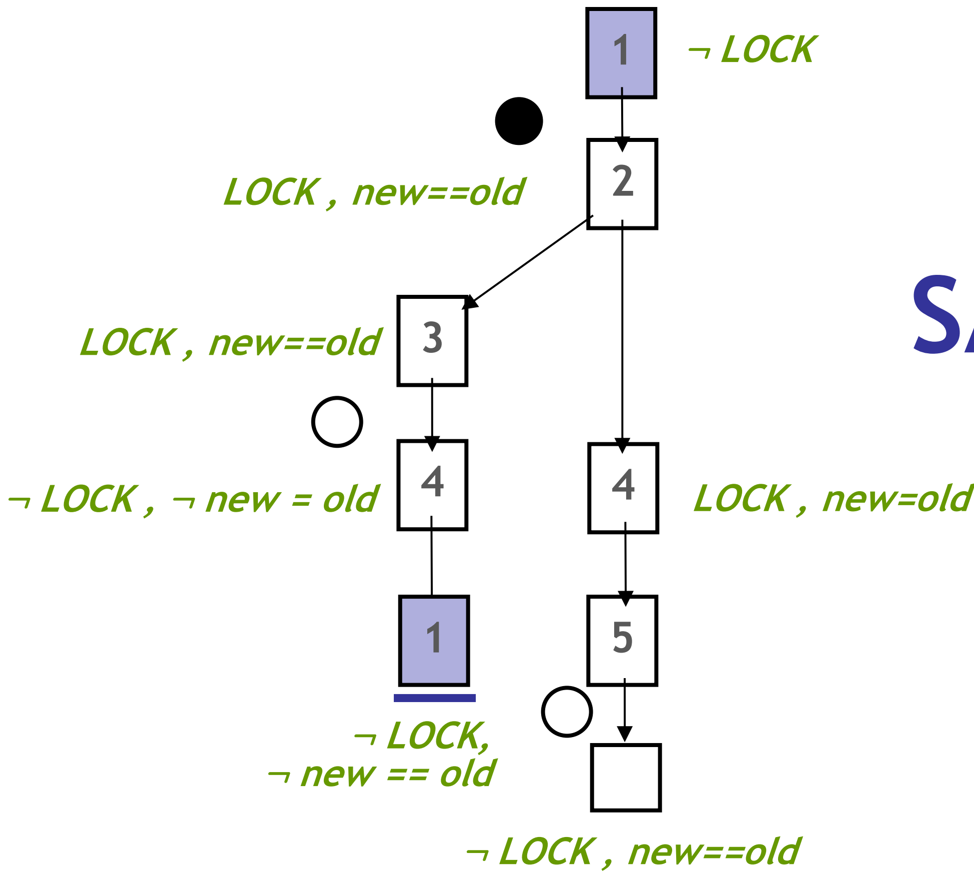
Reachability Tree

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 ();
}
```



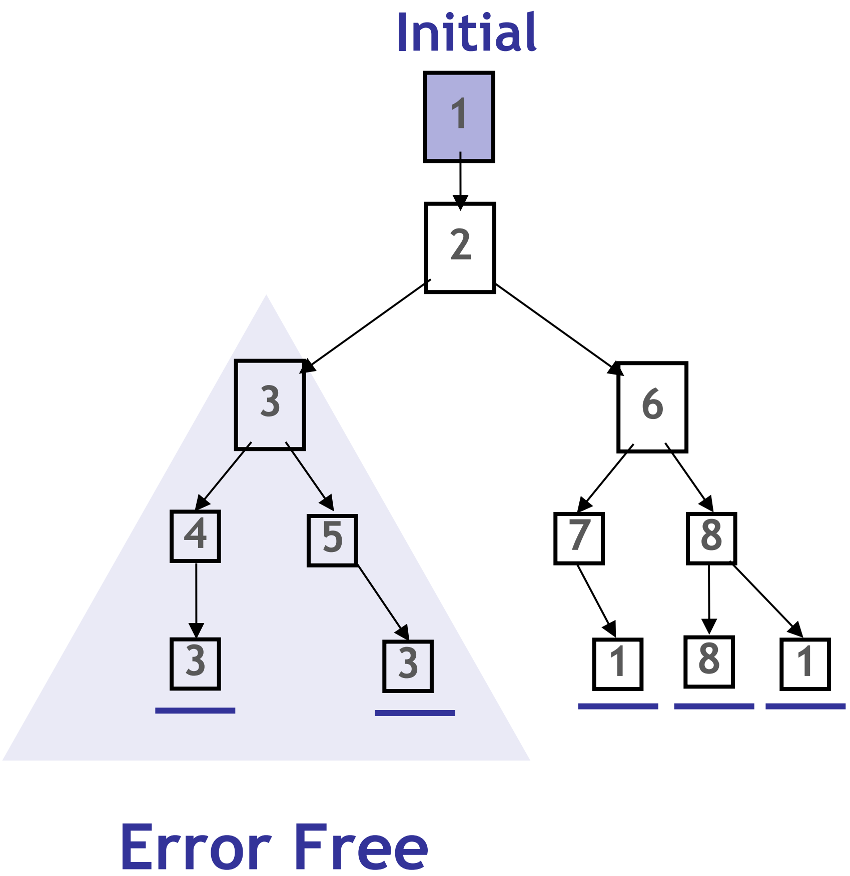
Predicates: *LOCK, new==old*



SAFE

Reachability Tree

Key Idea: Reachability Tree



Unroll

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

Find min spurious suffix

- Learn new predicates
- Rebuild subtree with new preds.

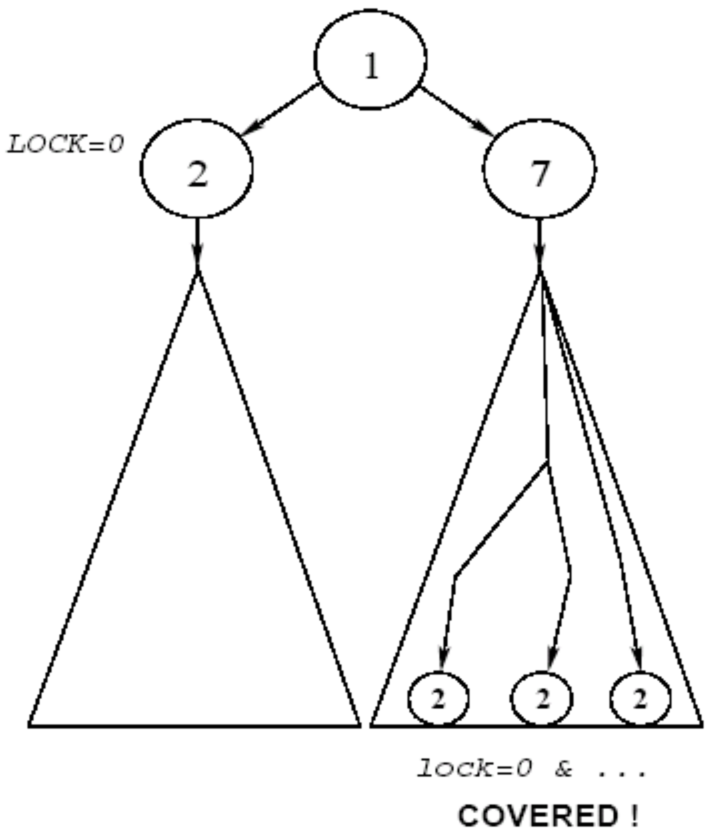
SAFE

S1: Only Abstract Reachable States

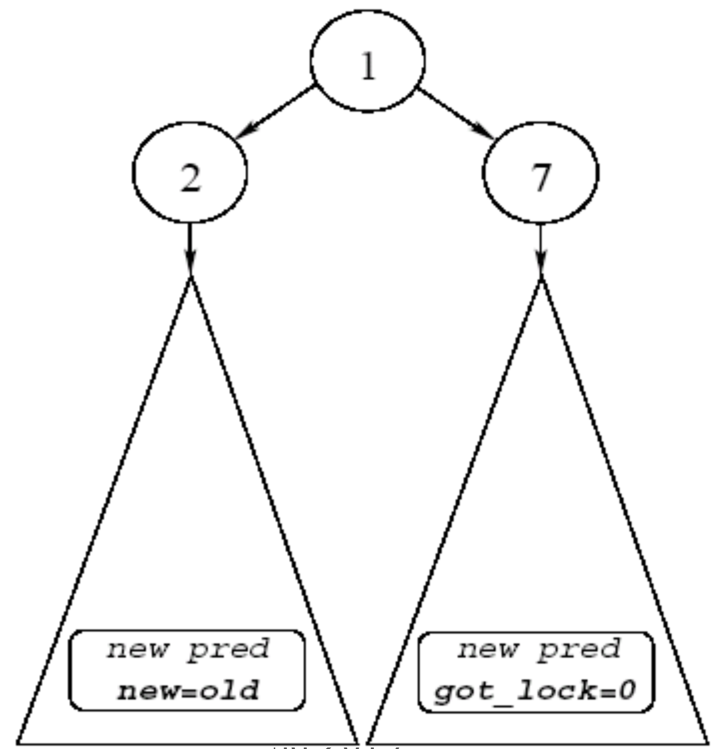
S2: Don't refine error-free regions

Blast Techniques, Graphically

- Explores reachable state, not all paths
  - Stops when state already seen on another path



- Lazy Abstraction
  - Uses predicates on demand
  - Only applies predicate to relevant part of tree



Experimental Results

Program	Postprocessed LOC	Predicates		BLAST Time (sec)	Ctrex analysis (sec)	Proof Size (bytes)
		Total	Active			
qpmouse.c	23539	2	2	0.50	0.00	175
ide.c	18131	5	5	4.59	0.01	253
aha152x.c	17736	2	2	20.93	0.00	
tlan.c	16506	5	4	428.63	403.33	405
cdaudio.c	17798	85	45	1398.62	540.96	156787
floppy.c	17386	62	37	2086.35	1565.34	
[fixed]		93	44	395.97	17.46	60129
kbfiltr.c	12131	54	40	64.16	5.89	
		48	35	256.92	165.25	
[fixed]		37	34	10.00	0.38	7619
mouclass.c	17372	57	46	54.46	3.34	
parport.c	61781	193	50	1980.09	519.69	102967

Termination

- Not guaranteed
  - The system could go on generating predicates forever
- Can guarantee termination
  - Restrict the set of possible predicates to a finite subset
    - Finite height lattices in data flow analysis!
  - Those predicates are enough to predict observable behavior of program
    - E.g. the ordering of lock and unlock statements
    - Predicates are restricted in practice
      - E.g. likely can't handle arbitrary quantification as in Dafny
      - Model checking is hard if properties depend on heap data, for example
  - Can't prove arbitrary properties in this case
- In practice
  - Terminate abstraction refinement after a time bound

Key Points of CEGAR

- To prove a property, may need to strengthen it
  - Just like strengthening induction hypothesis
- CEGAR figures out strengthening automatically
  - From analyzing why errors are spurious
- Blast uses *lazy abstraction*
  - Only uses an abstraction in the parts of the program where it is needed
  - Only builds the part of the abstract state that is reached
  - Explored state space is ***much*** smaller than potential state space

Blast in Practice

- Has scaled past 100,000 lines of code
  - Realistically starts producing worse results after a few 10K lines
- Sound up to certain limitations
  - Assumes restricted (“safe”) use of C
    - No aliases of different types; how realistic?
  - No recursion, no function pointers
  - Need models for library functions
- Has also been used to find memory safety errors, race conditions, generate test cases