Lecture 12

Predicate Abstraction

Zvonimir Rakamarić
University of Utah

slides acknowledgements: Ranjit Jhala

Last Time and Announcements

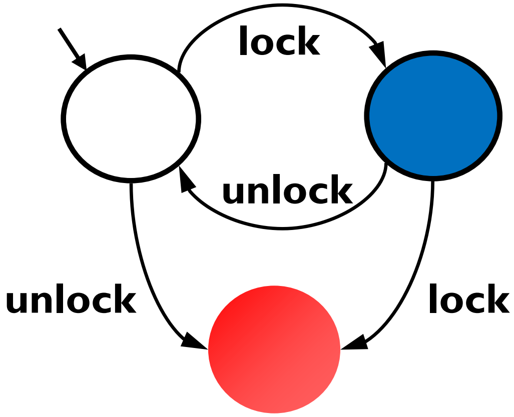
- Strategies for proving programs correct

- Snow an example of a vacuous proof in Dafny
- Graded homework 3
- Posted homework 4
  - Due on Thursday next week
  - No homework assignments over spring break

Checking Interface Usage Rules

- Interface rules in documentation
  - Define order of operations
  - Often incomplete and imprecise
- Violated rules cause bad behavior
  - System crash or deadlock
  - Unexpected exceptions
  - Failed runtime checks

Example Property: Double Locking

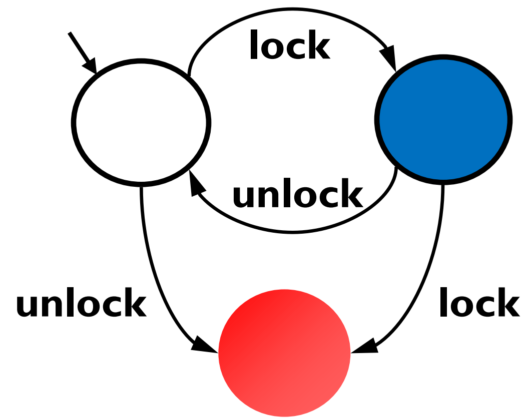


“An attempt to re-acquire an acquired lock or release a released lock will cause a **deadlock***.”*

Calls to lock and unlock must **alternate**.

Example Program

```
   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;
   }
```



Program Transitions

State

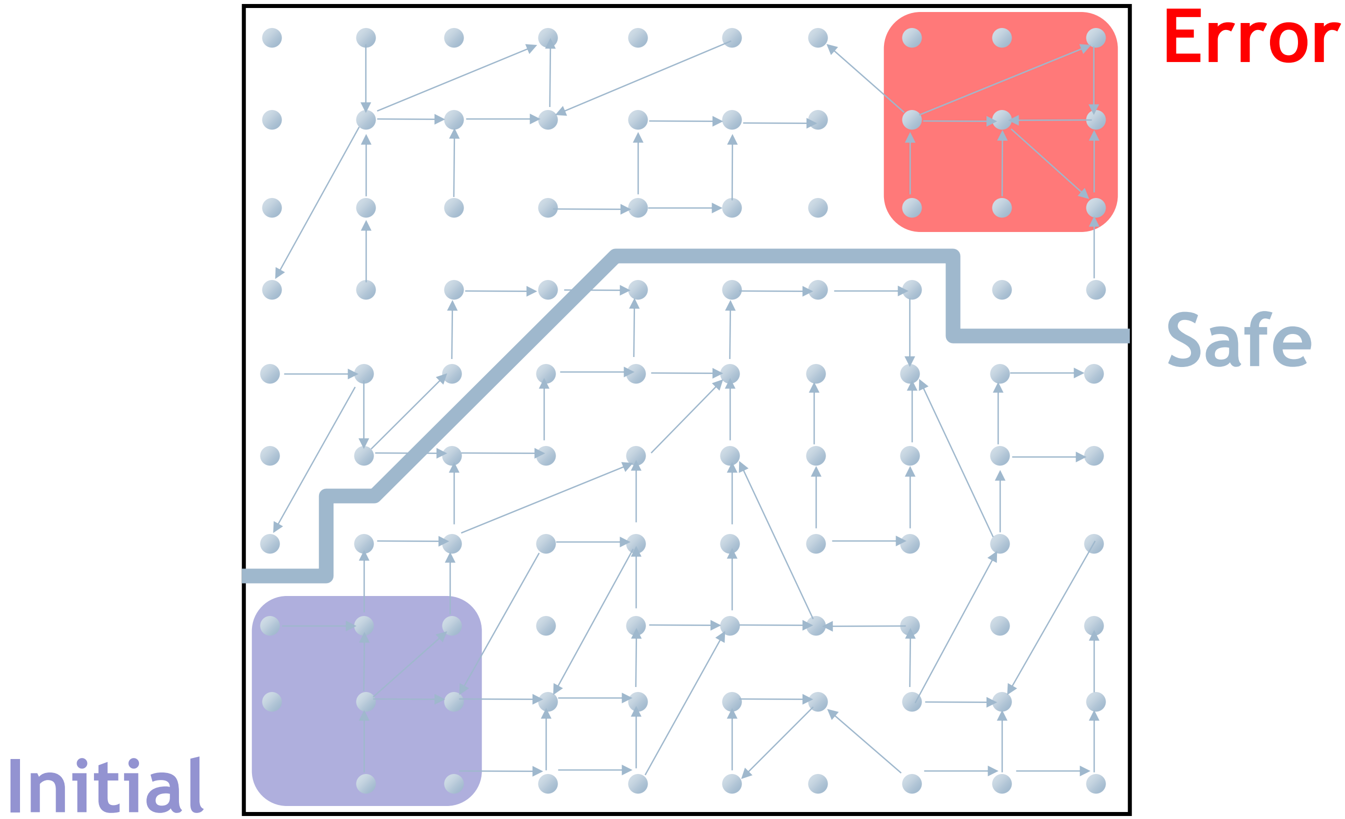
pc	$\mapsto$ 3
lock	$\mapsto$ ●
old	$\mapsto$ 5
new	$\mapsto$ 5
q	$\mapsto$ 0x133a

```
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



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

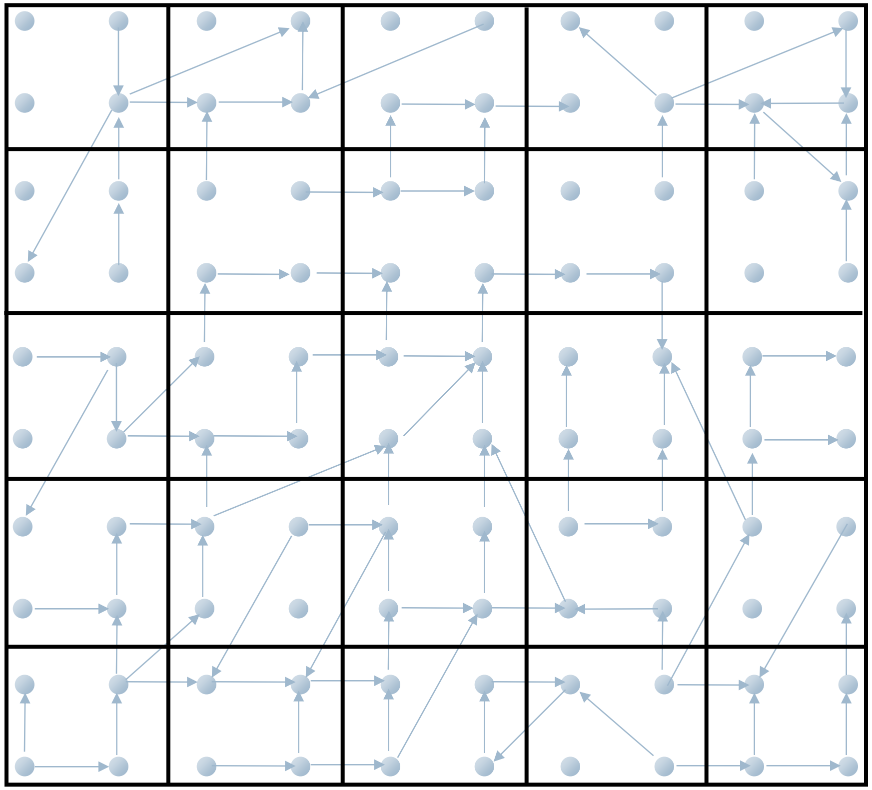
Problem: Infinite state graph

Solution: Set of states is a logical formula

Representing States as Formulas

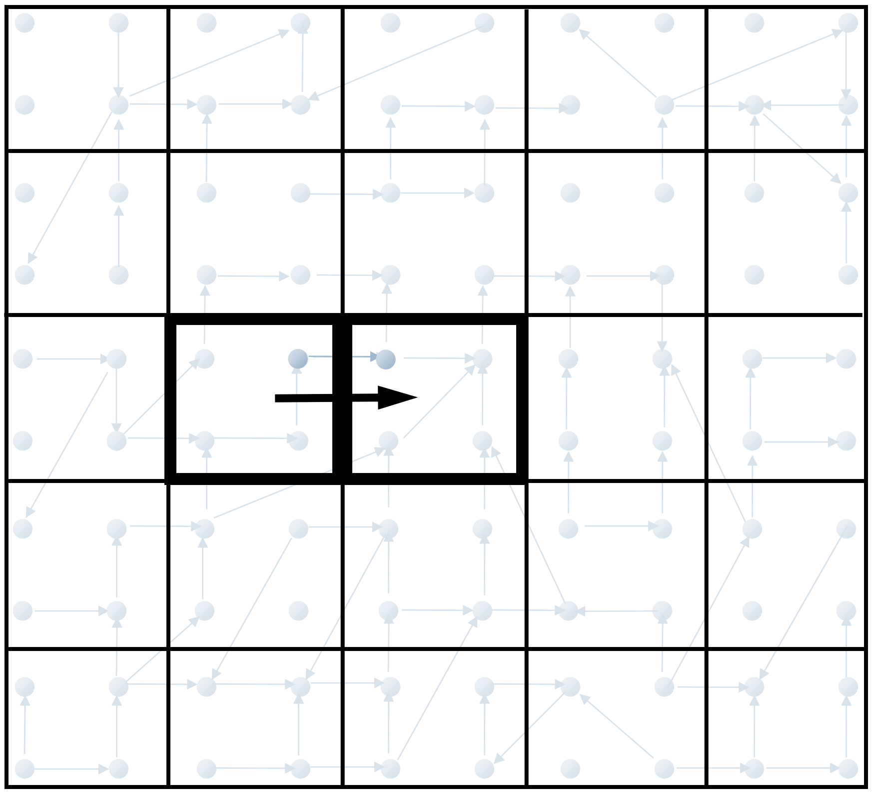
States	Formulas
$[F]$ states satisfying F $\{s \mid s \vDash F\}$	F FO formula over prog.vars
$[F_1] \cap [F_2]$	$F_1 \wedge F_2$
$[F_1] \cup [F_2]$	$F_1 \vee F_2$
$[F]$	$\neg F$
$[F_1] \subseteq [F_2]$	$F_1 \rightarrow F_2$ i.e. $F_1 \wedge \neg F_2$ unsatisfiable

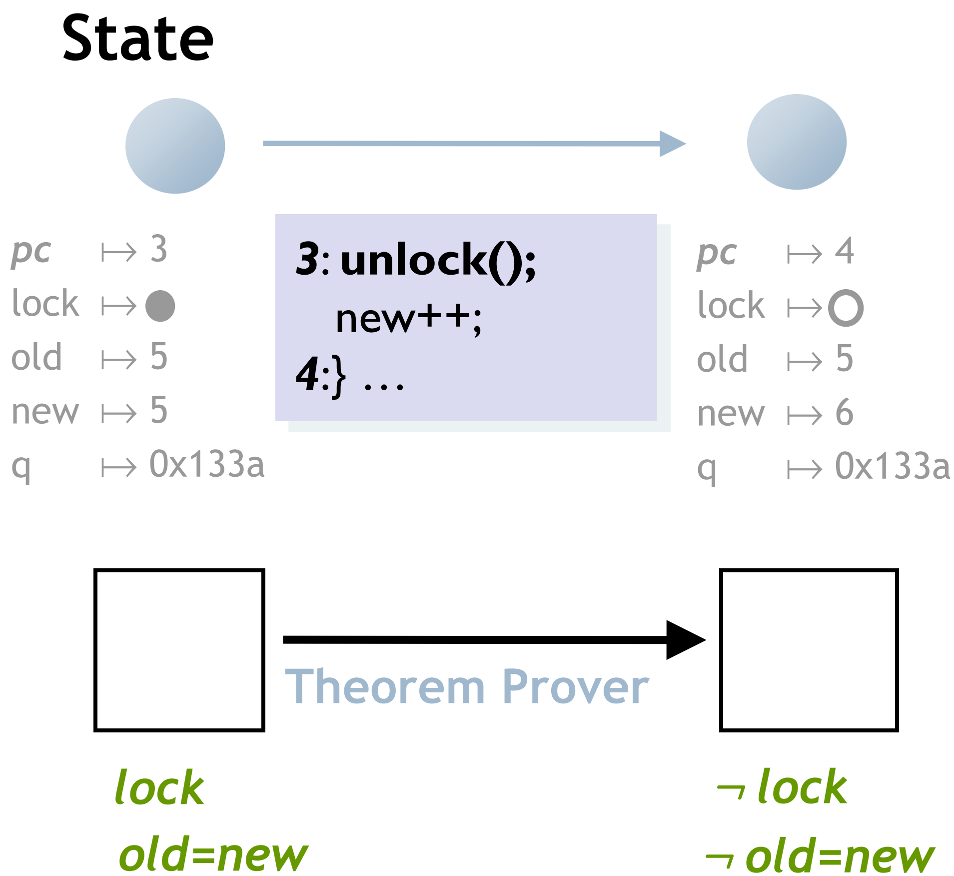
Idea: Predicate Abstraction



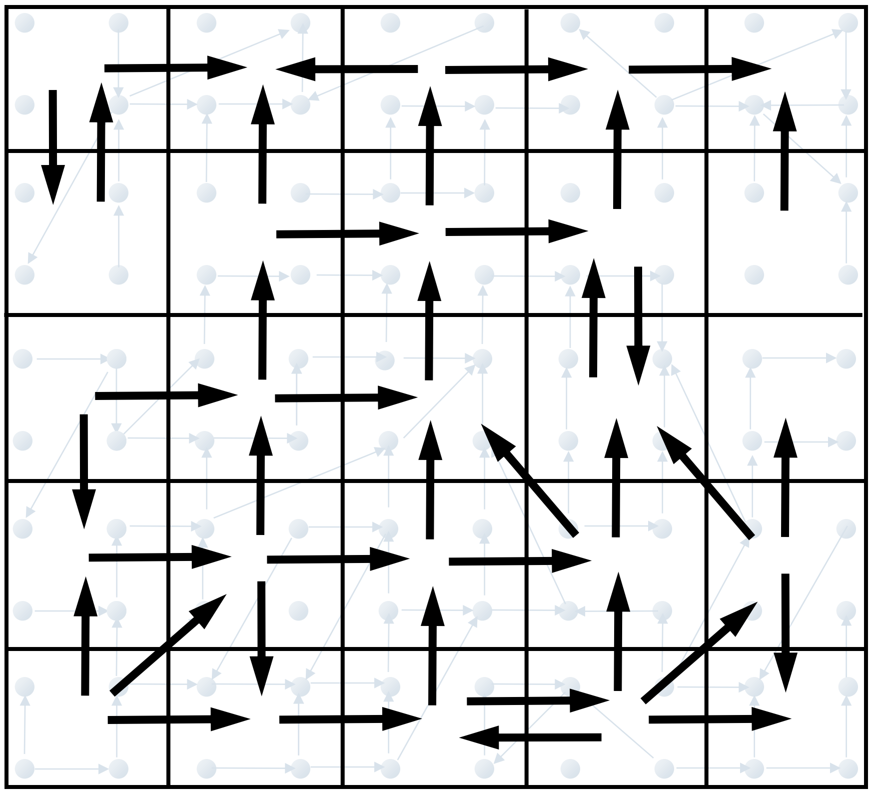
- Predicates on program state:
  `lock`
  `old = new`
- States satisfying same predicates are equivalent
  - Merged into one abstract state
- #abstract states is finite

Abstract States and Transitions

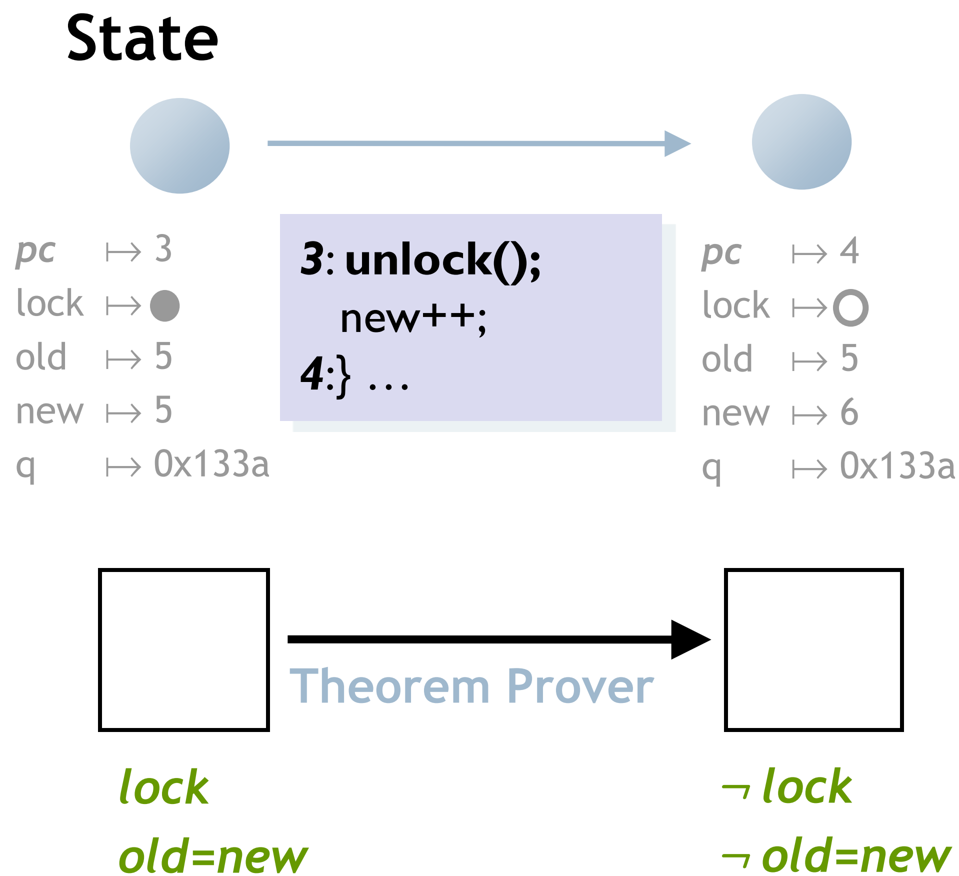




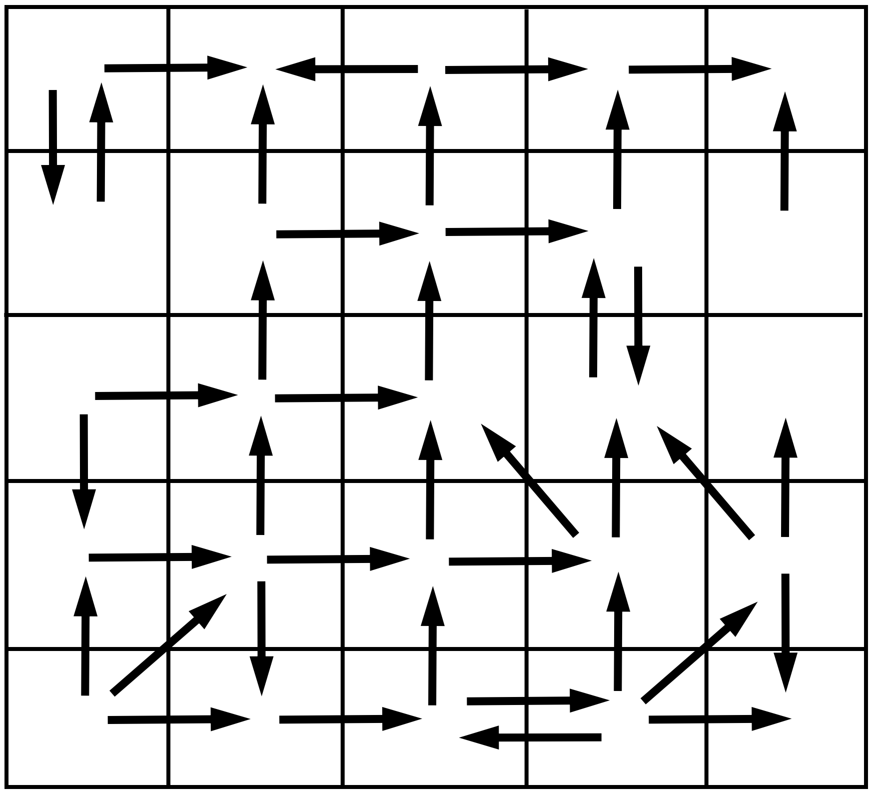
Abstraction

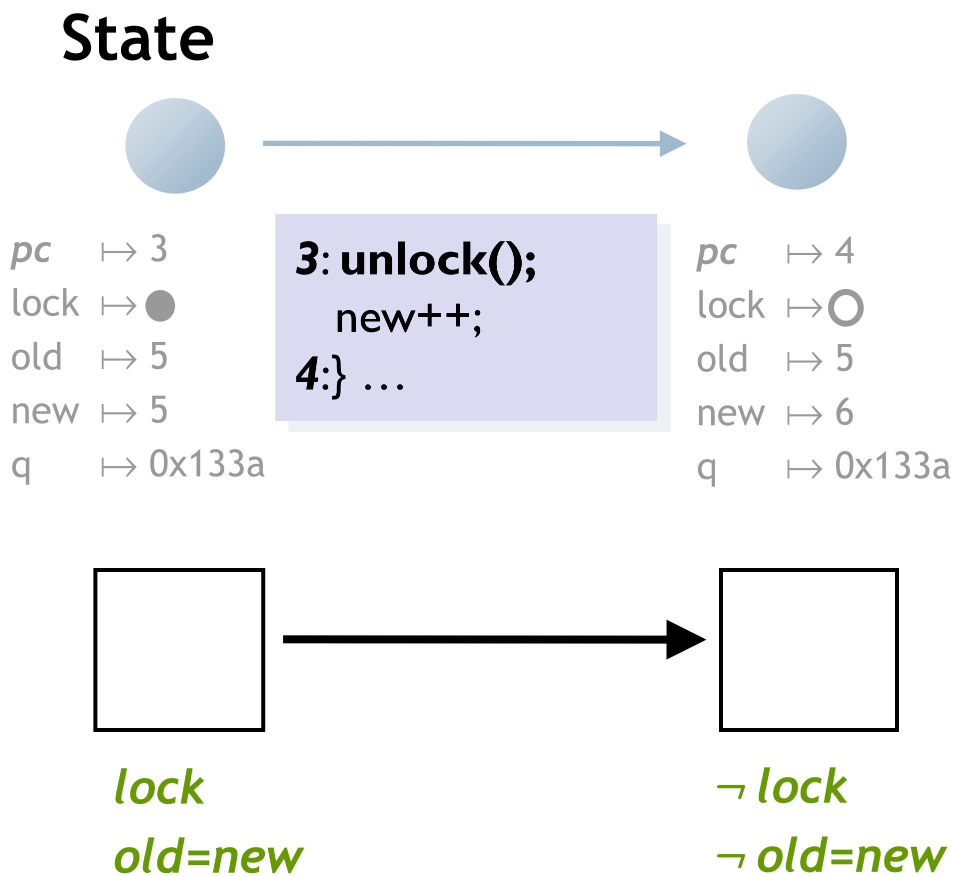


Existential Lifting

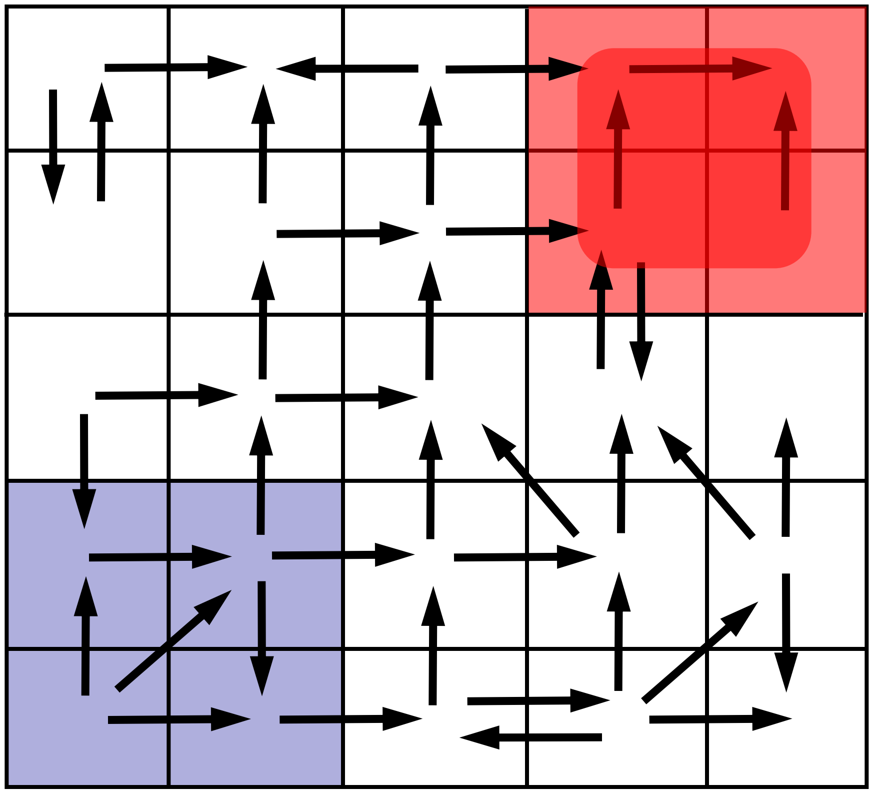


Abstraction





Analyze Abstraction



Analyzing finite graph

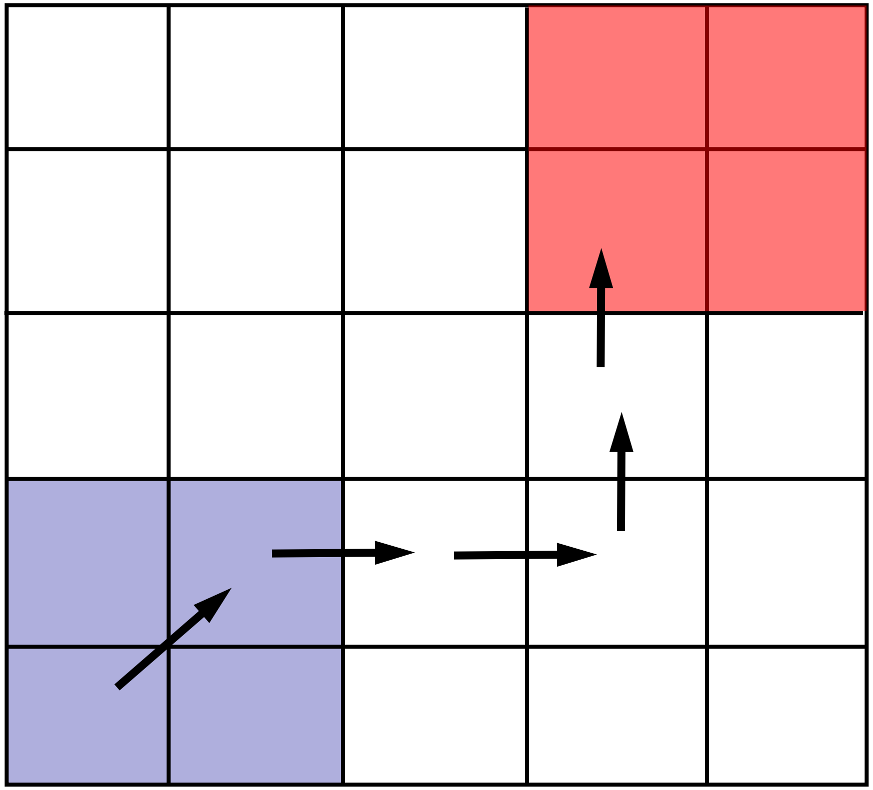
Over-approximate:

- Safe means that system is safe
- No false negatives

Problem:

Spurious counterexamples

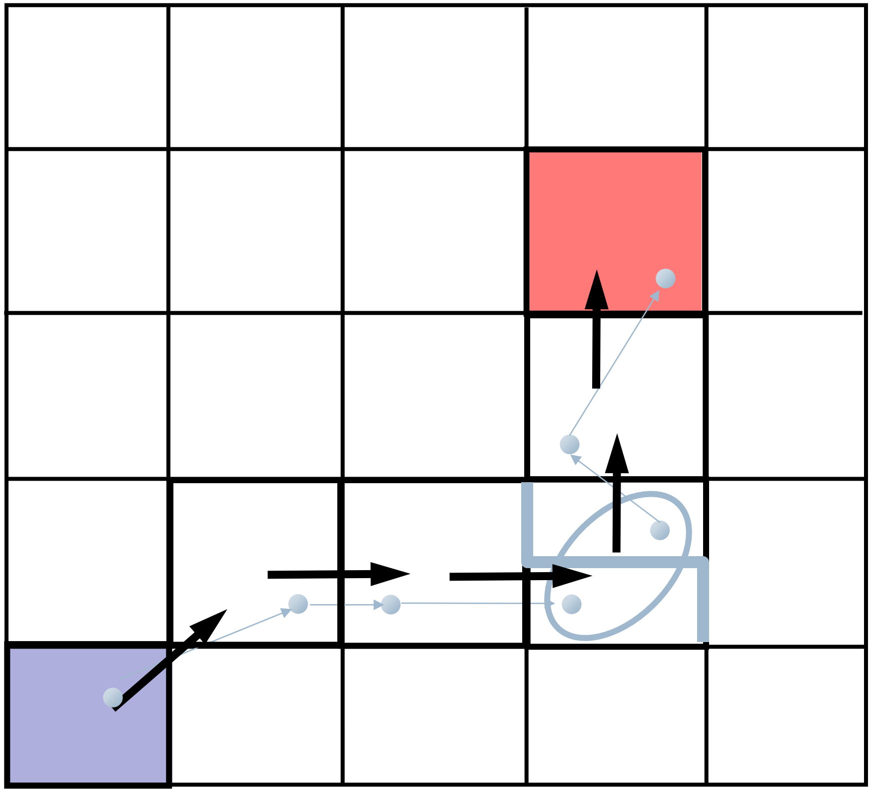
Idea: Counterex.-Guided Refinement



Solution:

Use spurious counterexamples to refine abstraction

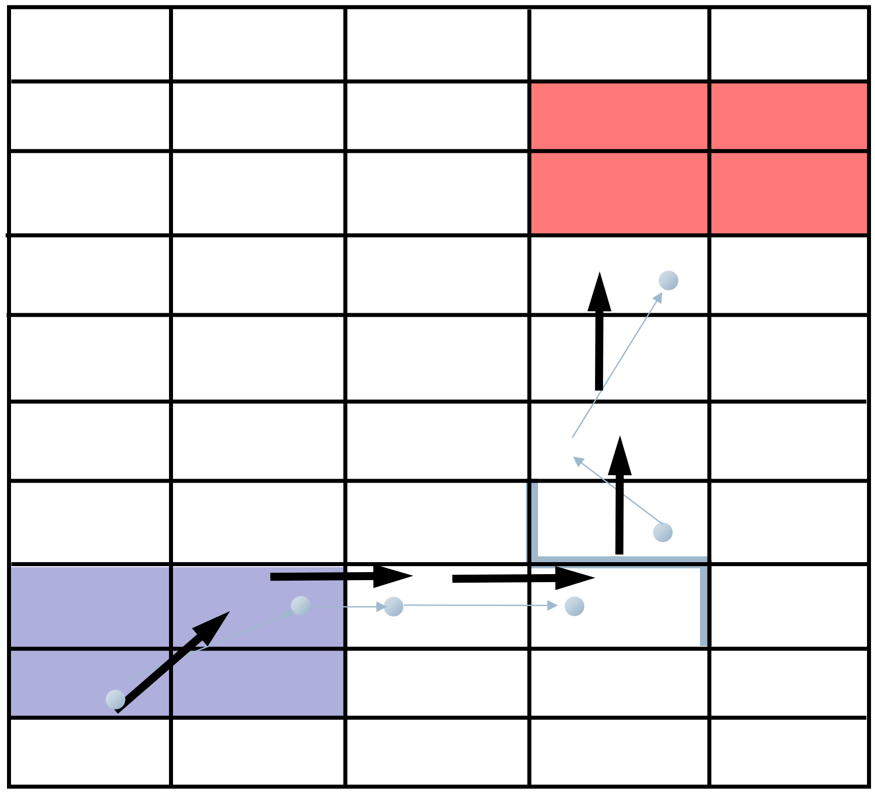
Idea: Counterex.-Guided Refinement



Solution:
Use spurious counterexamples to refine abstraction

1. Add predicates to distinguish states across cut

Iterative Abstraction Refinement



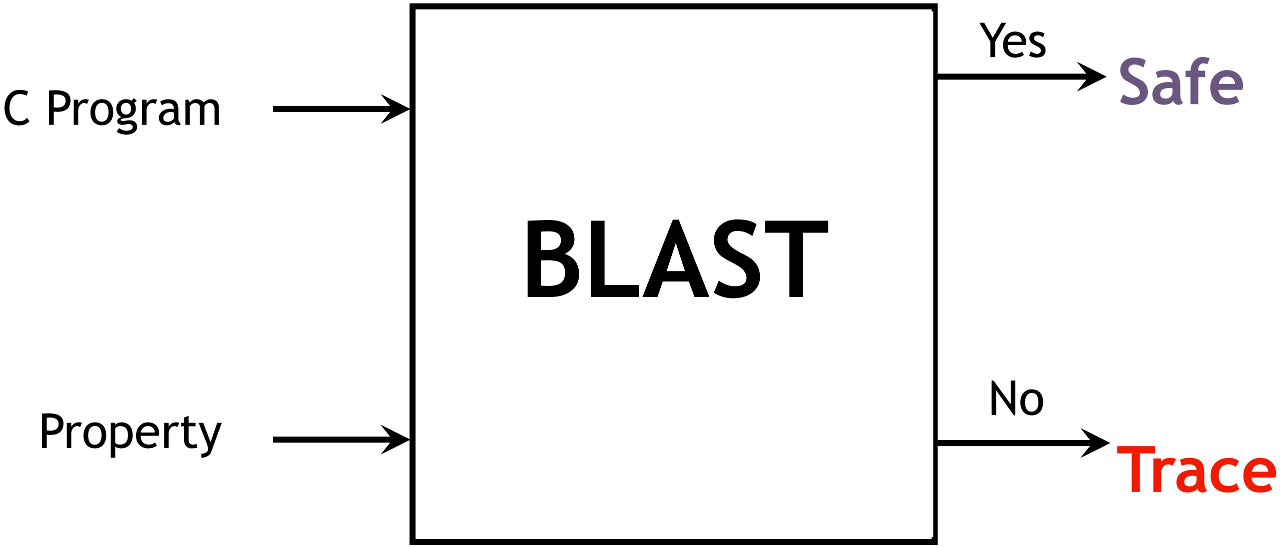
Solution:
Use spurious counterexamples to refine abstraction

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

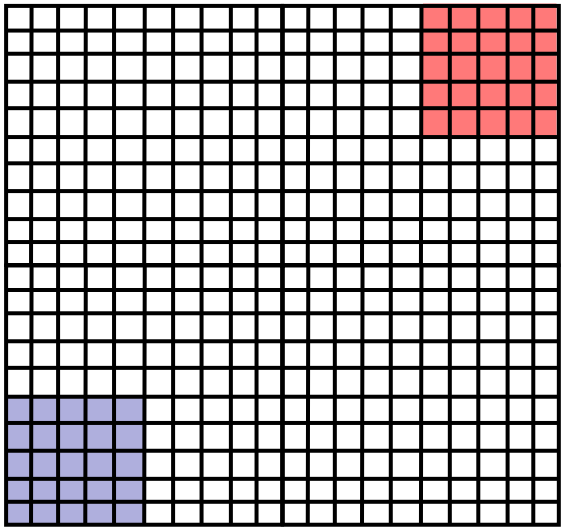
Predicate-Abstraction-Based Tools

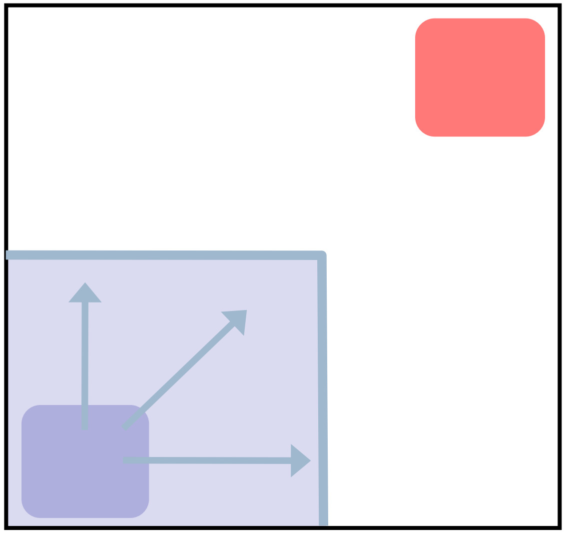
- SLAM
- BLAST
- SATABS
- CPAchecker
- IMPACT
- VCEGAR

Lazy Abstraction



Problem: Abstraction is Expensive





Reachable

Problem

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

Exponential Thm. Prover queries

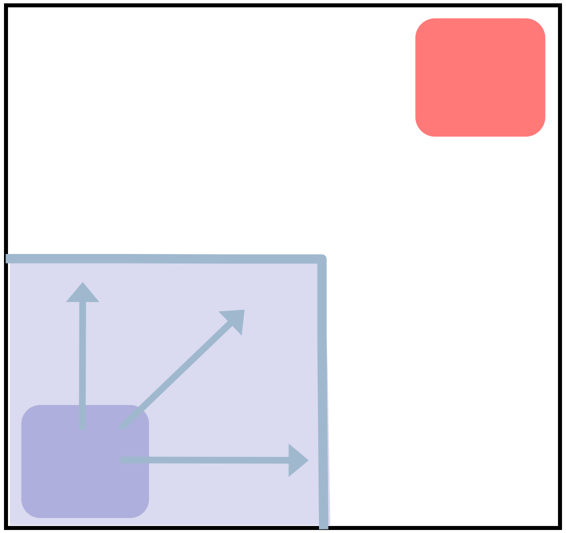
Observe

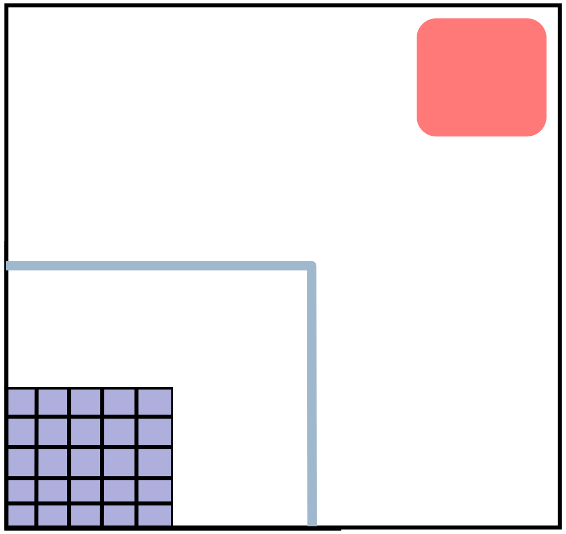
Fraction of state space reachable

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

#Reach ~ 1000's

Solution1: Only Abstract Reachable States





Problem

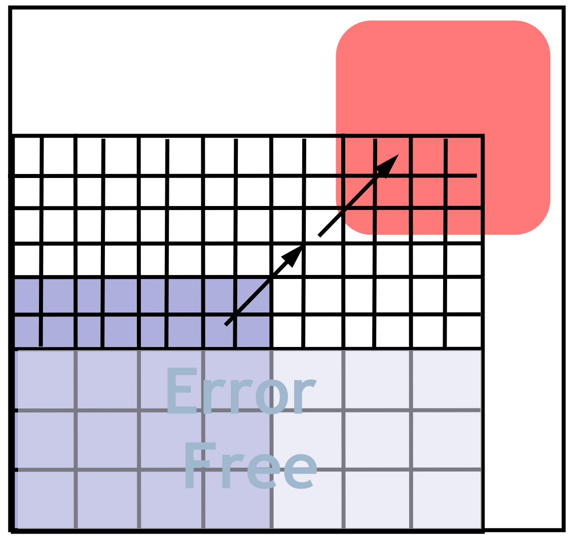
#abstract states = $2^{\#predicates}$

Exponential Thm. Prover queries

Solution

Build abstraction during search

Solution2: Don't Refine Error-Free Regions



Problem

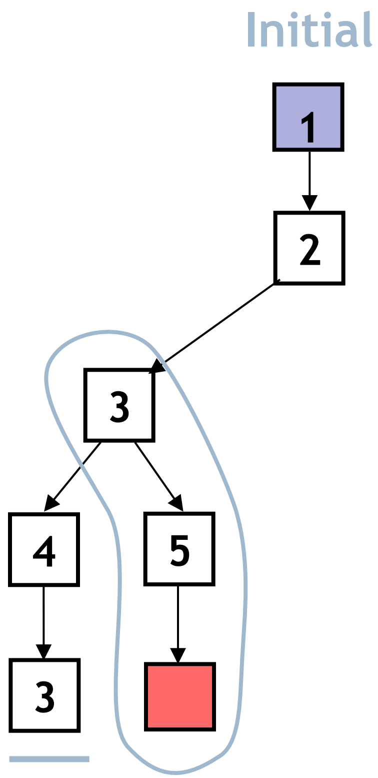
#abstract states = $2^{\#predicates}$

Exponential Thm. Prover queries

Solution

Don't refine error-free regions

Key Idea: Reachability Tree



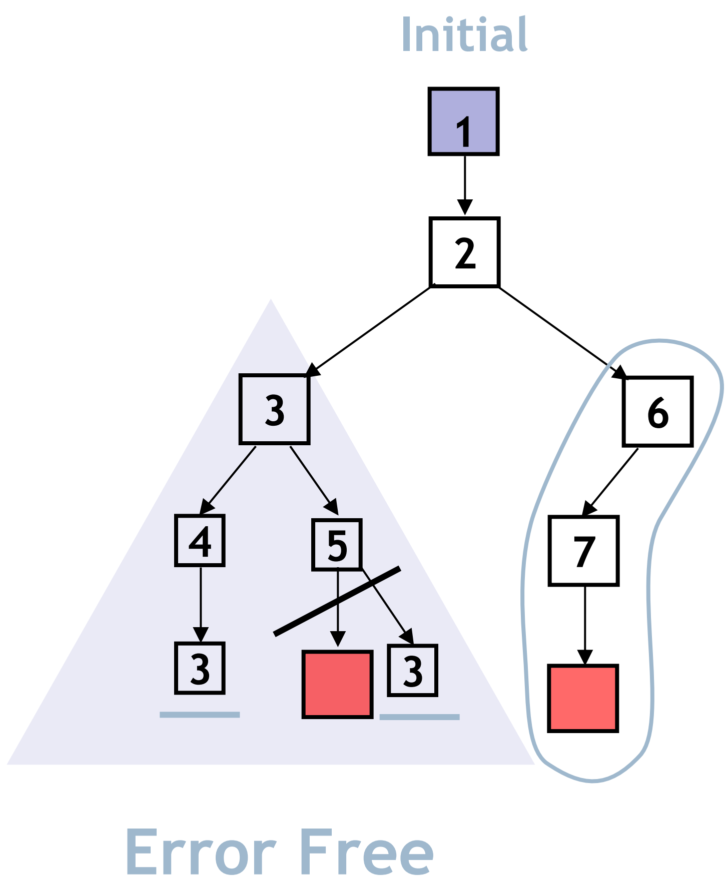
Unroll Abstraction

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



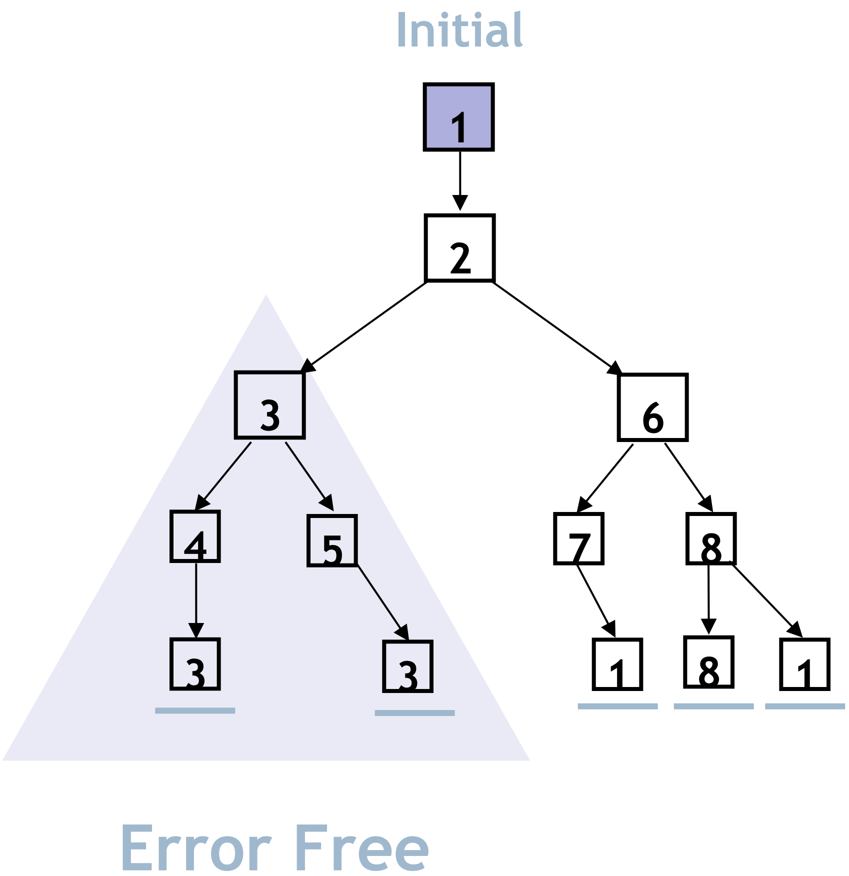
Unroll Abstraction

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.

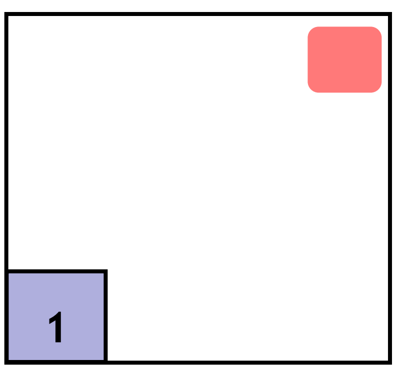
SAFE

S1: Only Abstract Reachable States

S2: Don’t refine error-free regions

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 ();
}
```

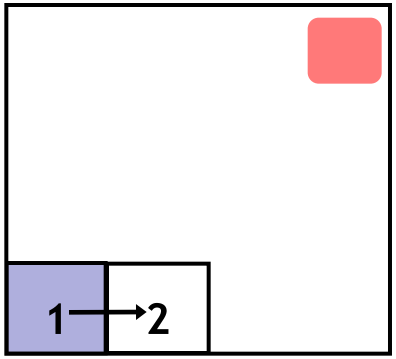


Predicates: *LOCK*

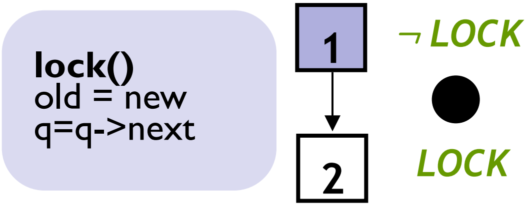
Reachability Tree

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 ();
}
```



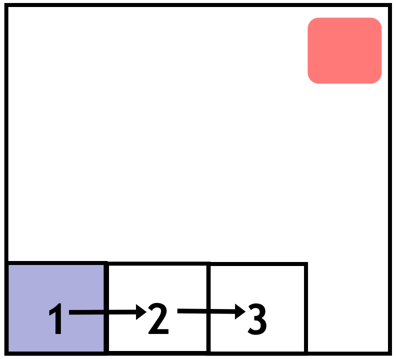
Predicates: *LOCK*



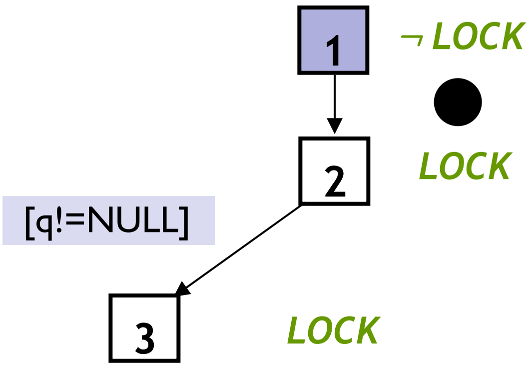
Reachability Tree

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 ();
}
```



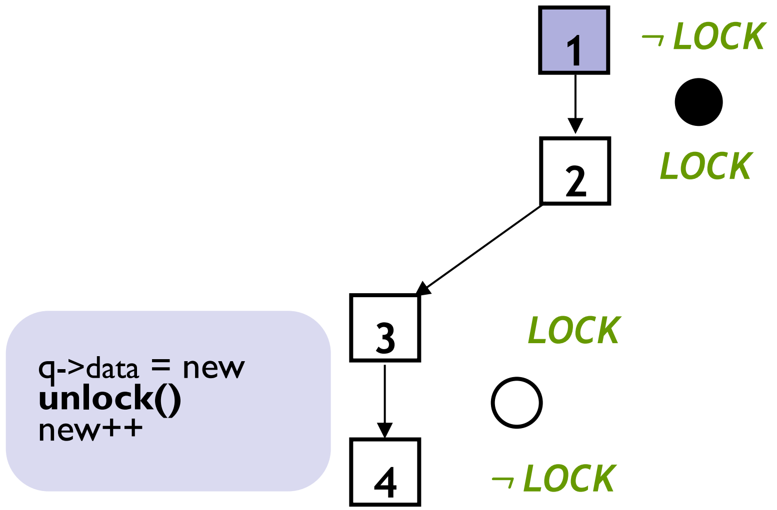
Predicates: *LOCK*

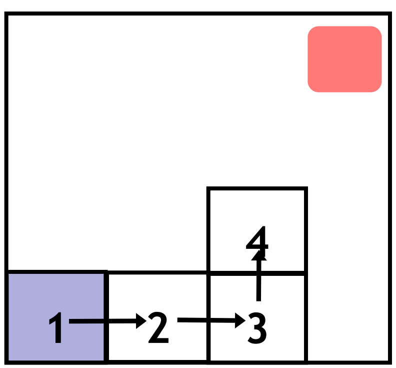


Reachability Tree

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 ();
}
```



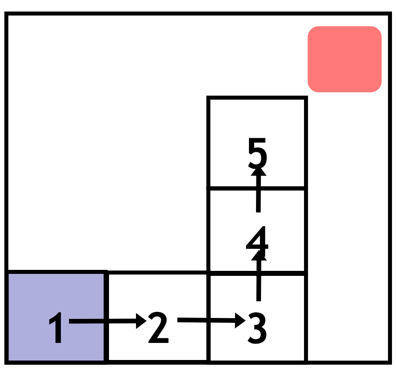


Predicates: *LOCK*

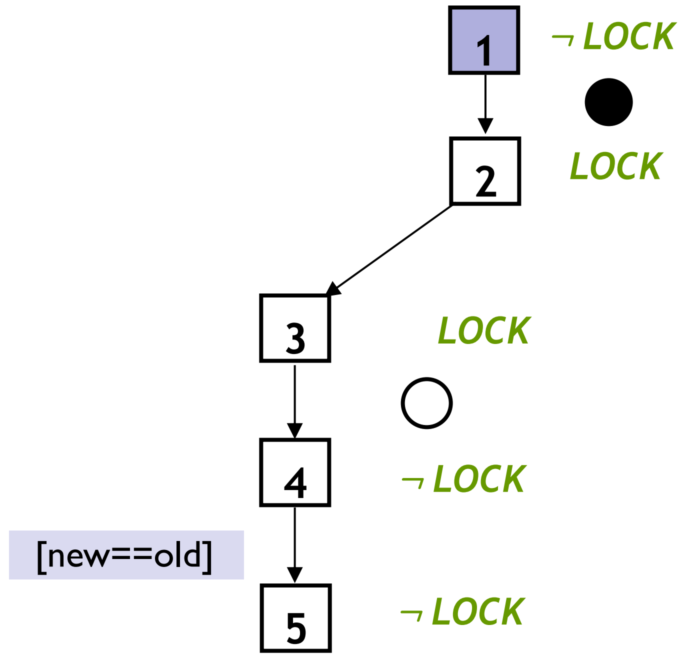
Reachability Tree

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 ();
}
```



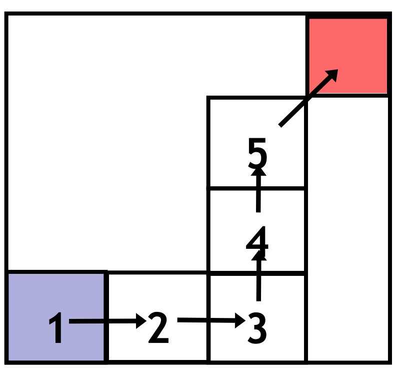
Predicates: *LOCK*



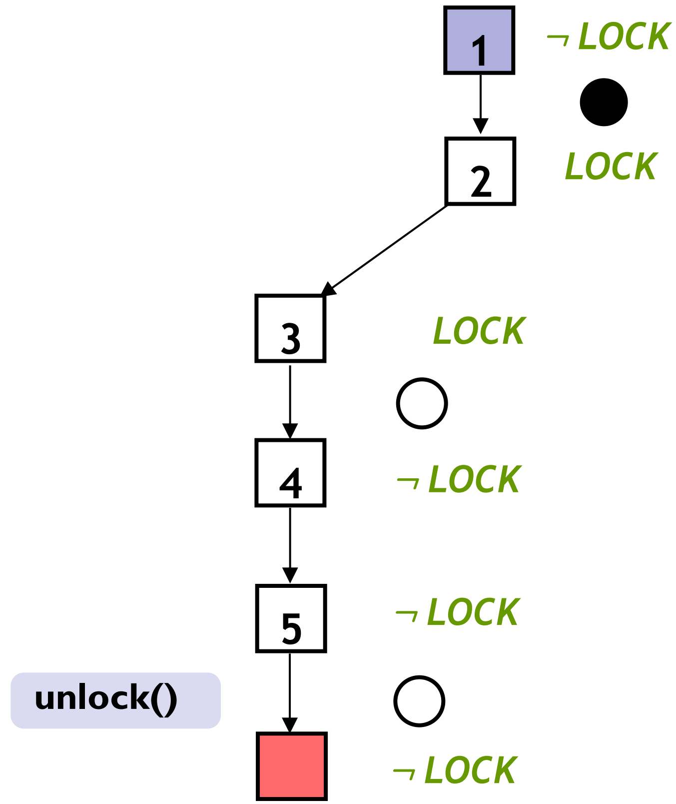
Reachability Tree

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 ();
}
```



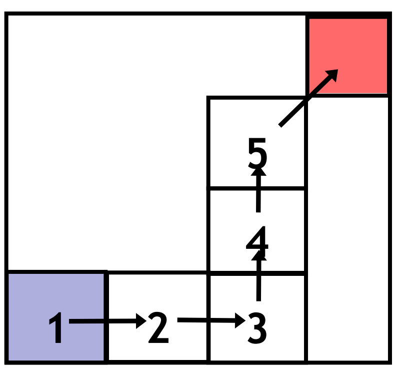
Predicates: *LOCK*



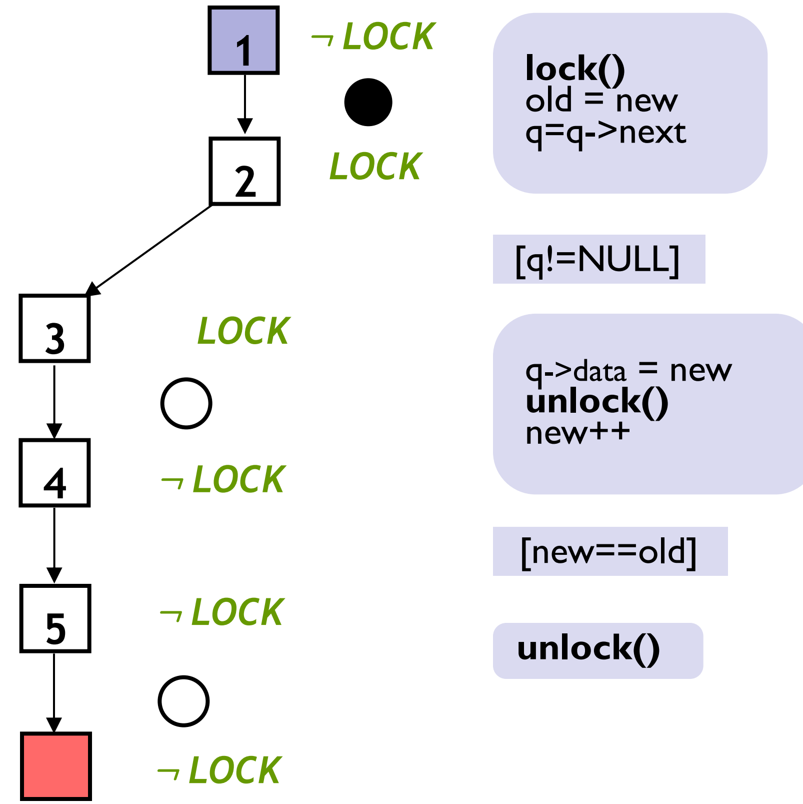
Reachability Tree

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 ();
}
```



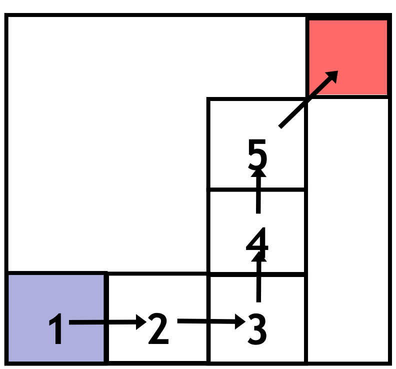
Predicates: *LOCK*



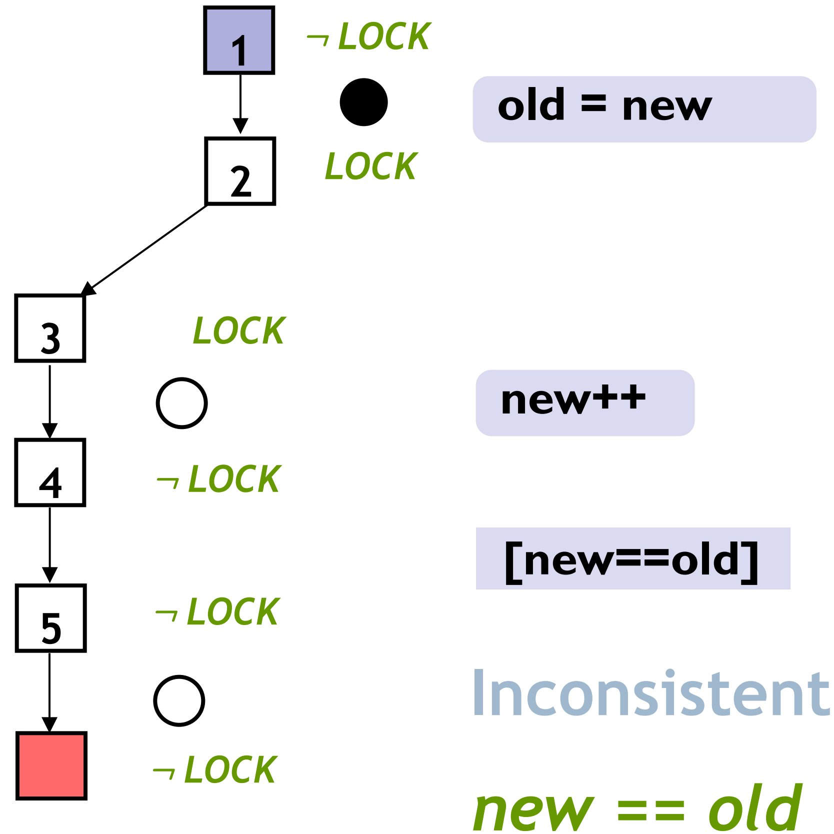
Reachability Tree

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 ();
}
```



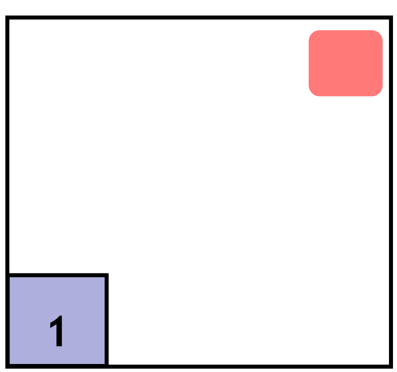
Predicates: *LOCK*



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 ();
}
```

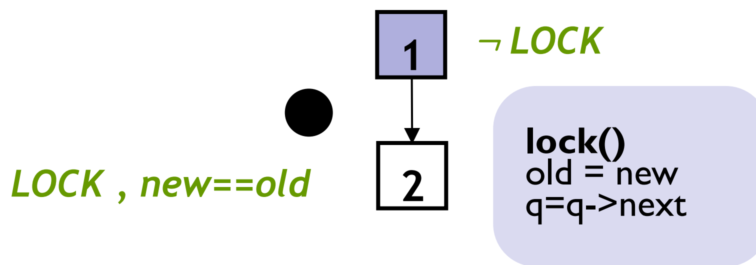


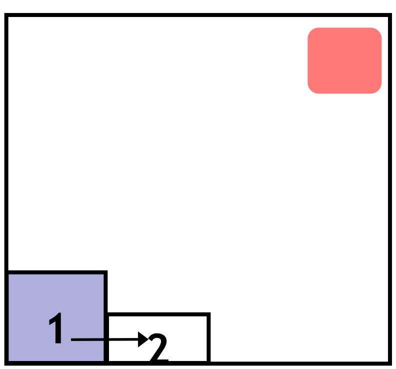
Predicates: *LOCK, new==old*

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 ();
}
```



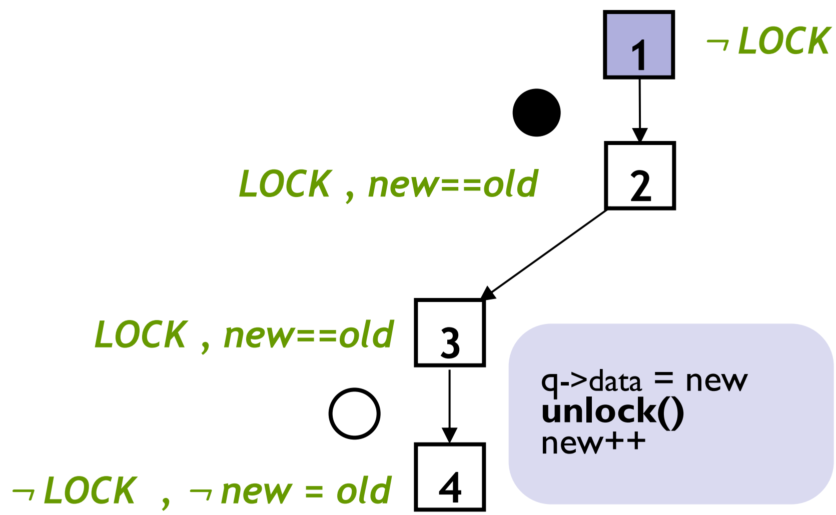


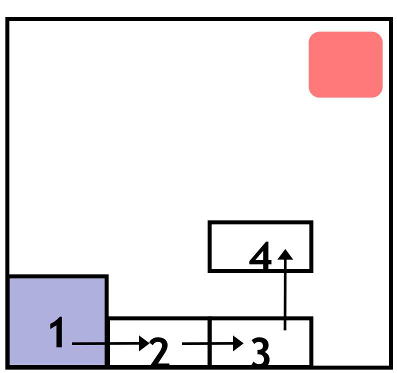
Predicates: *LOCK, new==old*

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 ();
}
```



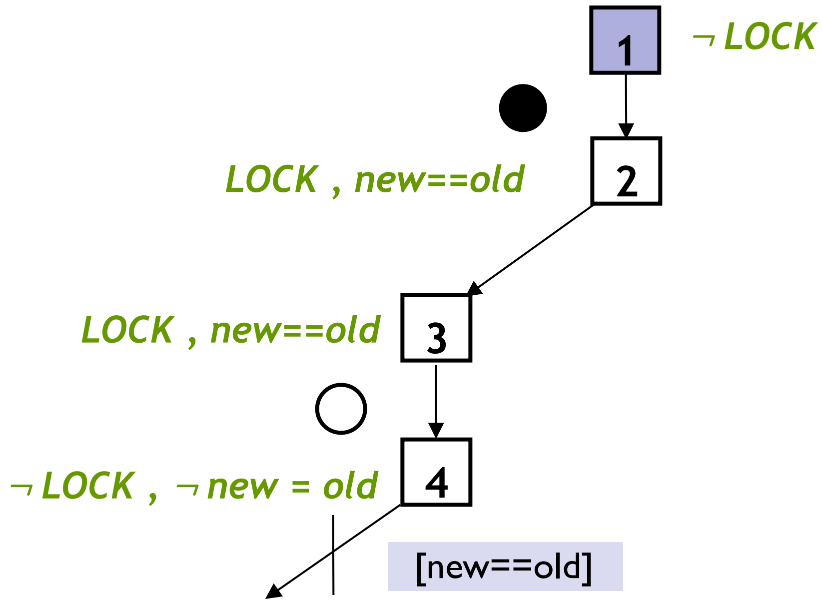


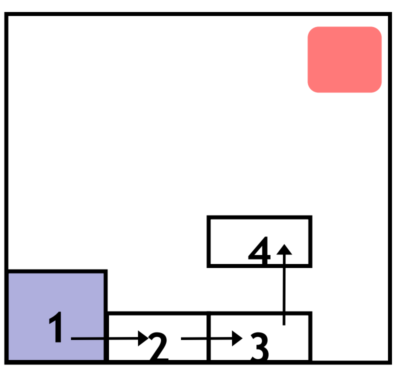
Predicates: *LOCK, new==old*

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 ();
}
```



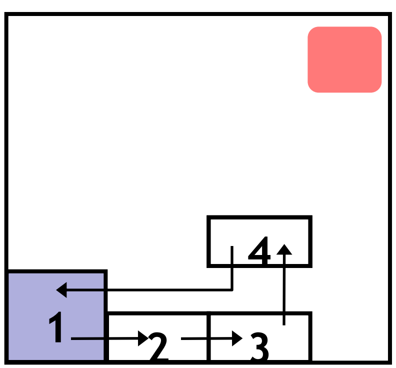


Predicates: *LOCK, new==old*

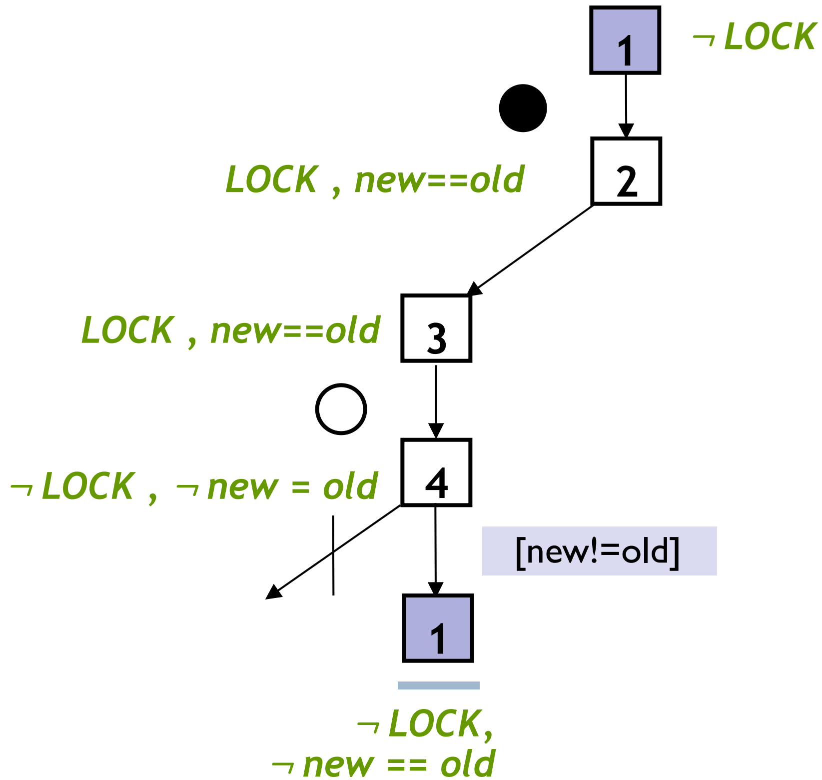
Reachability Tree

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 ();
}
```



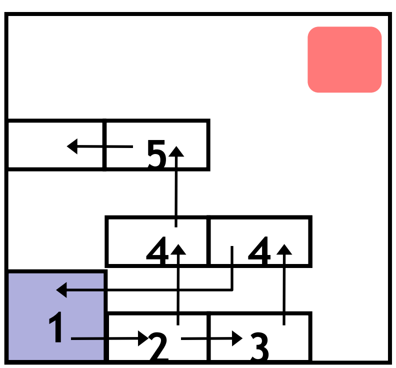
Predicates: *LOCK, new==old*



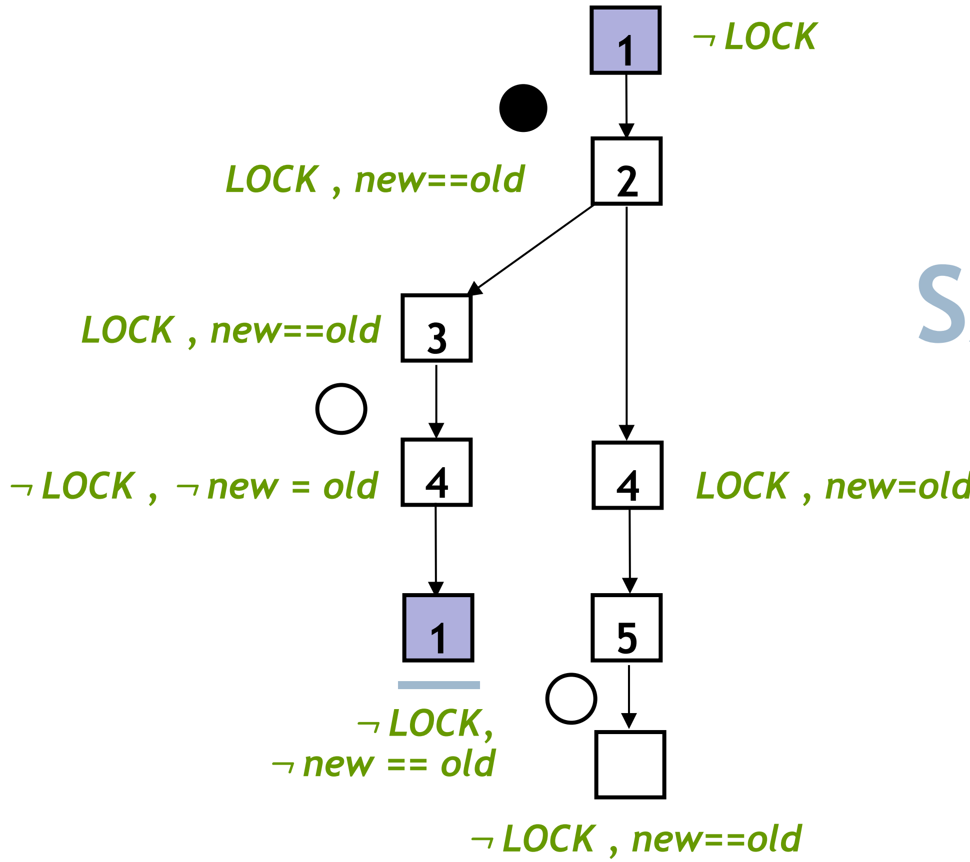
Reachability Tree

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 ();
}
```



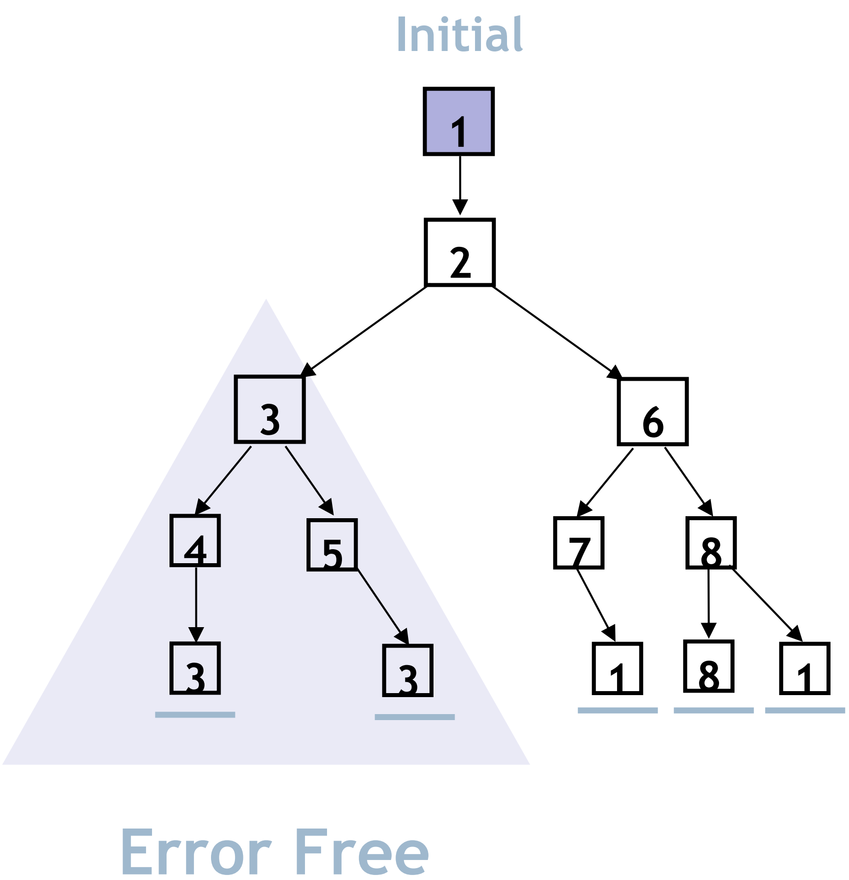
Predicates: *LOCK, new==old*



SAFE

Reachability Tree

Key Idea: Reachability Tree



Unroll

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

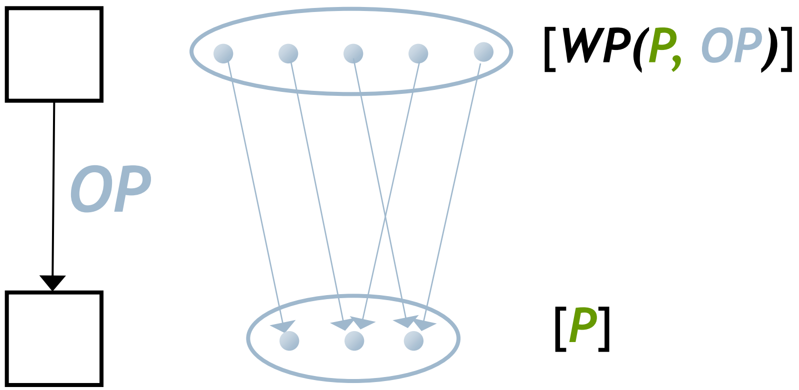
Weakest Preconditions

WP(P,OP)

Weakest formula *P'* s.t.

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

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



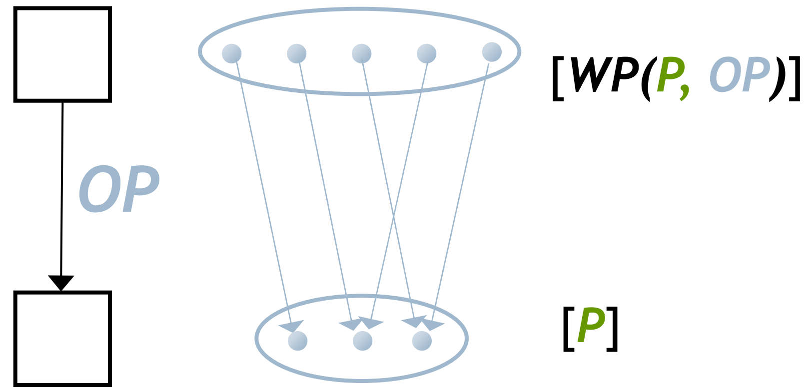
Weakest Preconditions

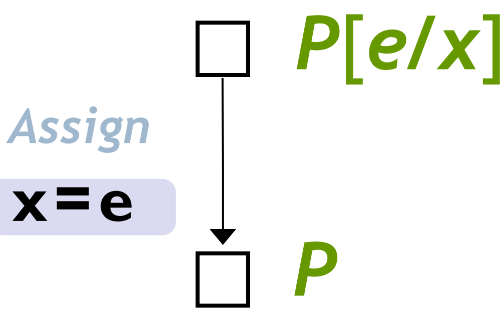
WP(P,OP)

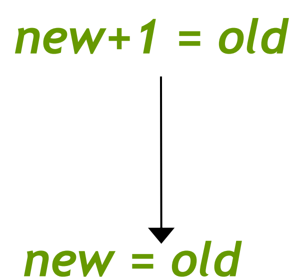
Weakest formula *P'* s.t.

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

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







new=new+1

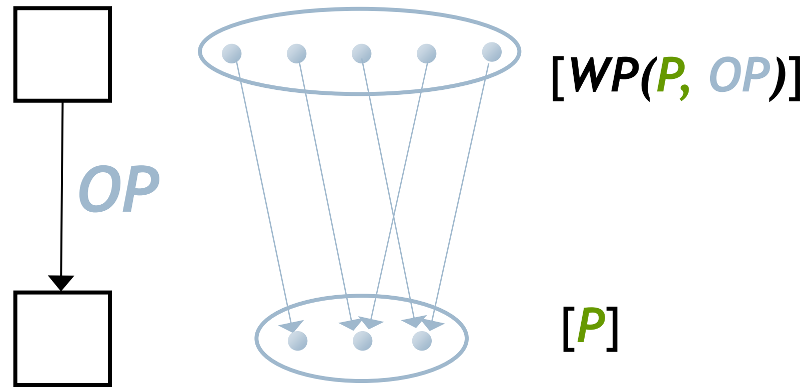
Weakest Preconditions

WP(P,OP)

Weakest formula *P'* s.t.

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

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



$c \Rightarrow P$

Assume

[c]

P

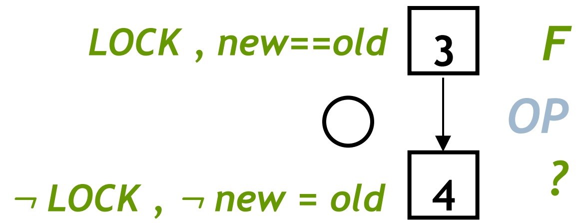
new=old $\Rightarrow$ *new=old*

[new=old]

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*

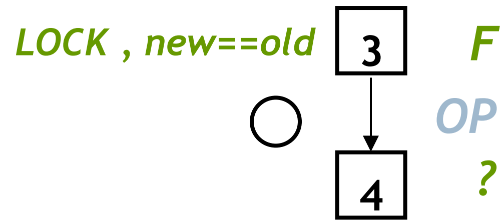
Q: When is *p* true after *OP* ?

- If ***WP(p, OP)*** is true before *OP* !
- We know *F* is true before *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

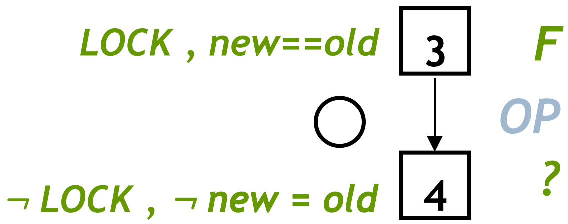
Q: When is p false <u>after</u> OP ?

- If $WP(\neg p, OP)$ is true <u>before</u> OP !
- We know F is true <u>before</u> 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* ?

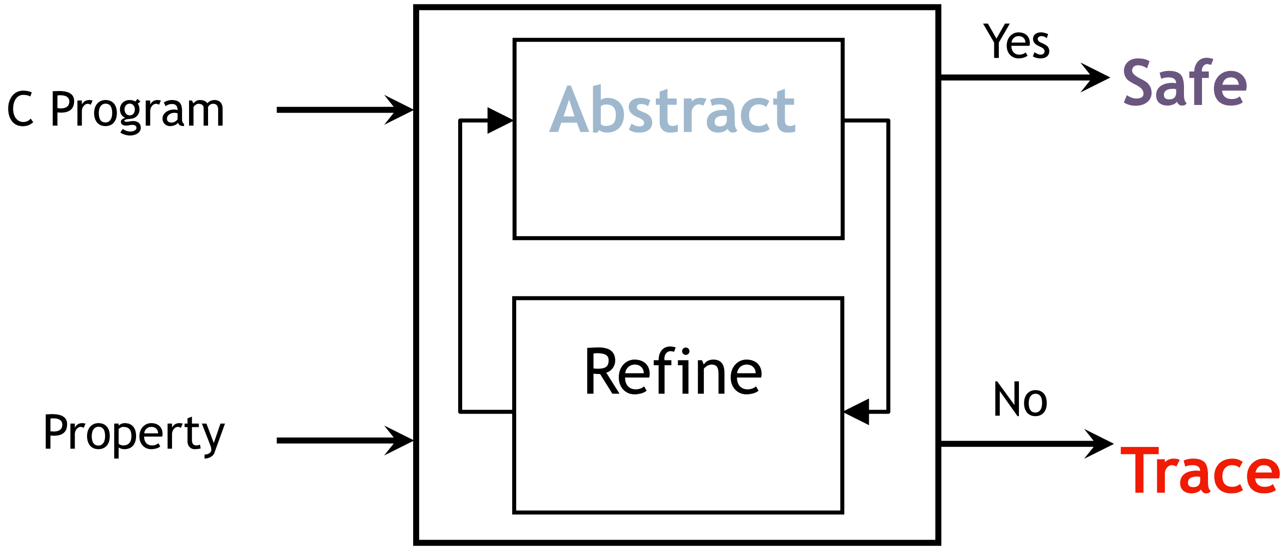
- If *WP*(¬ *p*, *OP*) is true before *OP* !
- We know *F* is true before *OP*
- Thm. Pvr. Query: $F \Rightarrow WP(\neg p, OP)$

Predicate: *new==old*

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

False ? $(LOCK \wedge new==old) \Rightarrow (new + 1 \neq old)$ YES

Lazy Abstraction



Problem: Abstraction is Expensive

Solution: 1. Abstract reachable states,
2. Avoid refining error-free regions

Key Idea: Reachability Tree