Model Checking, Theorem Proving, and Abstract Interpretation:

The Convergence of Formal Verification Technologies

Tom Henzinger

EPFL

Three Verification Communities

- Model checking:

  -automatic, but inefficient (does not scale)
  -successful in hardware verification

- Theorem proving:

  -precise, but interactive (requires user intervention)
  -much recent progress in decision procedures

- Static analysis:

  -efficient, but imprecise (many false positives)
  -standard in compilers

Convergence

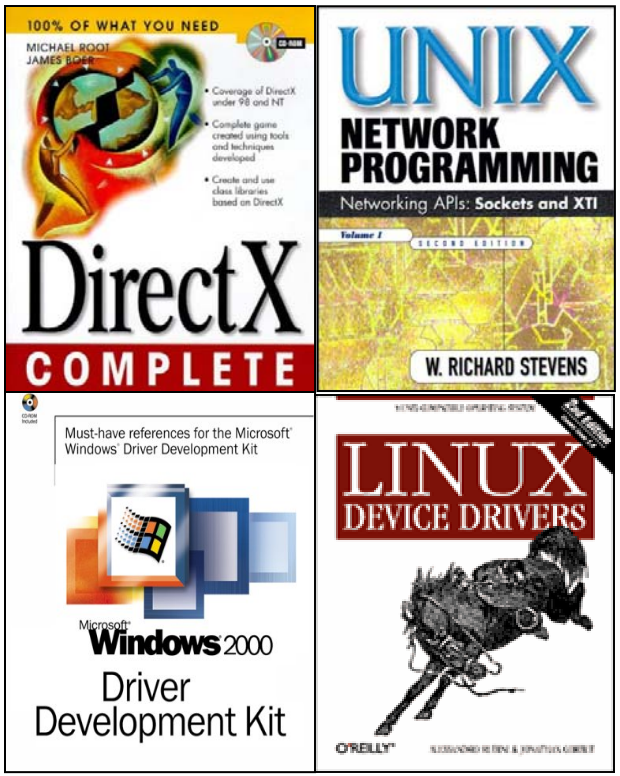
- Not just interaction of model checkers, theorem provers, and static analyzers via a shared code database
- But deep integration of the three technologies within a single tool
- Is it a model checker exploring states?
  A theorem prover manipulating formulas?
  A static analyzer interpreting an abstract program?

All of the above !

Property Checking

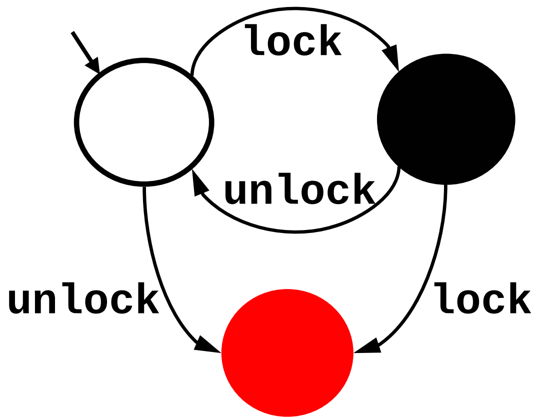
- Programmer gives partial specifications
  - Is there a complete spec for Word ? Emacs ?
  - Often implicit: code annotations, memory safety, race freedom
- Code checked for consistency with specification
- Different from program correctness
  - Partial specs of large programs rather than complete specs of small programs

Interface Usage Rules



- Rules in documentation
  - Order of operations and data access
  - Incomplete, unenforced, wordy
- Rule violations lead to bad behavior
  - System crash or deadlock
  - Unexpected exceptions

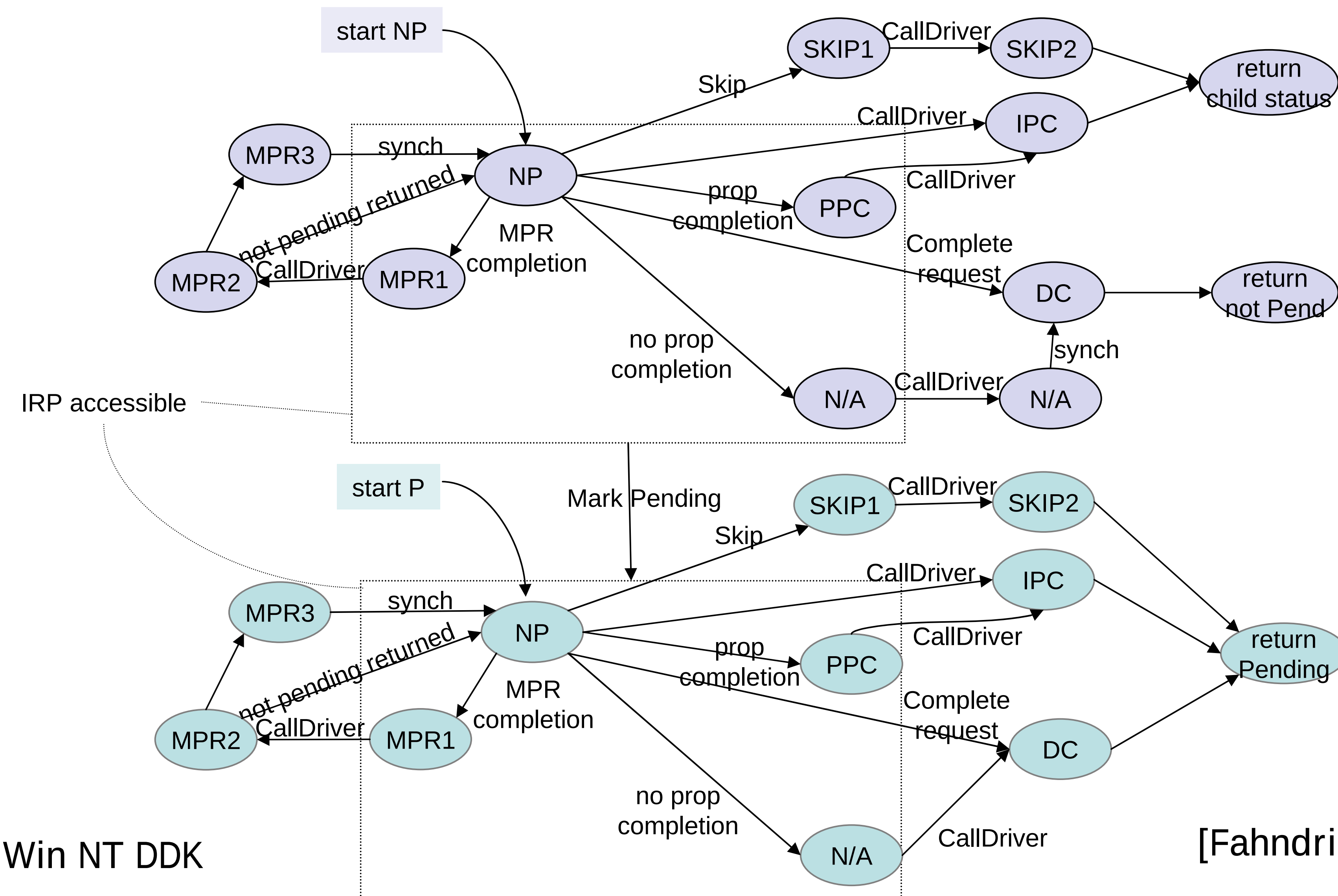
Property 1: Double Locking



"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: IRP Handler

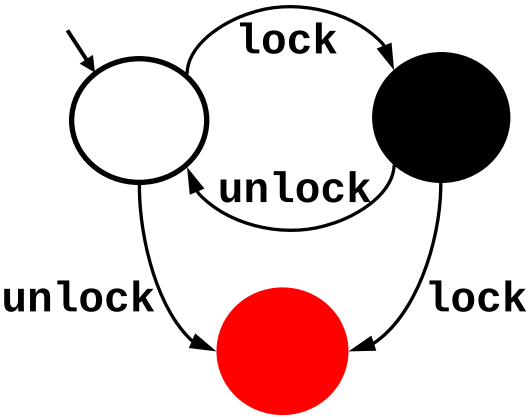


Win NT DDK

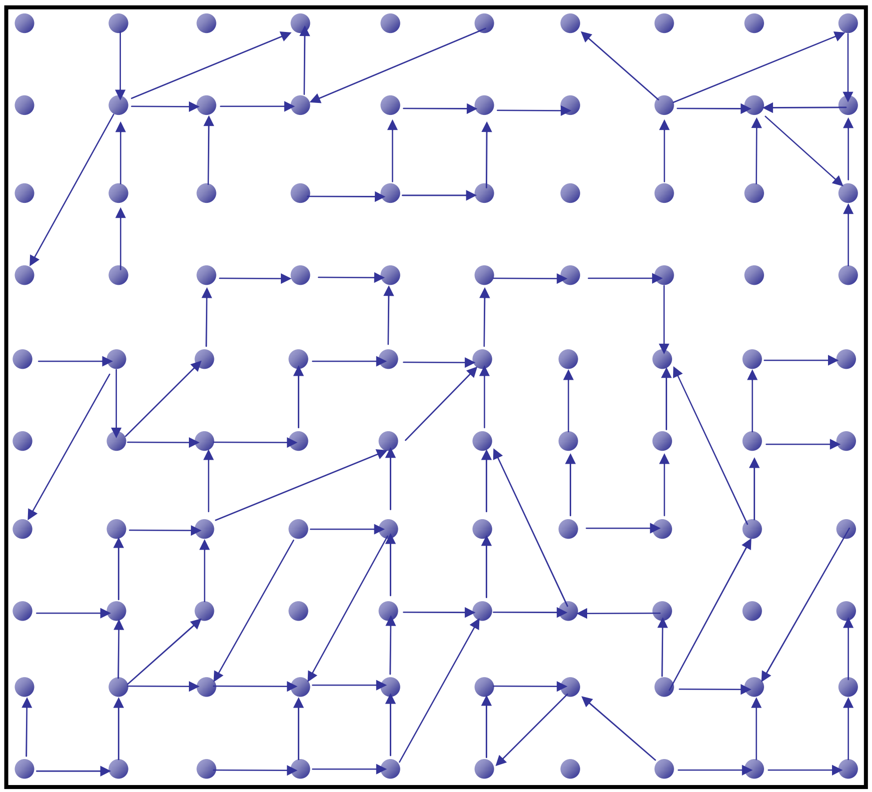
[Fahndrich]

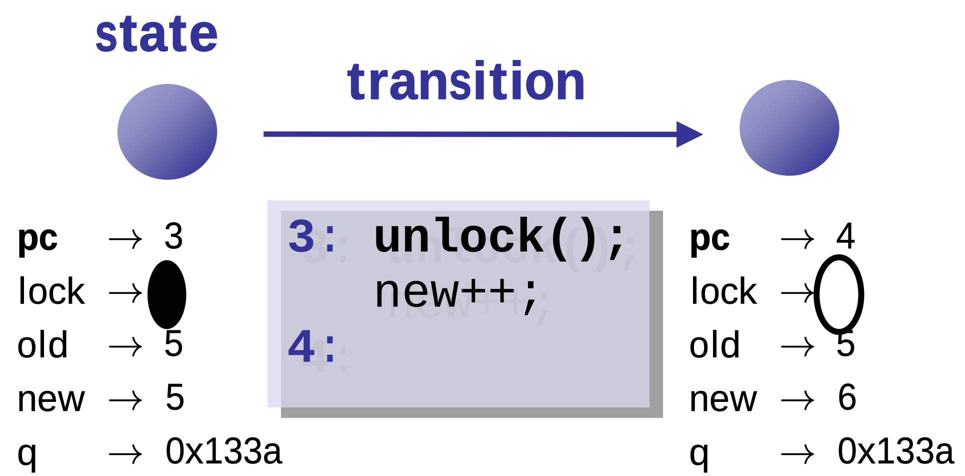
Locking Example

```
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 Model Checking View: Programs are State Transition Systems

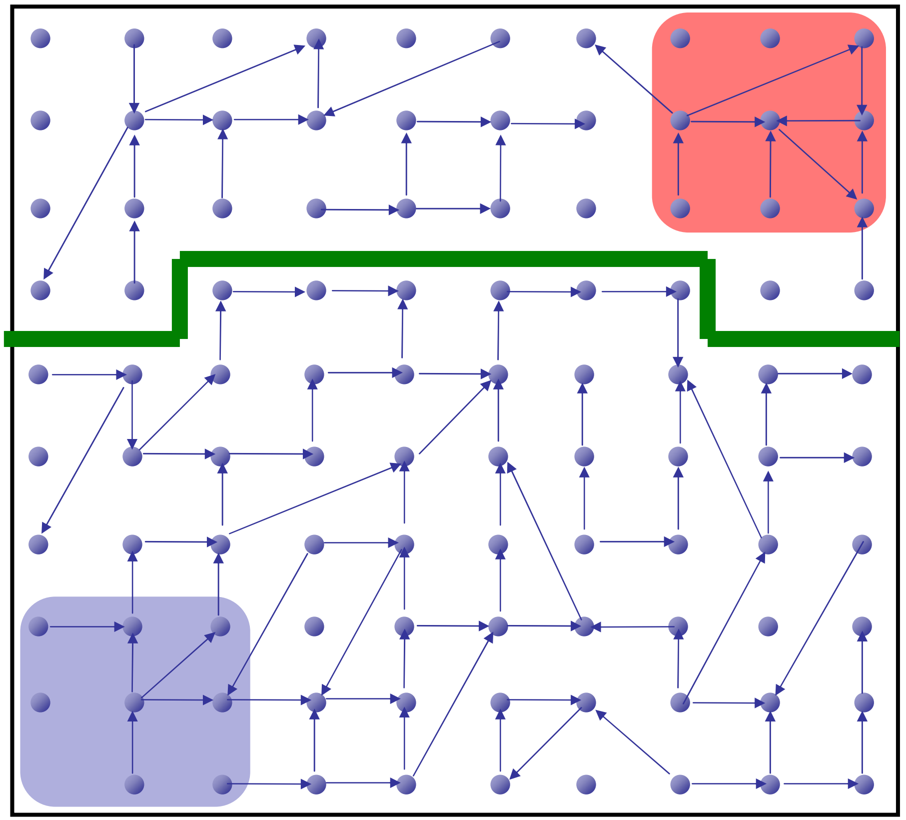





```
Example () {
1: do {
        lock();
        old = new;
        q = q->next;
2:      if (q != NULL) {
3:            q->data = new;
              unlock();
              new ++;
        }
4: } while (new != old);
5: unlock();
   return;}
```

[Clarke-Emerson, Queille-Sifakis]

The Safety Verification Problem



Error

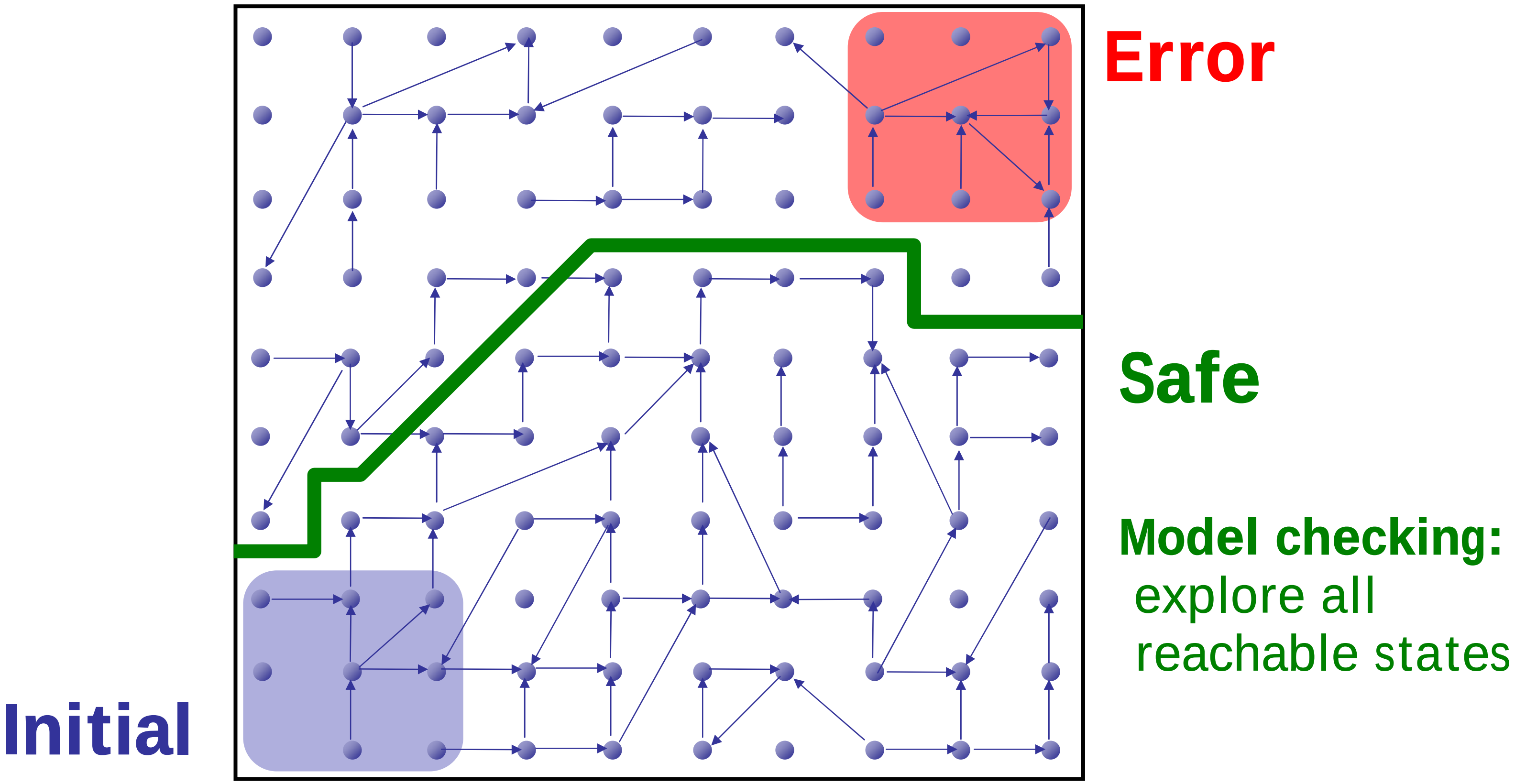
Safe

Theorem proving:
guess and verify
inductive invariant

Initial

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

The Safety Verification Problem

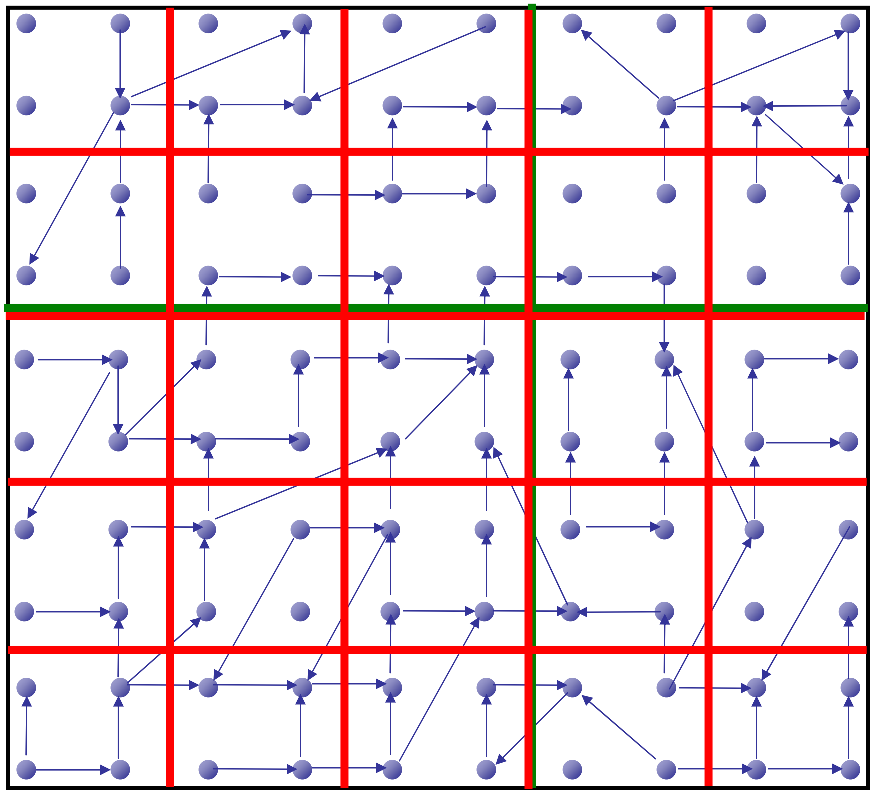


Is there a *path* from an initial state to an error state ?

Problem: infinite state graph

Solution: set of states $\simeq$ logical formula

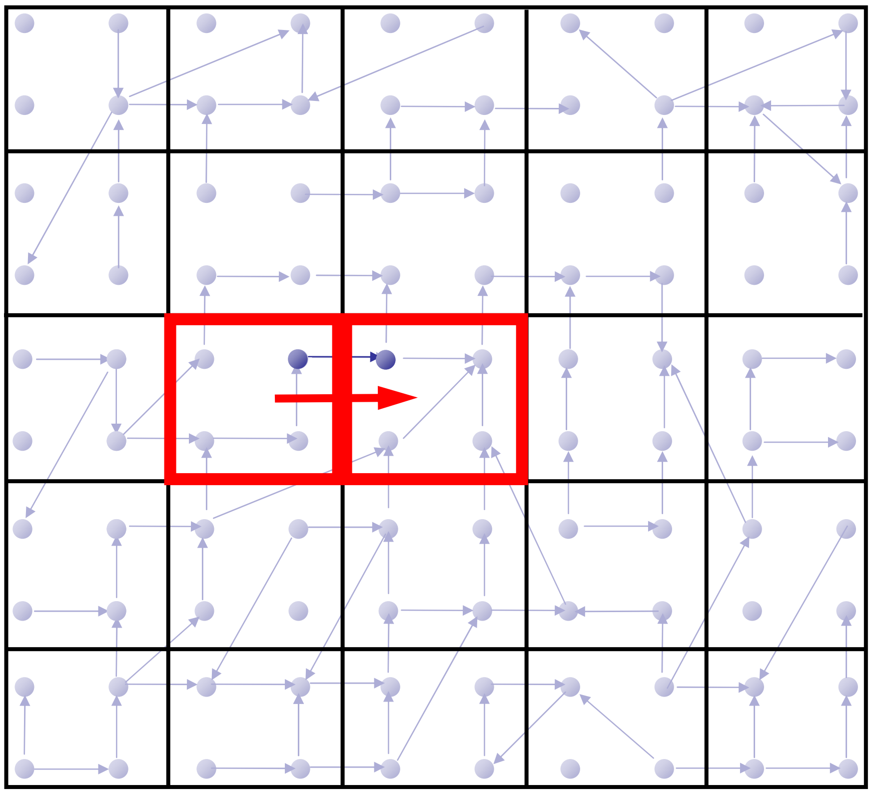
Predicate Abstraction

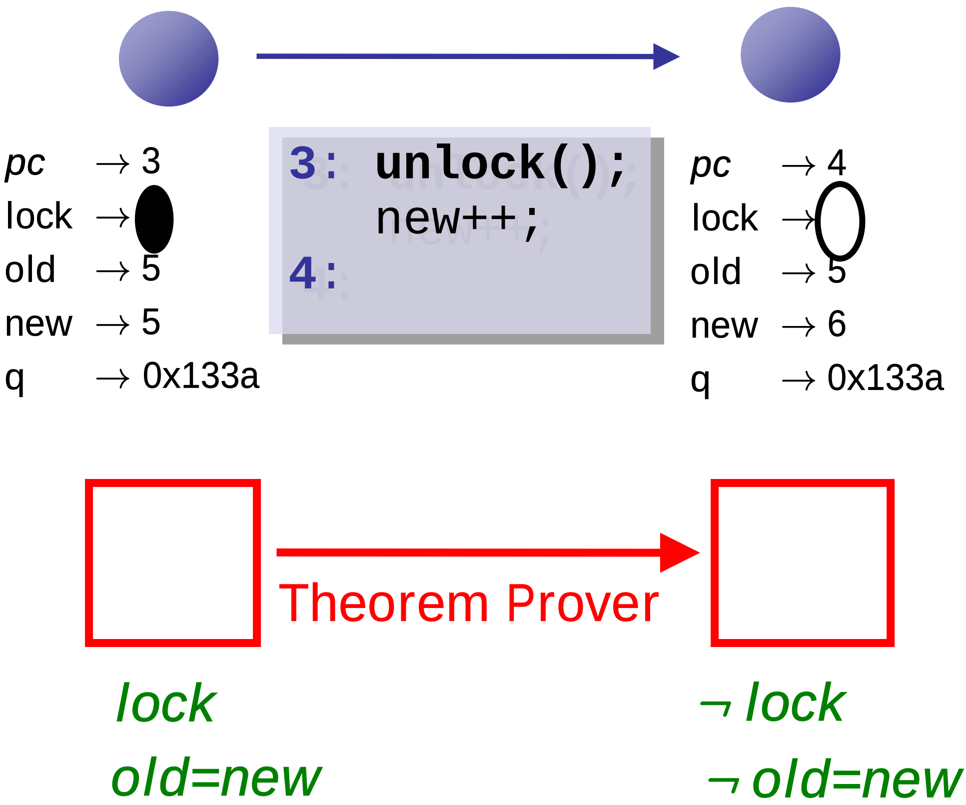


- Predicates on program state:
  lock
  old = new
- States satisfying same predicates are equivalent:
  merged into one abstract state
- Number of abstract states is finite

[Cousot-Cousot, Graf-Saidi]

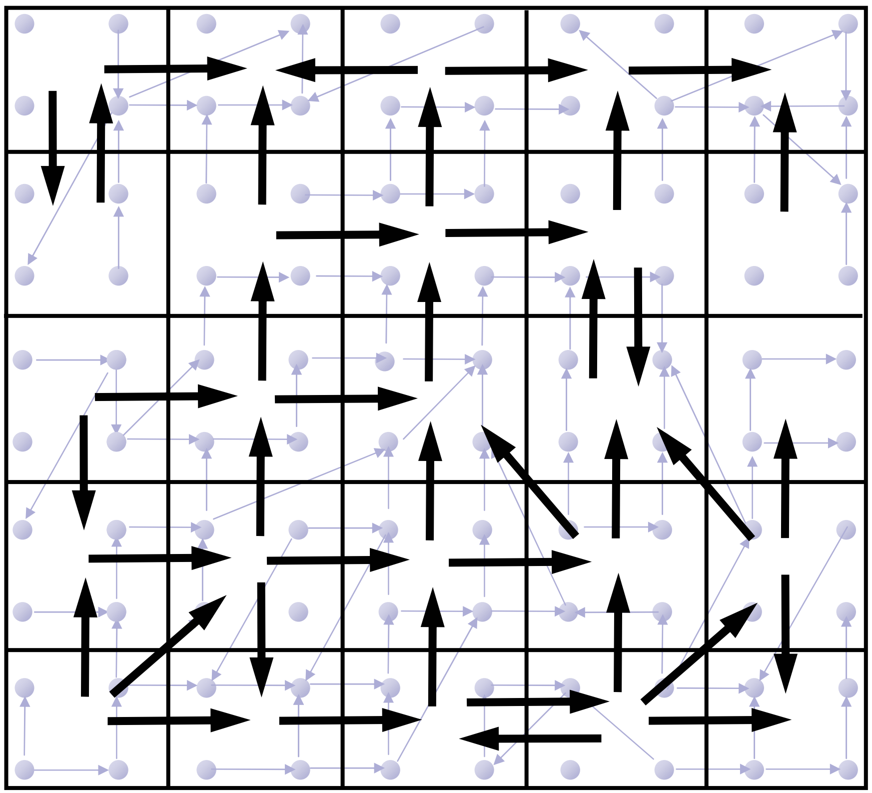
Abstract States and Transitions



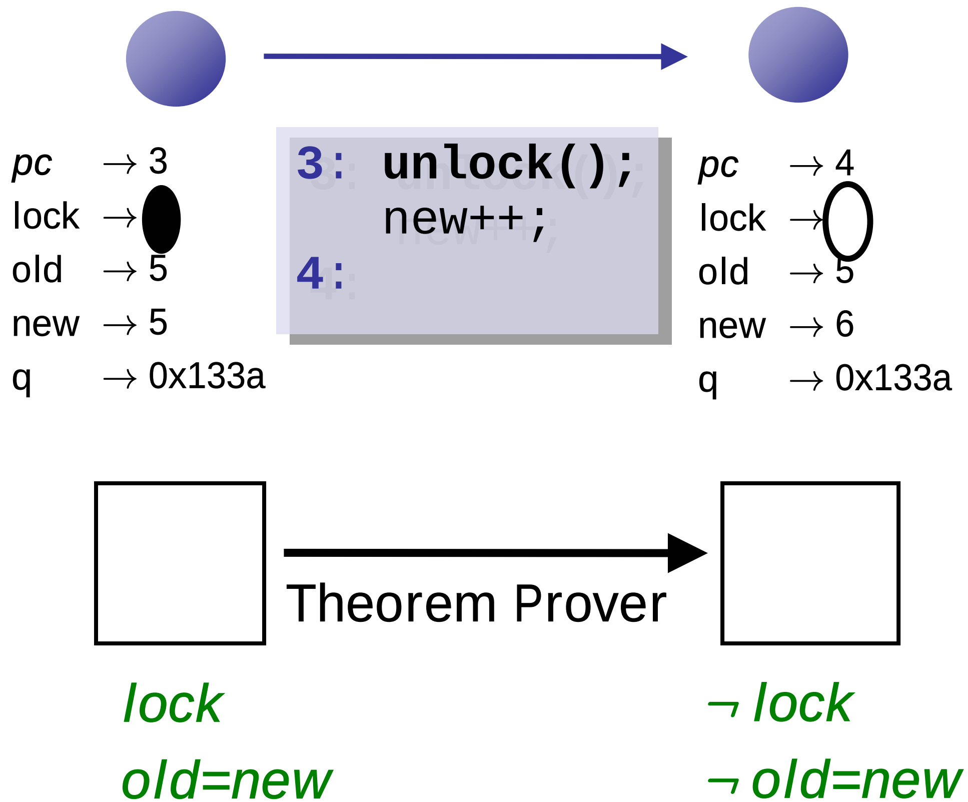


[Cousot-Cousot, Graf-Saidi]

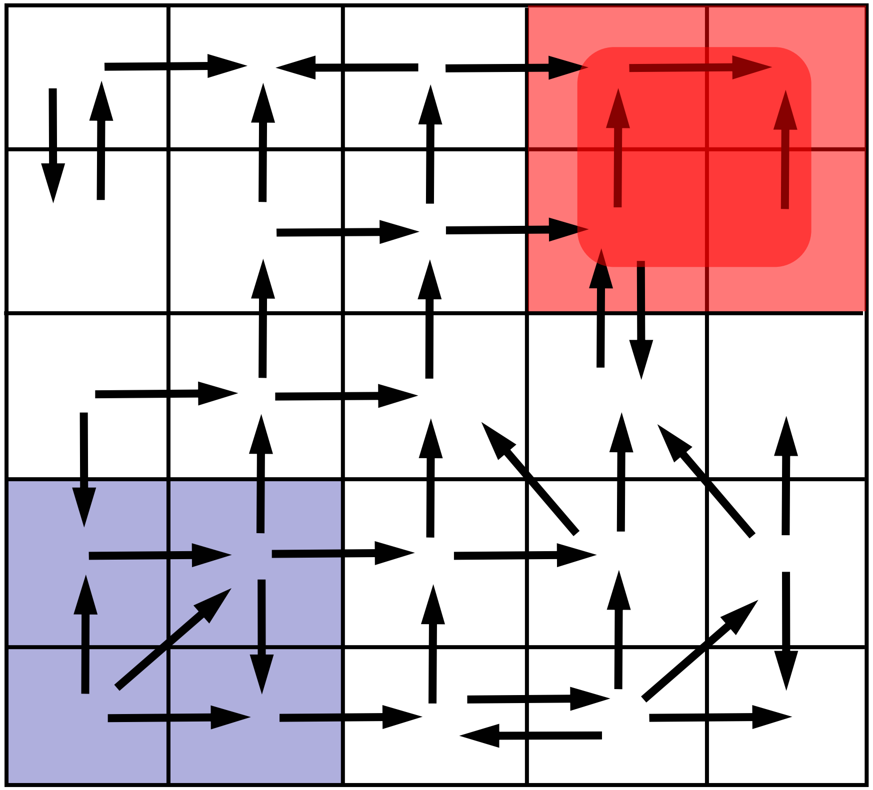
Abstraction



Existential Lifting

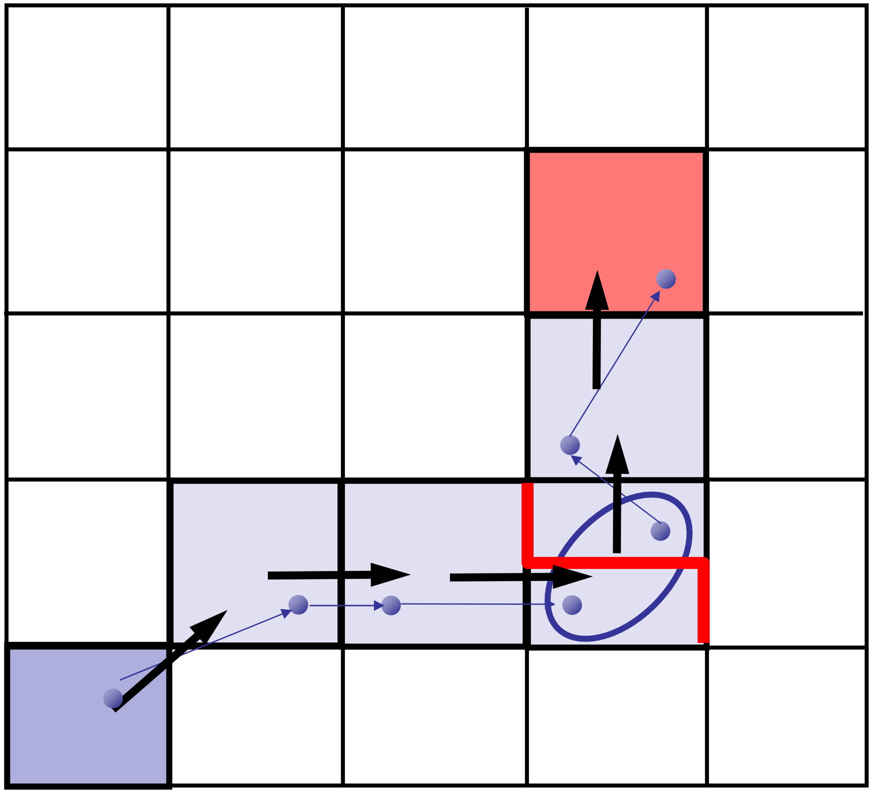


Analyze Abstraction



- Analyze finite graph
- **Over**approximation:
  - safe ⇒ program safe
  - no false negatives
- Problem: spurious counterexamples
  - false positives!

Counterexample-guided Refinement



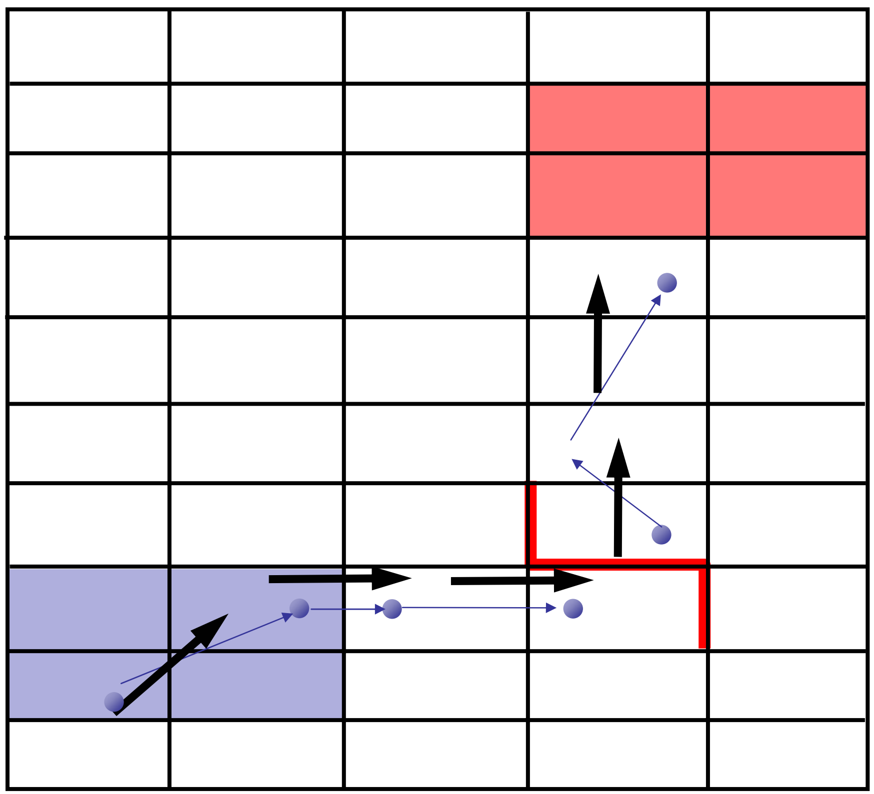
Solution:

Use spurious counterexamples to refine abstraction

1. Add predicates to distinguish states across cut

Imprecision due to merge

Iterative Abstraction Refinement



[Kurshan et al., Clarke et al.]

[Ball-Rajamani 01: SLAM]

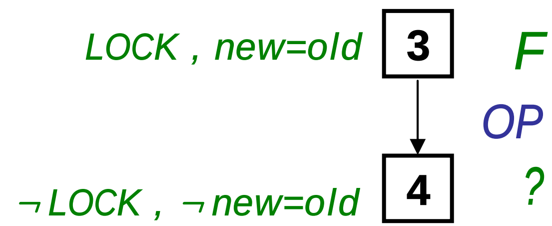
Solution:

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 found or program proved safe

How to Compute Successors ?

```
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 predicate *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 Prover query: $F \Rightarrow WP(p, OP)$?

Predicates: *LOCK, new=old*

How to Compute Successors ?

```
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();
}
```

LOCK , new=old **3** *F*

$\downarrow$

¬LOCK , ¬new=old **4** *?*

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

For each predicate *p*

check if *p* is true (or false) after *OP*

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

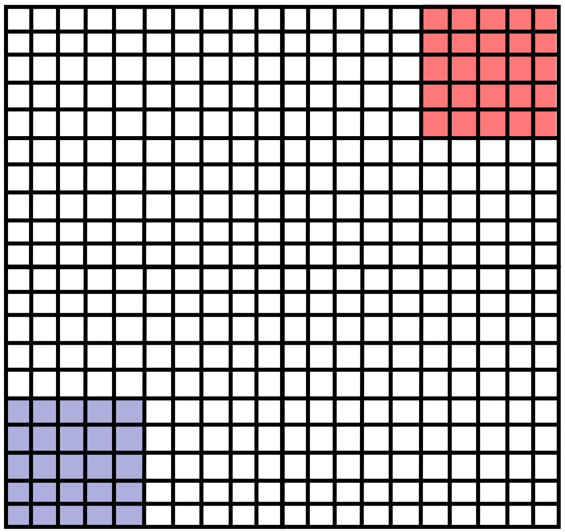
- if *WP(p, OP)* is true <u>before</u> *OP*
- we know *F* is true <u>before</u> *OP*
- Thm Prover query: *F* $\Rightarrow$ *WP(p, OP)*

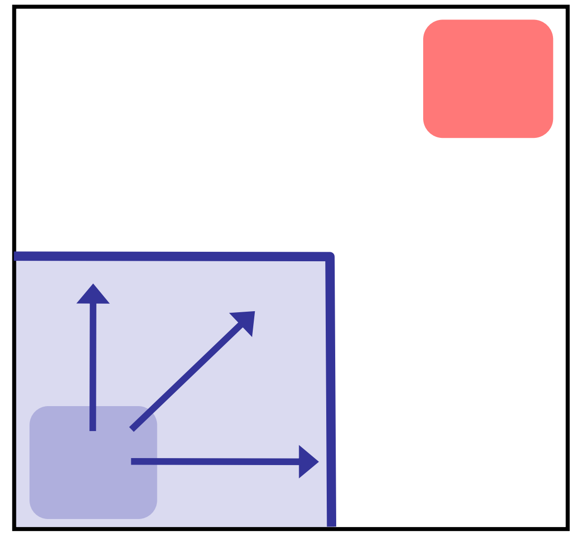
Predicate: *new=old*

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

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

Abstraction is Expensive





Reachable

Problem:

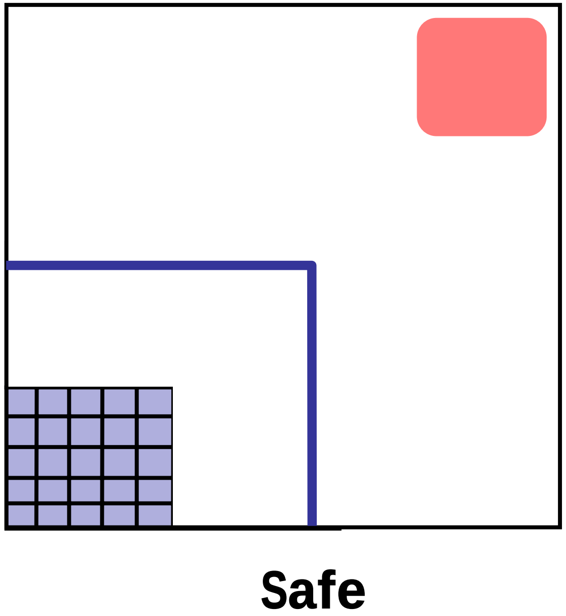
-#abstract states = $2^{\#predicates}$

-exponential Thm Prover queries

Observe:

often only fraction of state space reachable

Abstract Only Reachable States



Problem:

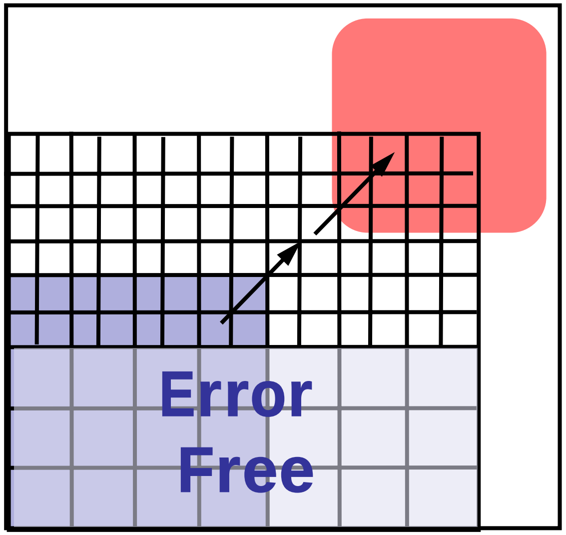
-#abstract states = $2^{\#predicates}$

-exponential Thm Prover queries

Solution:

build abstraction during search

Don’t Refine Error-Free Regions



Problem:

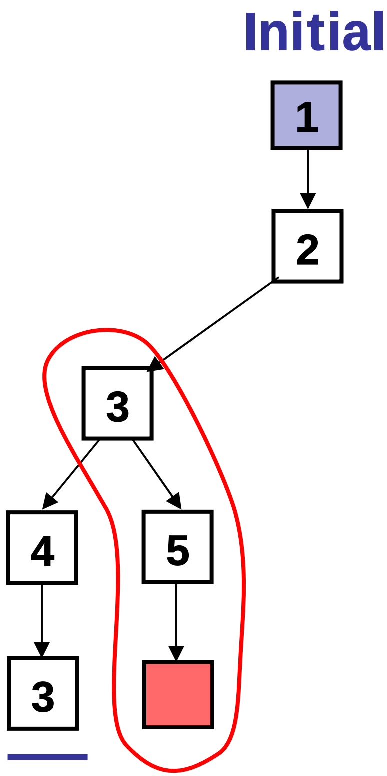
-#abstract states = $2^{\#predicates}$

-exponential Thm Prover queries

Solution:

Refine only spurious counterexamples

Putting It Together: Lazy Abstraction



Unroll abstraction

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

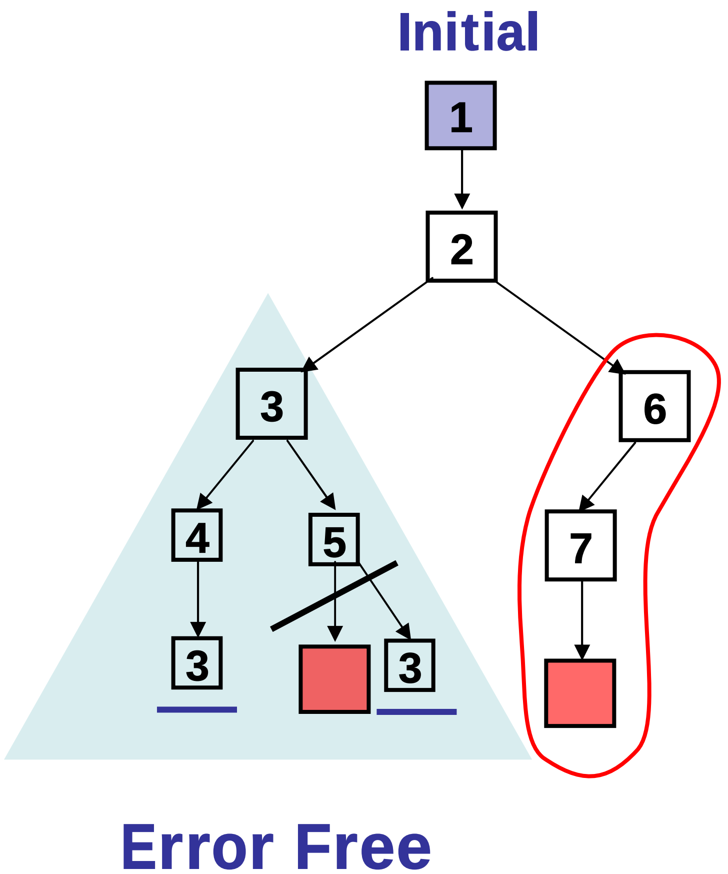
Find min. spurious suffix

- learn new predicates
- rebuild subtree with new predicates

Abstract Reachability Tree

[Henzinger-Jhala-Majumdar-Sutre 03: BLAST]

Putting It Together: Lazy Abstraction



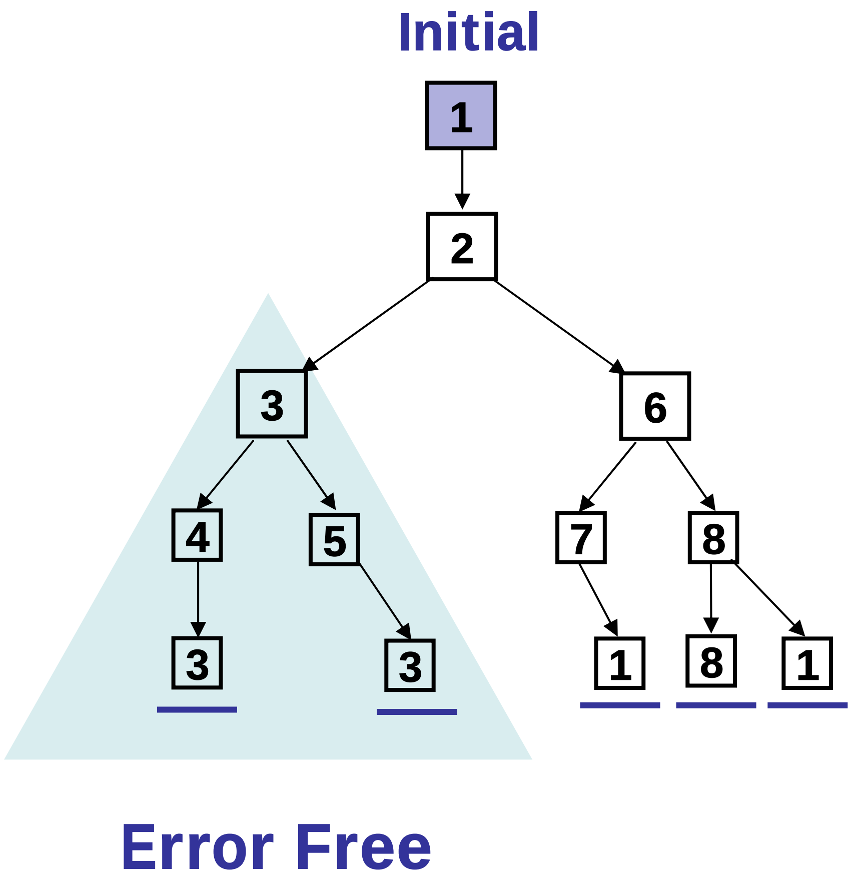
Unroll abstraction

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

Find min. spurious suffix

- learn new predicates
- rebuild subtree with new predicates

Putting It Together: Lazy Abstraction



SAFE

Unroll abstraction

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

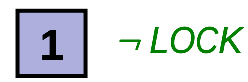
Find min. spurious suffix

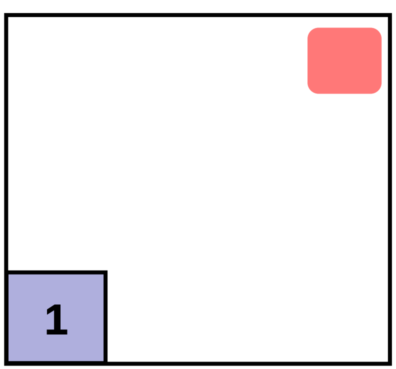
- learn new predicates
- rebuild subtree with new predicates

Only abstract reachable states
Don't refine error-free regions

Abstract Interpretation

```
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();
}
```



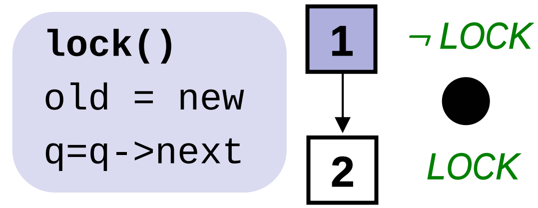


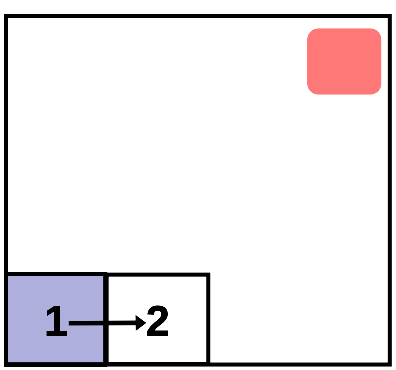
Predicates: *LOCK*

Abstract Reachability Tree

Abstract Interpretation

```
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 ();
}
```



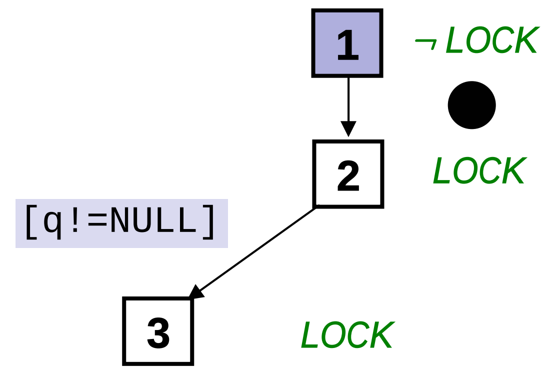


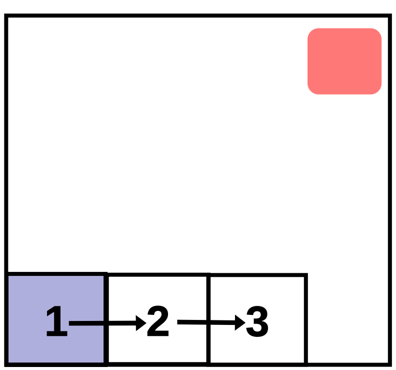
Predicates: *LOCK*

Abstract Reachability Tree

Abstract Interpretation

```
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();
}
```



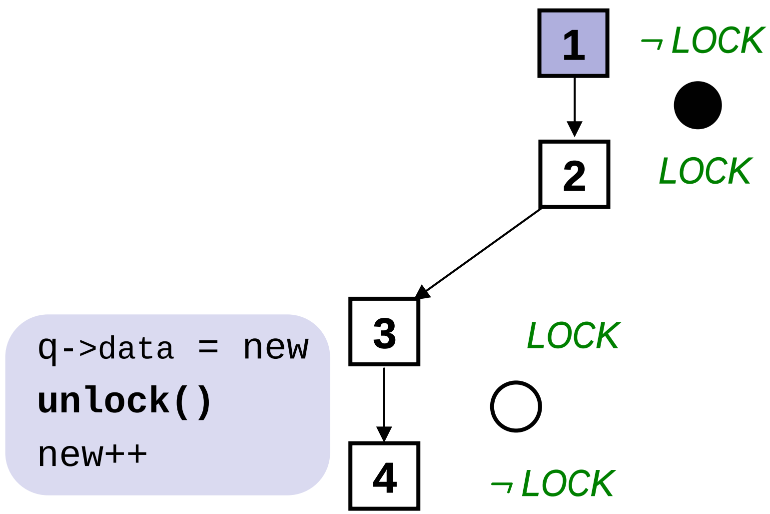


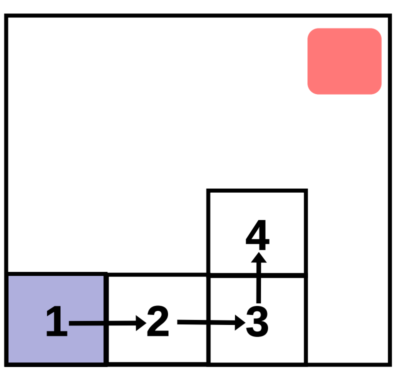
Predicates: *LOCK*

Abstract Reachability Tree

Abstract Interpretation

```
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();
}
```



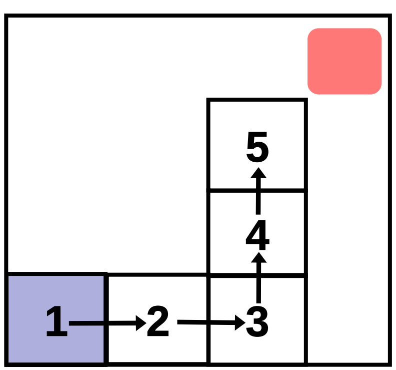


Predicates: *LOCK*

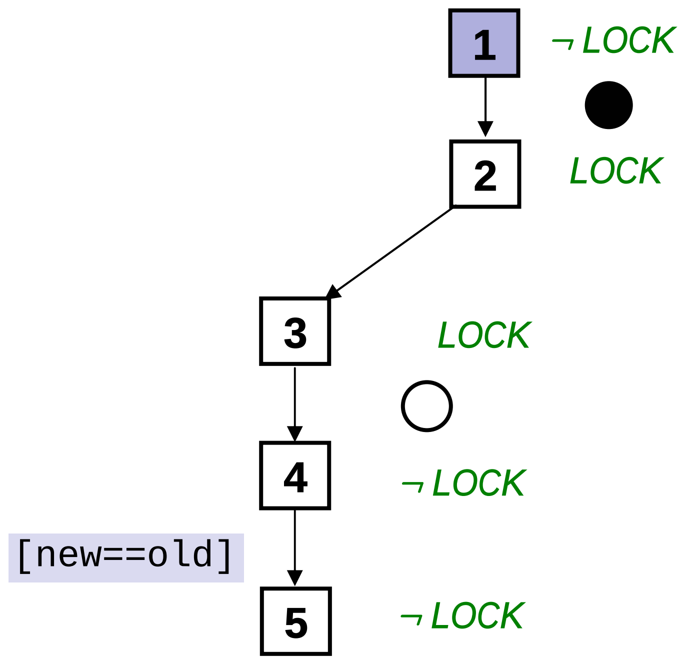
Abstract Reachability Tree

Abstract Interpretation

```
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();
}
```



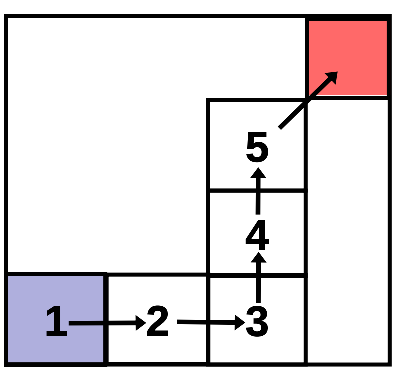
Predicates: *LOCK*



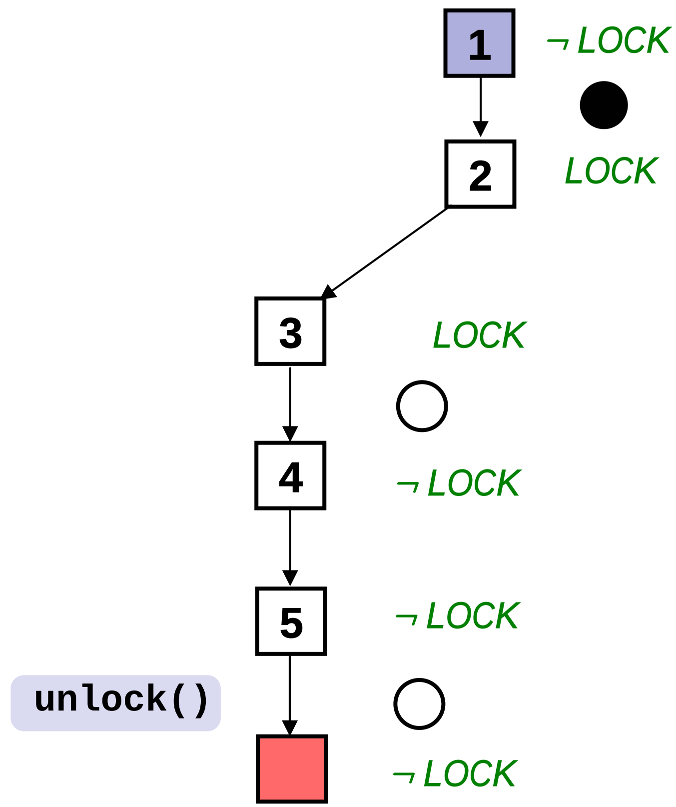
Abstract Reachability Tree

Abstract Interpretation

```
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();
}
```



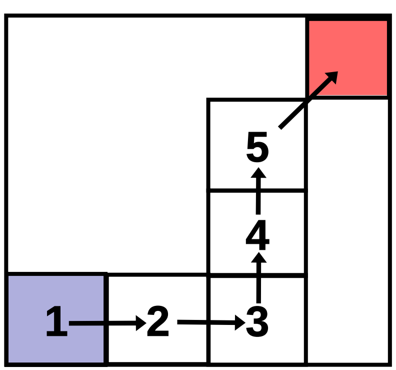
Predicates: *LOCK*



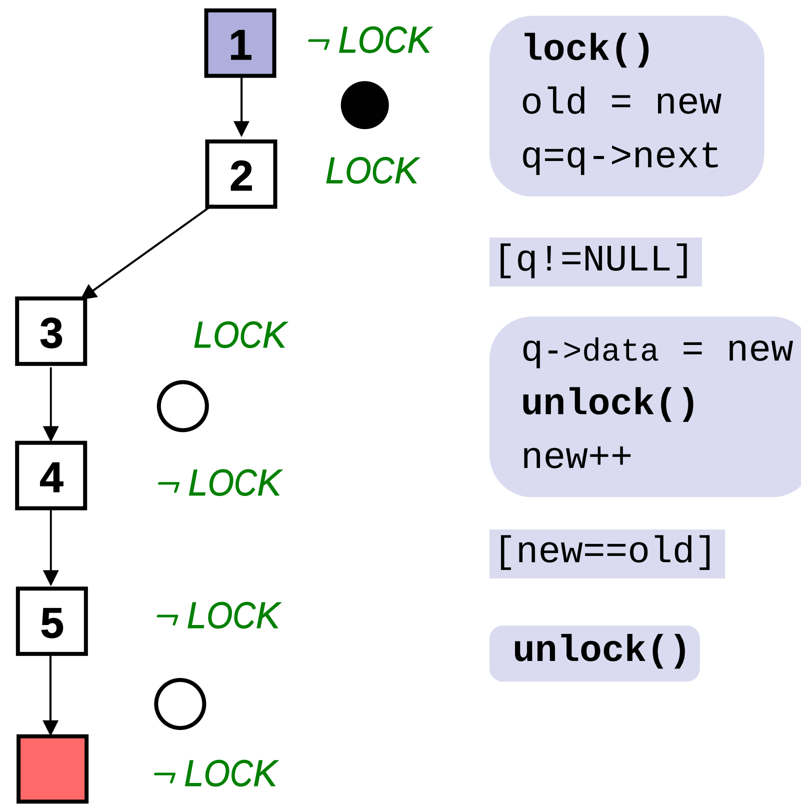
Abstract Reachability Tree

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();
}
```



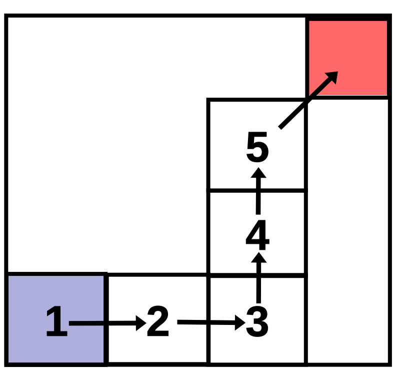
Predicates: *LOCK*



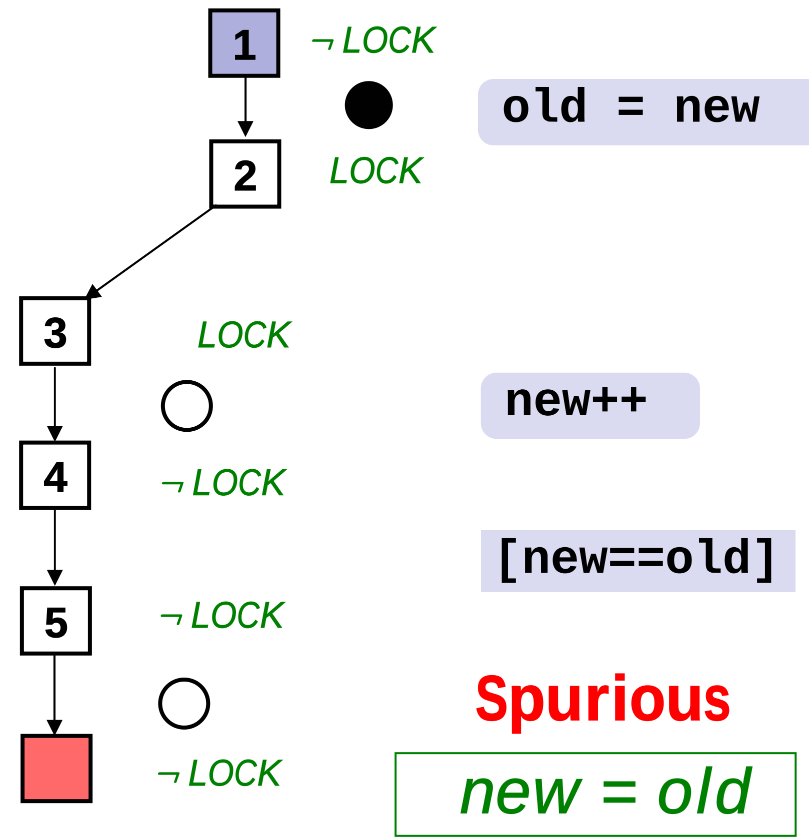
Abstract Reachability Tree

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();
}
```



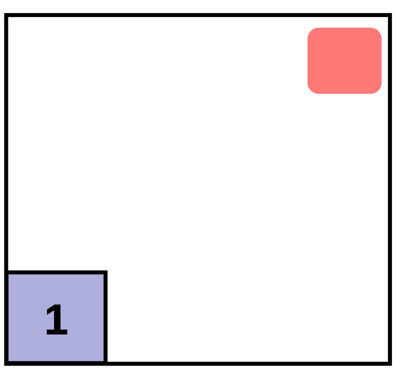
Predicates: *LOCK*



Abstract Reachability Tree

Refined Abstract Interpretation

```
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();
}
```

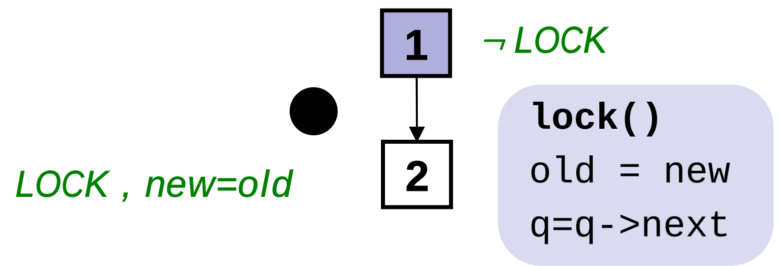


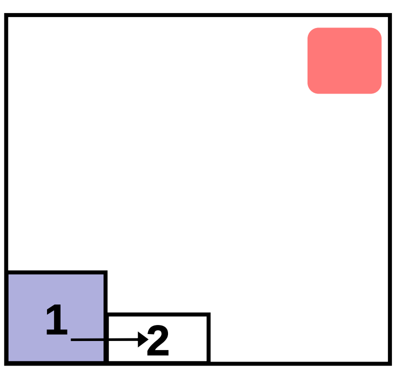
Abstract Reachability Tree

Predicates: *LOCK, new=old*

Refined Abstract Interpretation

```
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();
}
```



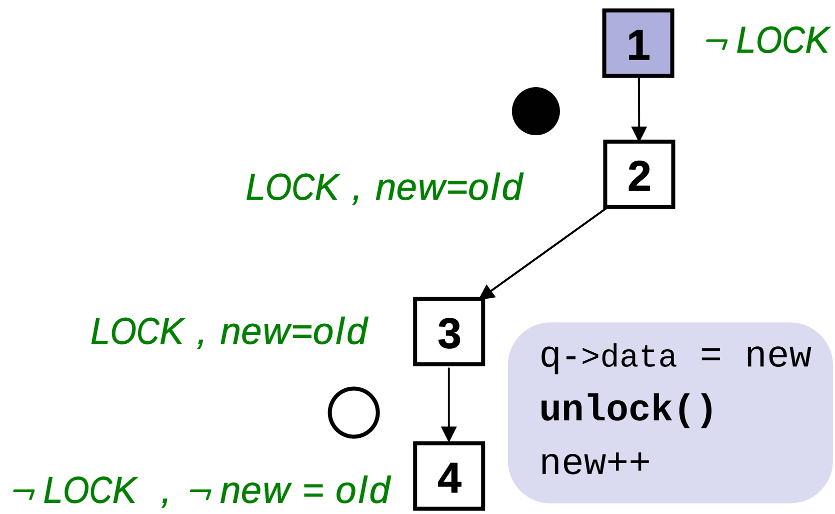


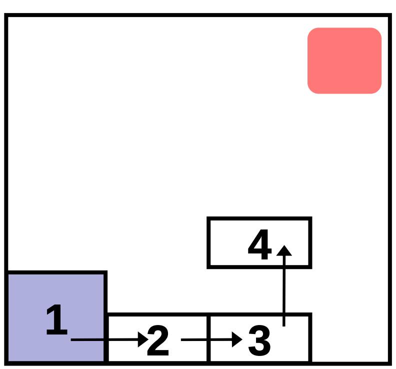
Predicates: *LOCK, new=old*

Abstract Reachability Tree

Refined Abstract Interpretation

```
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();
}
```



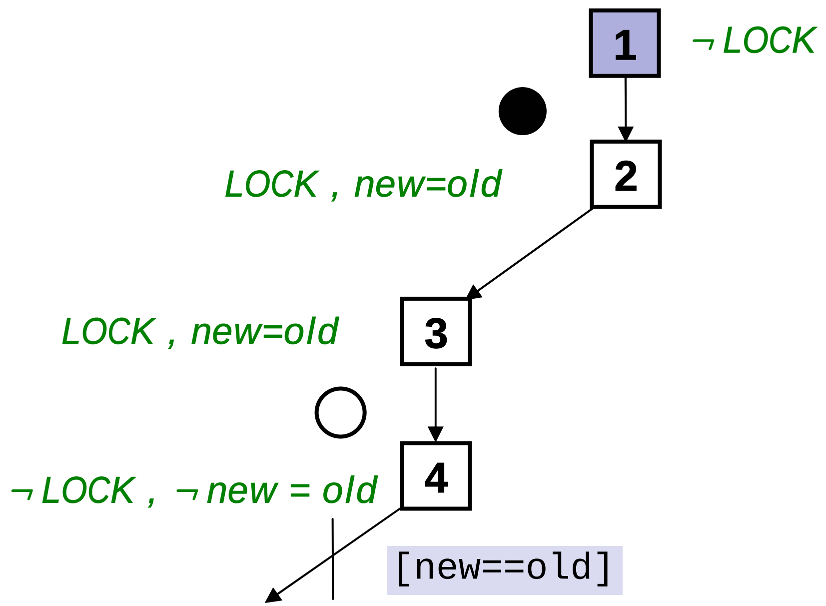


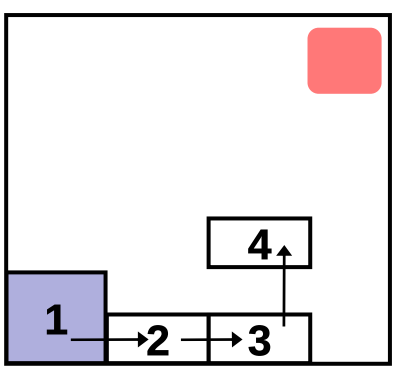
Predicates: *LOCK, new=old*

Abstract Reachability Tree

Refined Abstract Interpretation

```
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();
}
```



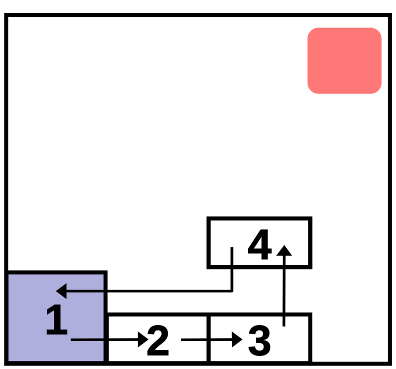


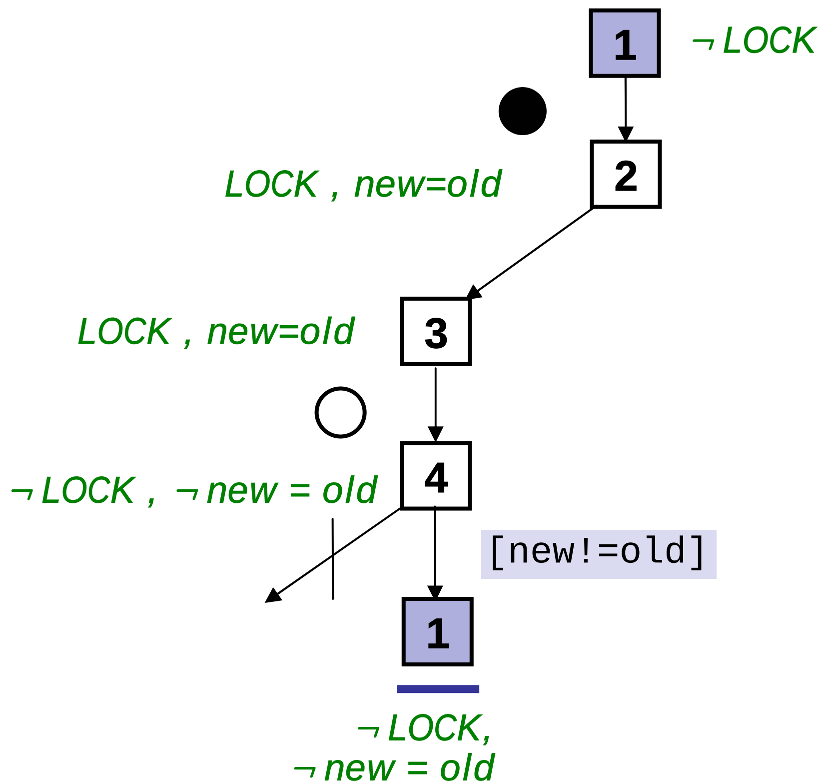
Abstract Reachability Tree

Predicates: *LOCK, new=old*

Refined Abstract Interpretation

```
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();
}
```



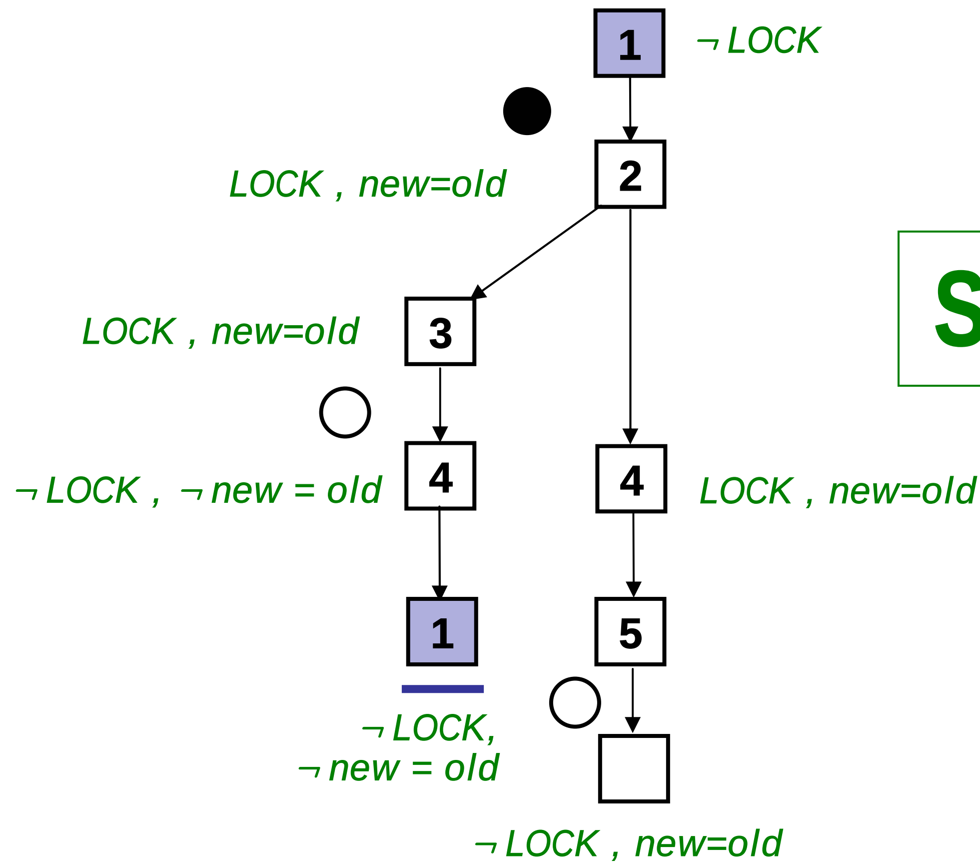


Abstract Reachability Tree

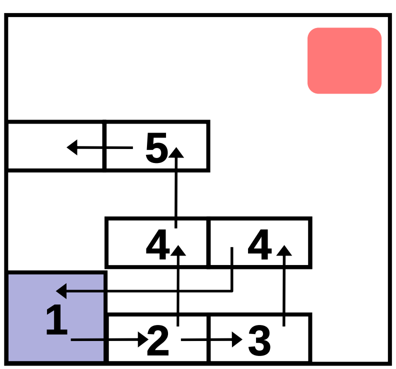
Predicates: *LOCK, new=old*

Refined Abstract Interpretation

```
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();
}
```



SAFE



Abstract Reachability Tree

Predicates: LOCK, new=old

#Predicates Grows with Program Size

```
    while(1){
T ● 1: if (p1) lock() ;
F      if (p1) unlock() ;
                 …
T ● 2: if (p2) lock() ;
       if (p2) unlock() ;
                 …
    n: if (pn) lock() ;
       if (pn) unlock() ;
    }
```

Tracking *lock* not enough

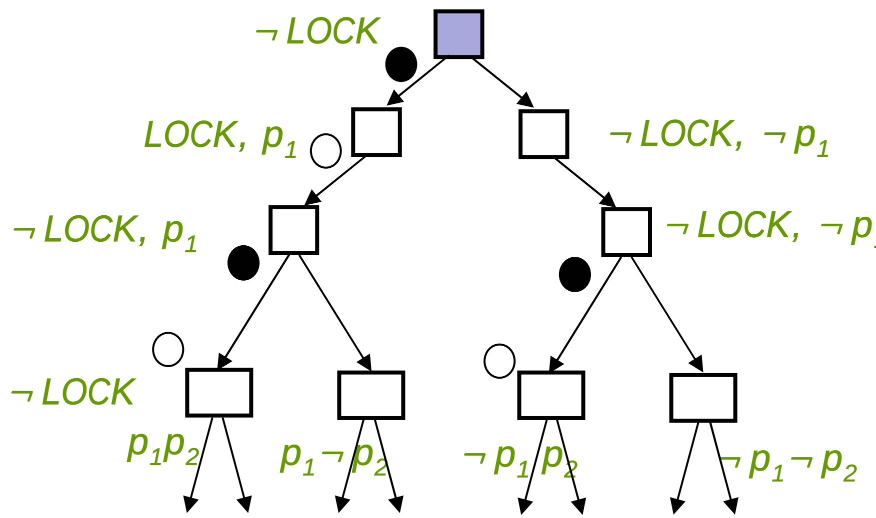
Problem:

$p_1,\ldots,p_n$ needed for verification

#Reachable abstract states exponential

#Predicates Grows with Program Size

```
while(1){
1: if (p1) lock() ;
   if (p1) unlock() ;
              …
2: if (p2) lock() ;
   if (p2) unlock() ;
              …
n: if (pn) lock() ;
   if (pn) unlock() ;
}
```



2^n abstract states

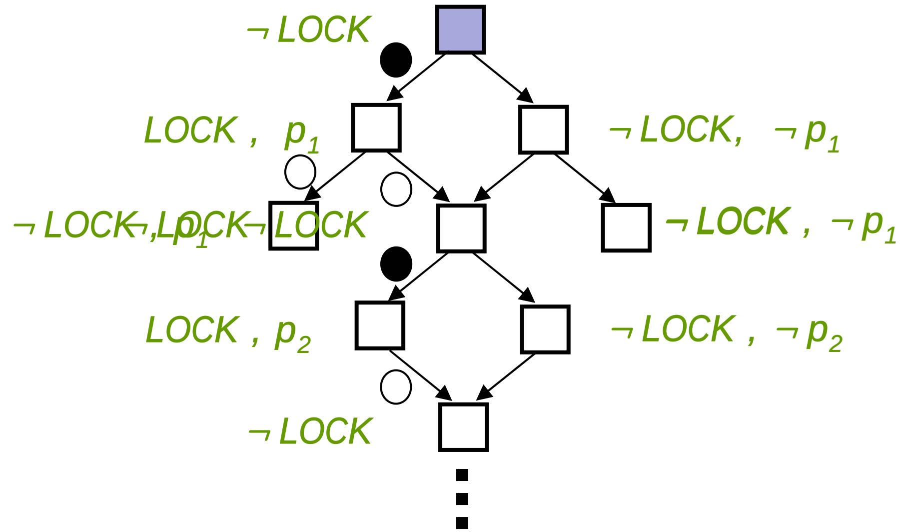
Problem:

$p_1,\ldots,p_n$ needed for verification

#Reachable abstract states exponential

Predicates useful *Locally*

```
      while(1){
      1: if (p1) lock() ;
p1         if (p1) unlock() ;
                 …
      2: if (p2) lock() ;
p2         if (p2) unlock() ;
                 …
      n: if (pn) lock() ;
pn         if (pn) unlock() ;
      }
```



2n abstract states

Solution: Use predicates **only where needed**

Using **counterexamples**:

Q1. Find **predicates**

Q2. Find **where** predicates are needed

Counterexample Trace

```
1: x = ctr;
2: ctr = ctr + 1;
3: y = ctr;
4: if (x = i-1){
5:    if (y != i){
          ERROR: }
   }
```

```
1: x = ctr
2: ctr = ctr + 1
3: y = ctr
4: assume(x = i-1)
5: assume(y ≠ i)
```

$y = x + 1$

Build Trace Formula

1: `x = ctr`	1: x_1 = ctr_0	$x_1 = ctr_0$
2: `ctr = ctr+1`	2: ctr_1 = ctr_0+1	$\wedge\ ctr_1 = ctr_0 + 1$
3: `y = ctr`	3: y_1 = ctr_1	$\wedge\ y_1 = ctr_1$
4: `assume(x=i-1)`	4: assume(x_1=i_0-1)	$\wedge\ x_1 = i_0 - 1$
5: assume(y≠i)	5: assume(y_1≠i_0)	$\wedge\ y_1 \neq i_0$
Trace	SSA Trace	Trace Formula

Trace is **feasible** ⇔ TF is **satisfiable**

Which Predicate is Needed ?

Trace	Trace Formula (TF)
1: x = ctr	$x_1 = ctr_0$
2: ctr = ctr + 1	$\wedge\ ctr_1 = ctr_0 + 1$
3: y = ctr	$\wedge\ y_1 = ctr_1$
4: assume(x = i-1)	$\wedge\ x_1 = i_0 - 1$
5: assume(y ≠ i)	$\wedge\ y_1 \neq i_0$

Which Predicate is Needed ?

Trace

```
1: x = ctr

2: ctr = ctr + 1

3: y = ctr

4: assume(x = i-1)

5: assume(y ≠ i)
```

Trace Formula (TF)

$x_1 = ctr_0$

$\wedge \quad ctr_1 = ctr_0 + 1$

$\wedge \quad y_1 = ctr_1$

$\wedge \quad x_1 = i_0 - 1$

$\wedge \quad y_1 \neq i_0$

Relevant Information

1. ... after executing trace **prefix**

Predicate ...

... implied by TF **prefix**

Which Predicate is Needed ?

Trace	Trace Formula (TF)
`1: x = ctr`	$x_1 = ctr_0$
`2: ctr = ctr + 1`	$\wedge \quad ctr_1 = ctr_0 + 1$
`3: y = ctr`	$\wedge \quad y_1 = ctr_1$
`4: assume(x = i-1)`	$\wedge \quad x_1 = i_0 - 1$
`5: assume(y ≠ i)`	$\wedge \quad y_1 \neq i_0$

Relevant Information

1. ... after executing trace **prefix**
2. ... has **present values** of variables

Predicate ...

... implied by TF **prefix**

... on **common** variables

Which Predicate is Needed ?

Trace

```
1: x = ctr
2: ctr = ctr + 1
3: y = ctr
4: assume(x = i-1)
5: assume(y ≠ i)
```

Trace Formula (TF)

$$x_1 = ctr_0$$
$$\wedge \quad ctr_1 = ctr_0 + 1$$
$$\wedge \quad y_1 = ctr_1$$
$$\wedge \quad x_1 = i_0 - 1$$
$$\wedge \quad y_1 \neq i_0$$

Relevant Information

1. ... after executing trace **prefix**
2. ... has **present values** of variables
3. ... makes trace **suffix** infeasible

Predicate ...

... implied by TF **prefix**

... on **common** variables

... and TF **suffix** is **unsatisfiable**

Predicate = Interpolant

Trace		Trace Formula
`1: x = ctr`		$x_1 = ctr_0$
`2: ctr = ctr + 1`	$\wedge$	$ctr_1 = ctr_0 + 1$
`3: y = ctr`	$\wedge$	$y_1 = ctr_1$
`4: assume(x = i-1)`	$\wedge$	$x_1 = i_0 - 1$
`5: assume(y ≠ i)`	$\wedge$	$y_1 \neq i_0$

Predicate at 4:

$y = x + 1$

ψ^-

Interpolate Φ

ψ^+

$y_1 = x_1 + 1$

Craig Interpolant

Predicate ...

... implied by TF prefix

Computable from proof of unsatisfiability

[Krajicek. Pudlak, McMillan]

... on common variables

... and TF suffix is unsatisfiable

Building Predicate Maps

Predicate Map
2: $x = ctr$

Trace	Trace Formula
***1*: `x = ctr`**	$x_1 = ctr_0$
***2*: `ctr = ctr + 1`**	$\wedge \quad ctr_1 = ctr_0 + 1$
***3*: `y = ctr`**	$\wedge \quad y_1 = ctr_1$
***4*: `assume(x = i-1)`**	$\wedge \quad x_1 = i_0 - 1$
***5*: `assume(y ≠ i)`**	$\wedge \quad y_1 \neq i_0$

ψ^-

ψ^+

Interpolate $\Rightarrow$ $x_1 = ctr_0$

- Cut + interpolate at **each** point
- Predicate Map: $\mathbf{pc_i} \rightarrow$ interpolant from cut i

Building Predicate Maps

Predicate Map
2: $x = ctr$
3: $x = ctr-1$

Trace	Trace Formula
`1: x = ctr`	$x_1 = ctr_0$
`2: ctr = ctr + 1`	$\wedge\ ctr_1 = ctr_0 + 1$
`3: y = ctr`	$\wedge\ y_1 = ctr_1$
`4: assume(x = i-1)`	$\wedge\ x_1 = i_0 - 1$
`5: assume(y ≠ i)`	$\wedge\ y_1 \neq i_0$

ψ^-

ψ^+

Interpolate

$x_1 = ctr_1 - 1$

- Cut + interpolate at **each** point
- Predicate Map: **pc_i** $\rightarrow$ interpolant from cut i

Building Predicate Maps

Predicate Map
2: $x = ctr$
3: $x = ctr - 1$
4: $y = x + 1$

Trace	Trace Formula
`1: x = ctr`	$x_1 = ctr_0$
`2: ctr = ctr + 1`	$\wedge\ ctr_1 = ctr_0 + 1$
`3: y = ctr`	$\wedge\ y_1 = ctr_1$
`4: assume(x = i-1)`	$\wedge\ x_1 = i_0 - 1$
`5: assume(y ≠ i)`	$\wedge\ y_1 \neq i_0$

ψ^-

ψ^+

Interpolate

$y_1 = x_1 + 1$

- Cut + interpolate at **each** point
- Predicate Map: **pc$_i$** $\rightarrow$ interpolant from cut i

Building Predicate Maps

Predicate Map
2: $x = ctr$
3: $x = ctr - 1$
4: $y = x + 1$
5: $y = i$

Trace	Trace Formula
`1: x = ctr`	$x_1 = ctr_0$
`2: ctr = ctr + 1`	$\wedge \quad ctr_1 = ctr_0 + 1$
`3: y = ctr`	$\wedge \quad y_1 = ctr_1$
`4: assume(x = i-1)`	$\wedge \quad x_1 = i_0 - 1$
`5: assume(y ≠ i)`	$\wedge \quad y_1 \neq i_0$

ψ^-

Interpolate $\Rightarrow$ $y_1 = i_0$

ψ^+

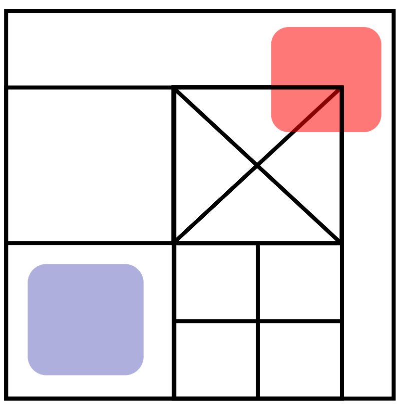
- Cut + interpolate at **each** point
- Predicate Map: **pc_i** $\rightarrow$ interpolant from cut i

Local Predicate Use

Use predicates **needed** at **location**

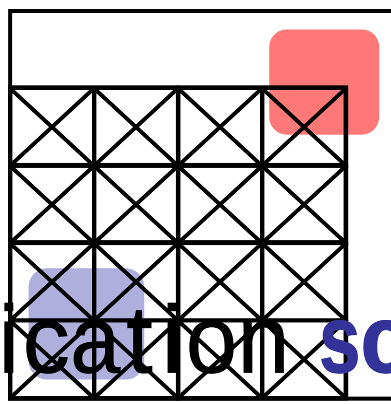
- #Preds grows with program size
- **#Predicates per location** small

Predicate Map
2: *x = ctr*
3: *x= ctr - 1*
4: *y= x + 1*
5: *y = i*



Local Predicate use

Ex: O(n) states



Verification **scales !**

Global Predicate use

Ex: 2^n states

BLAST

Property 3:
IRP Handler
Win NT DDK

Program	*Lines**	*Non-lazy Time (mins)*	*Lazy Time (mins)*	*Predicates Total*	*Predicates Average*
kbfiltr	12k	1	3	72	*6.5*
floppy	17k	7	25	240	*7.7*
diskprf	14k	5	13	140	*10*
cdaudio	18k	20	23	256	*7.8*
parport	61k	*DNF*	74	753	*8.1*
parclss	138k	*DNF*	77	382	*7.2*

* Pre-processed

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 formula

3. Check validity

Invariant:

$$lock \wedge new = old$$
$$\vee$$
$$\neg\, lock \wedge new \neq old$$

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 formula**
3. **Check validity**

- Loop invariants
- Multithreaded programs
+ Behaviors encoded in logic
+ Decision procedures

Precise [e.g. ESC]

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 due to fixed facts

+ Abstraction

+ Type/flow analyses

Scalable [e.g. CQUAL, ESP, MC]

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

- **Finite state model**
- **State explosion**

\+ **State exploration**

\+ **Counterexamples**

Automatic [e.g. SPIN, Bandera, JPF]

Combining Strengths

Theorem Proving

- **Loop invariants**

+ Behaviors encoded in logic
+ Theorem provers
Computing successors; refine

Program Analysis

- **Imprecise**

+ Abstraction
Shrink state space

Lazy Abstraction

Model Checking

- **Finite-state model, state explosion**

+ State space exploration
Path-sensitive analysis
+ Counterexamples
Finding relevant facts

Conclusions

-take the best of each technology:
 automatic, precise, scalable

-verify programs, not models:
 verifying compiler

-current research: concurrency, heap

http://mtc.epfl.ch/software-tools/blast

joint with R. Jhala (UCSD) and R. Majumdar (UCLA)