The Software Model Checker BLAST

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

BLAST 2.0 Team:
Dirk Beyer, Tom Henzinger,
Ranjit Jhala, and Rupak Majumdar

Guest Lecture in Viktor Kuncak’s Verification Class, 2008-05-08

Motivation

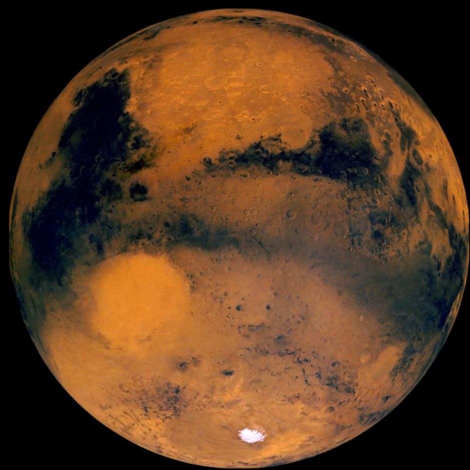
Software stands for

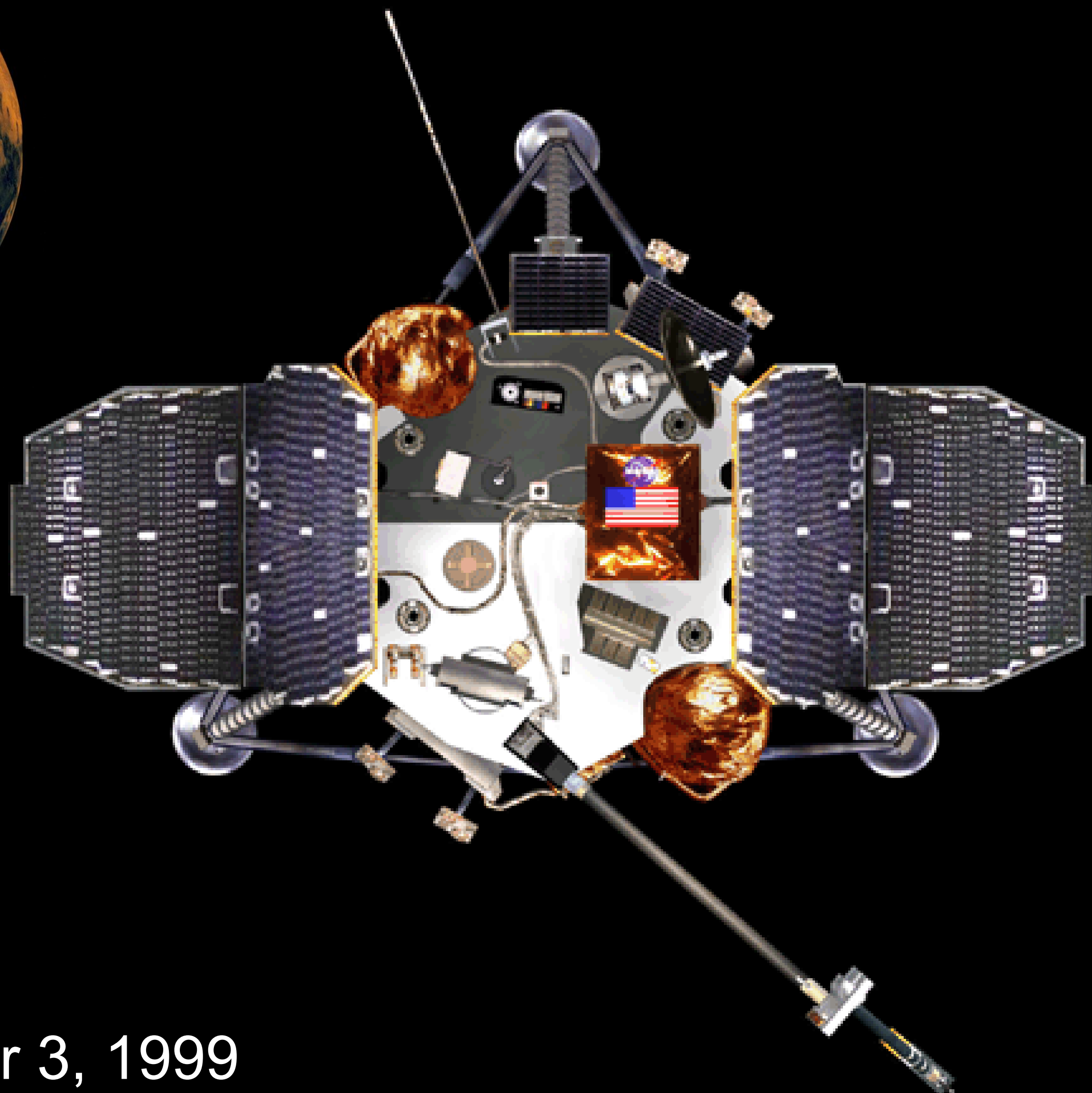
- Functionality
- Flexibility
- Affordability in today’s products and infrastructures.

Practice:

- Vulnerability
- Obstacle to redesign
- Cost overruns
- Buggy, brittle, insecure, and not interoperable.


French Guyana, June 4, 1996
$600 million software failure

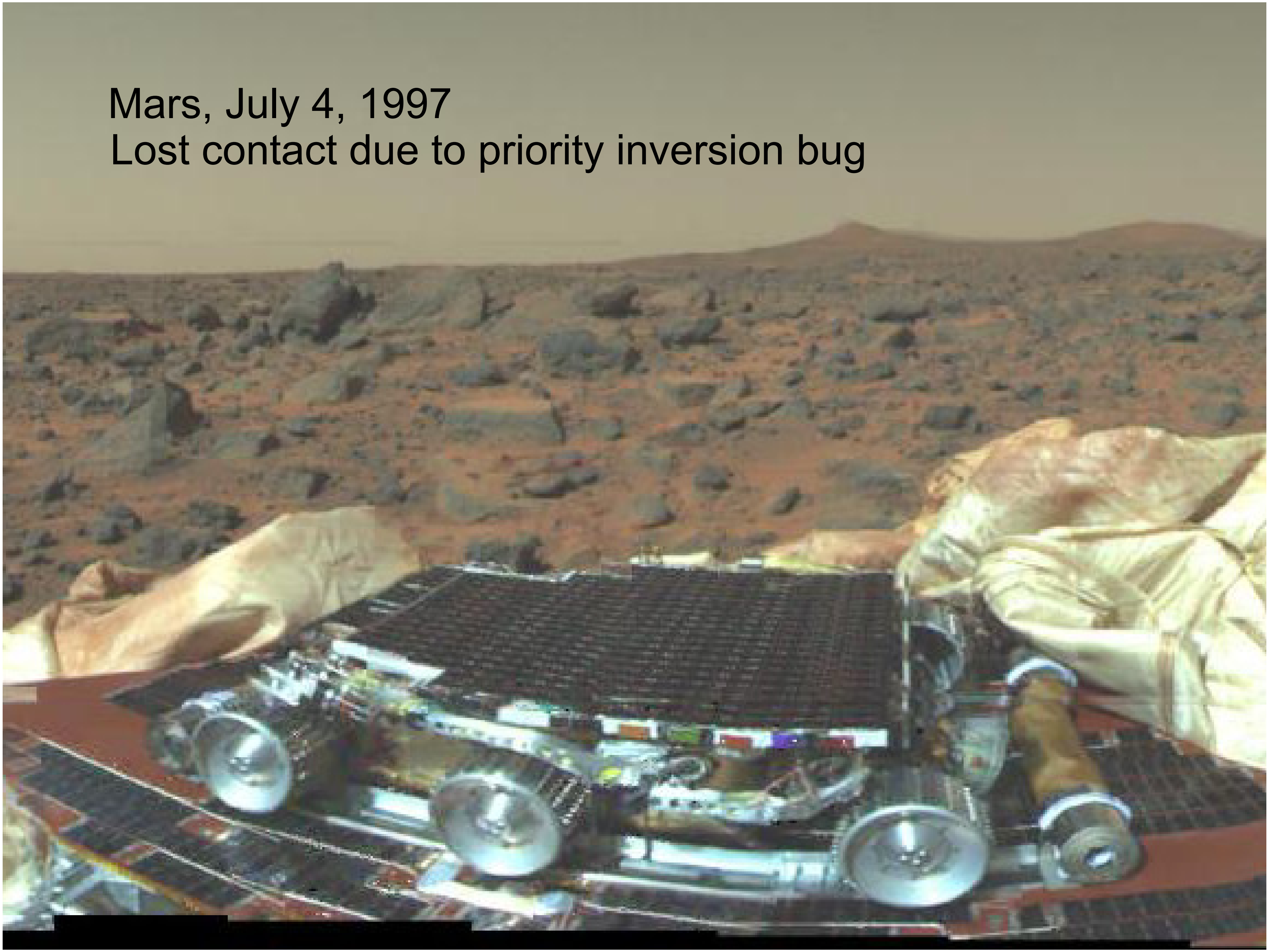




Mars, December 3, 1999
Crashed due to uninitialized variable

Mars, July 4, 1997

Lost contact due to priority inversion bug




Something reliable
Uptime: 68 years


Windows
An exception 06 has occured at 0028:C11B3ADC in VxD DiskTSD(03) +
00001660. This was called from 0028:C11B40C8 in VxD voltrack(04) +
00000000. It may be possible to continue normally.
* Press any key to attempt to continue.
* Press CTRL+ALT+RESET to restart your computer. You will
lose any unsaved information in all applications.
Press any key to continue

Our Application Areas

- Verification of systems code
  - Locking disciplines
  - Interface specifications
- Temporal properties
  - Require path-sensitive analysis
  - Swamped by false positives
- Really hard to check

Specifying and Checking Properties of Programs

- Goals
  - Defect detection
  - Partial validation
- Properties
  - Memory safety
  - Temporal safety
  - Security
  - …

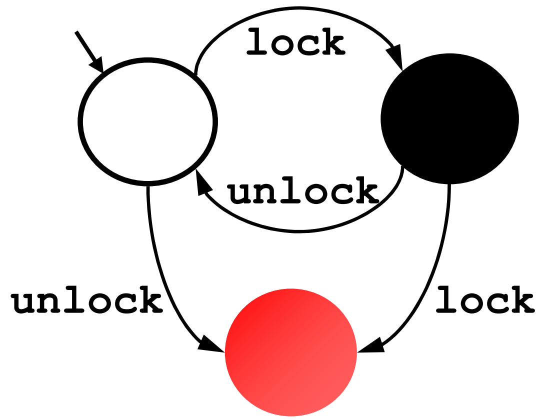
- Many (mature) techniques
  - Automated deduction
  - Program analysis
  - Type checking
  - Model checking
- Many projects

  Bandera, Blast, ESC-Java, FeaVer, JPF, LClint, OSQ, PolyScope, PREfix, SLAM, TVLA, Verisoft, xgcc, …

Property Checking

- Programmer gives partial specifications
- Code checked for consistency with spec
- Different from program correctness
  - Specifications are not complete
  - Are there actually complete specs?
  - **Look for problems that occur often**

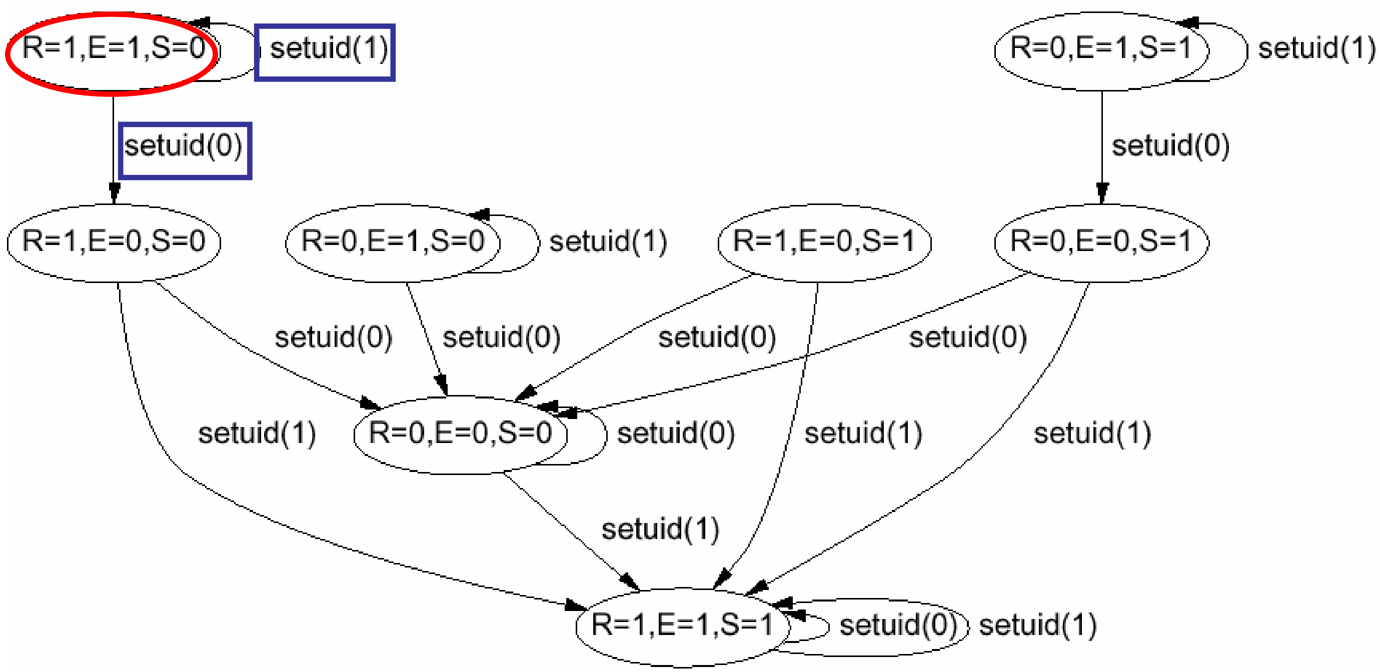
Property 1: Double Locking



*"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

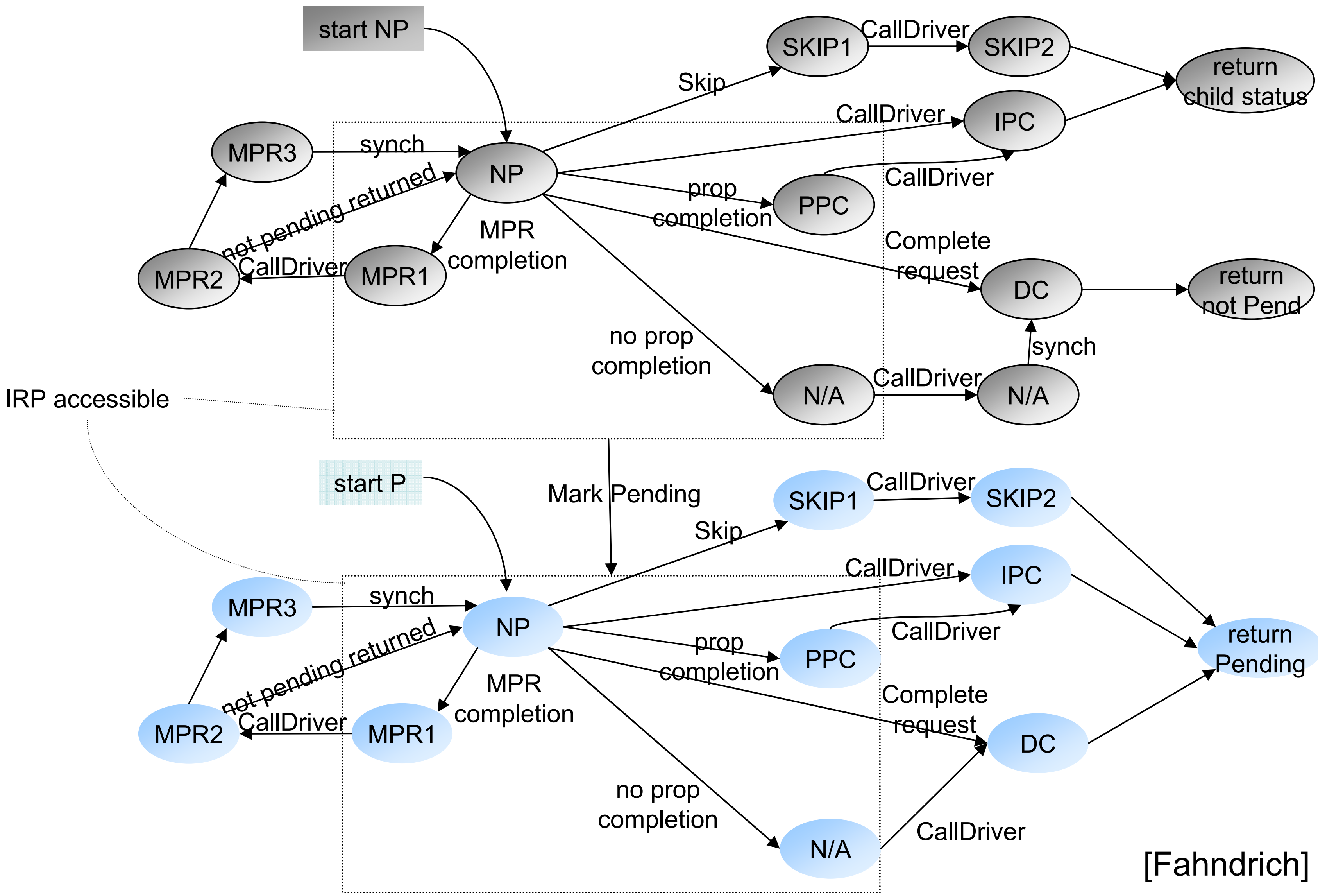


[Chen-Dean-Wagner '02]

"User applications must not run with root privilege"

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

Property 3 : IRP Handler

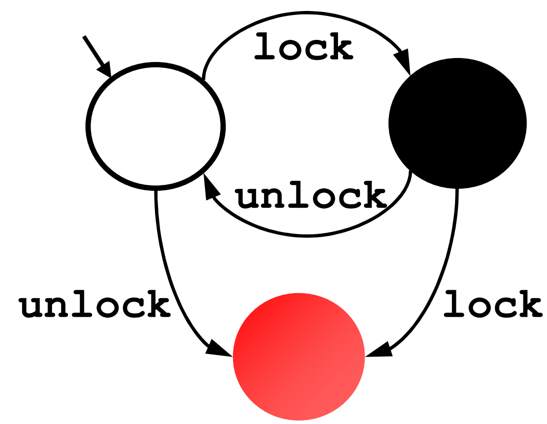


Does a given usage rule hold?

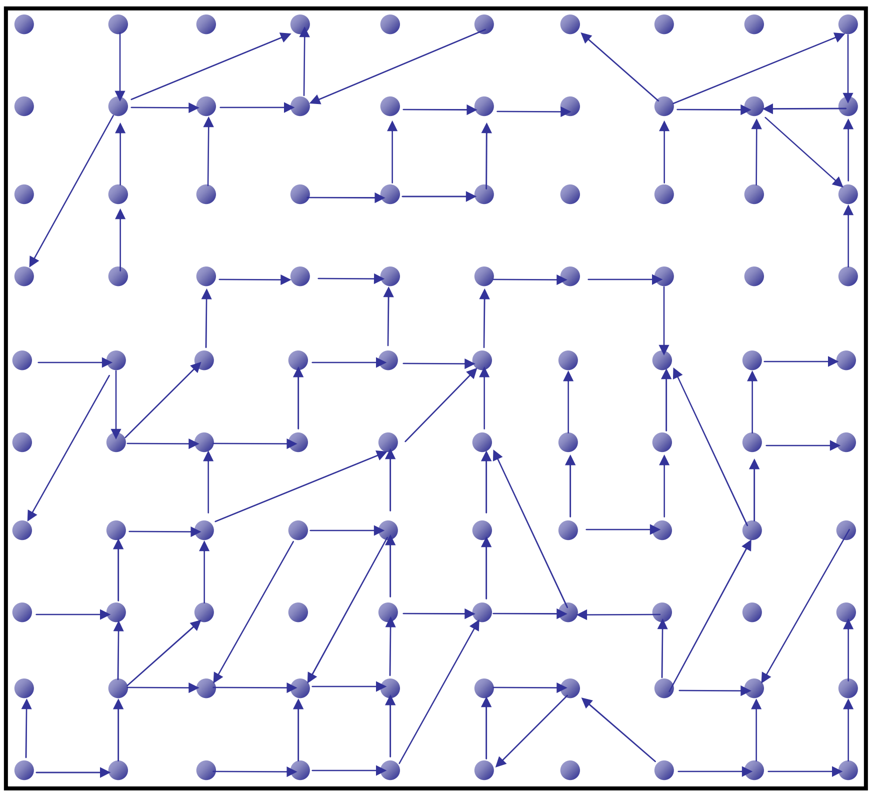
- Undecidable!
  - Equivalent to the halting problem
- Restricted computable versions are prohibitively expensive (PSPACE)
- Why bother ?
  - Just because a problem is undecidable, it doesn't go away!

Running 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;
}
```



What a program *really* is…



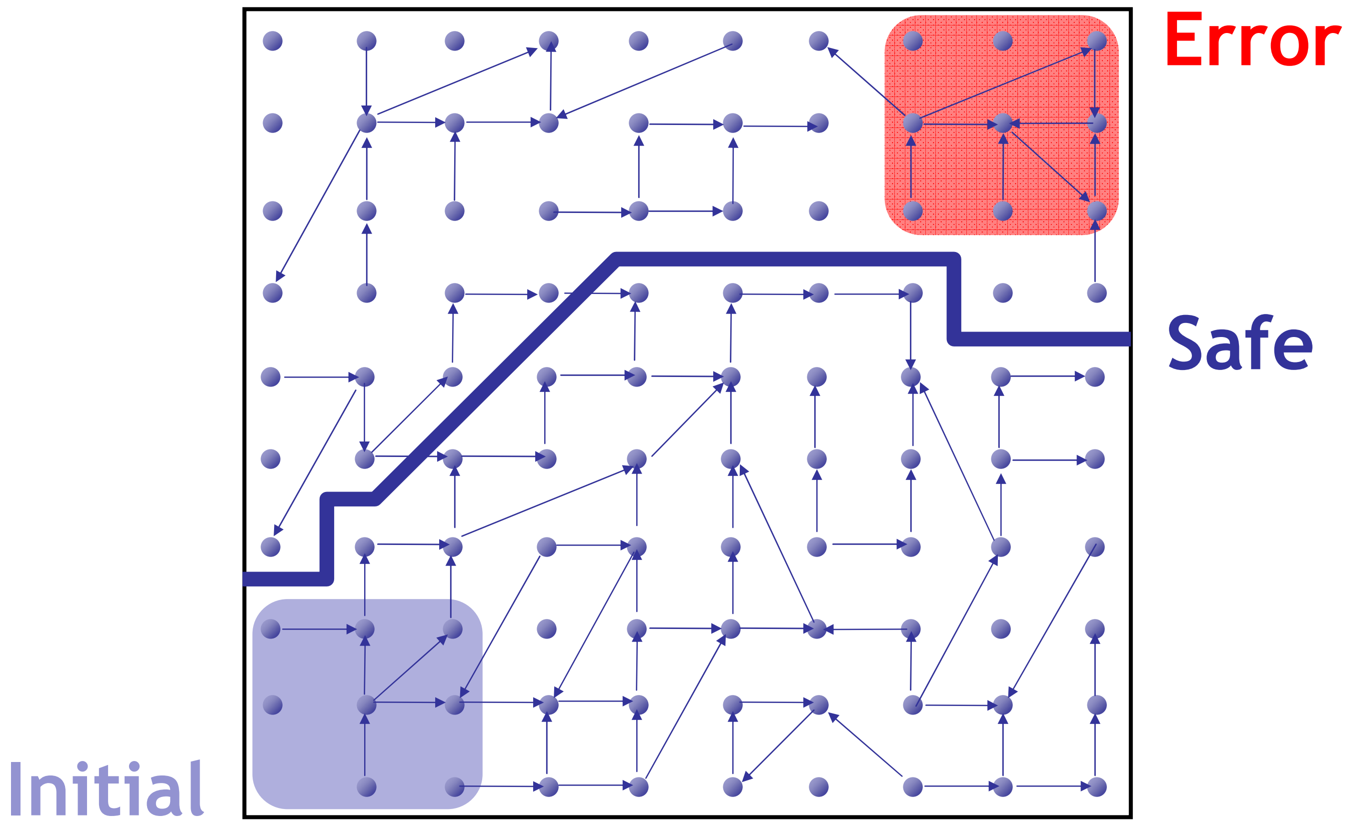
pc	↦ 3
lock	↦ ●
old	↦ 5
new	↦ 5
q	↦ 0x133a

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

pc	↦ 4
lock	↦ ○
old	↦ 5
new	↦ 6
q	↦ 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

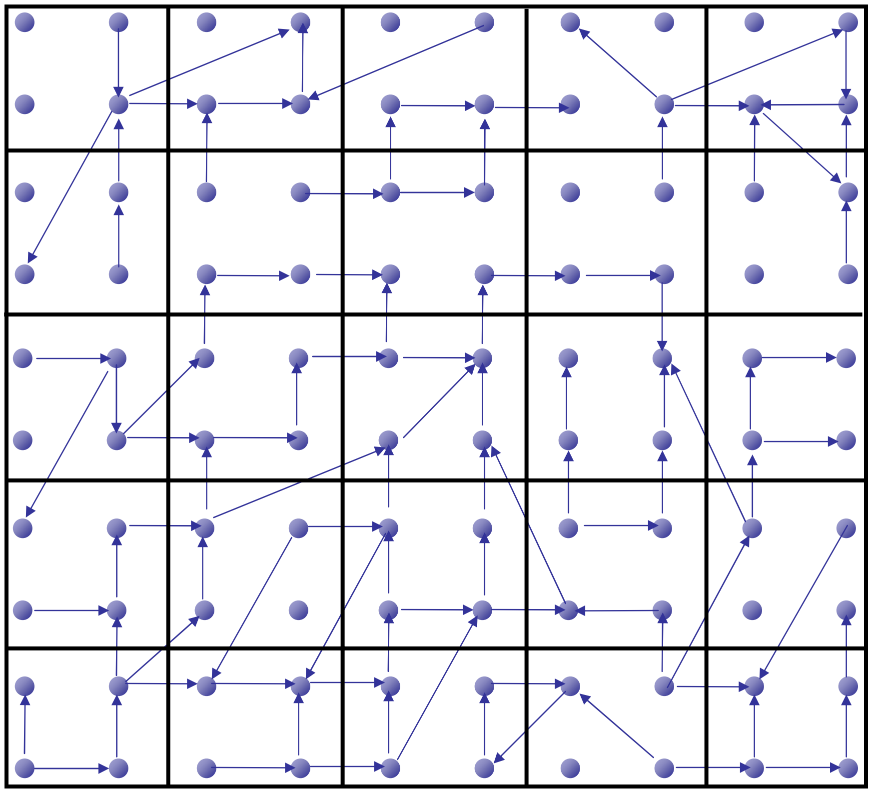
Solution : Set of states $\simeq$ logical formula

Representing 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 implies F_2

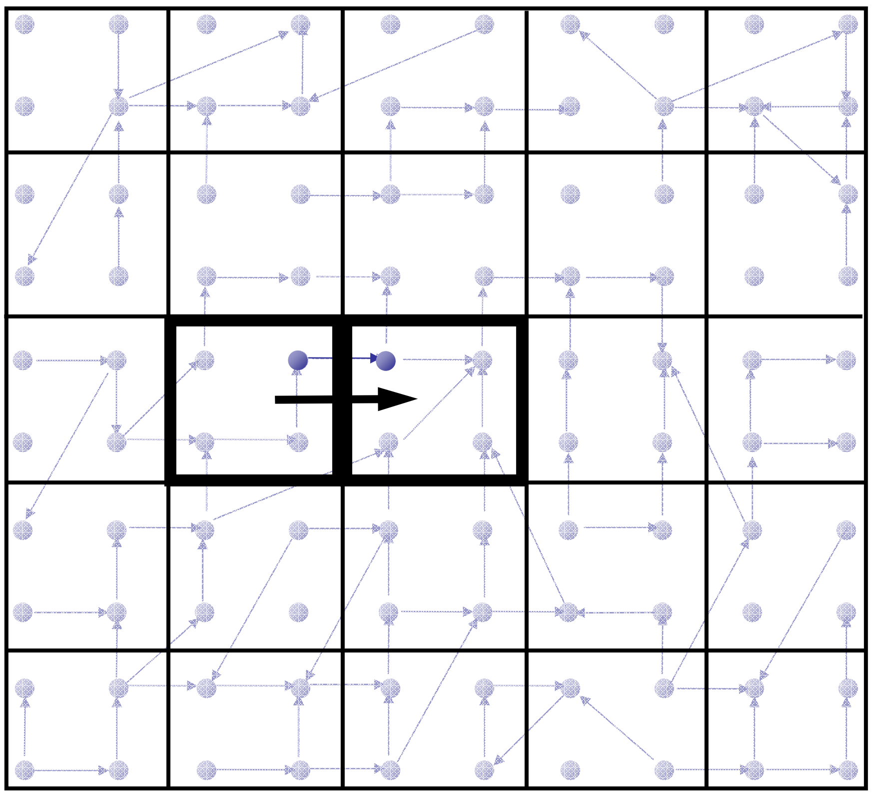
i.e. $F_1 \wedge \neg F_2$ unsatisfiable

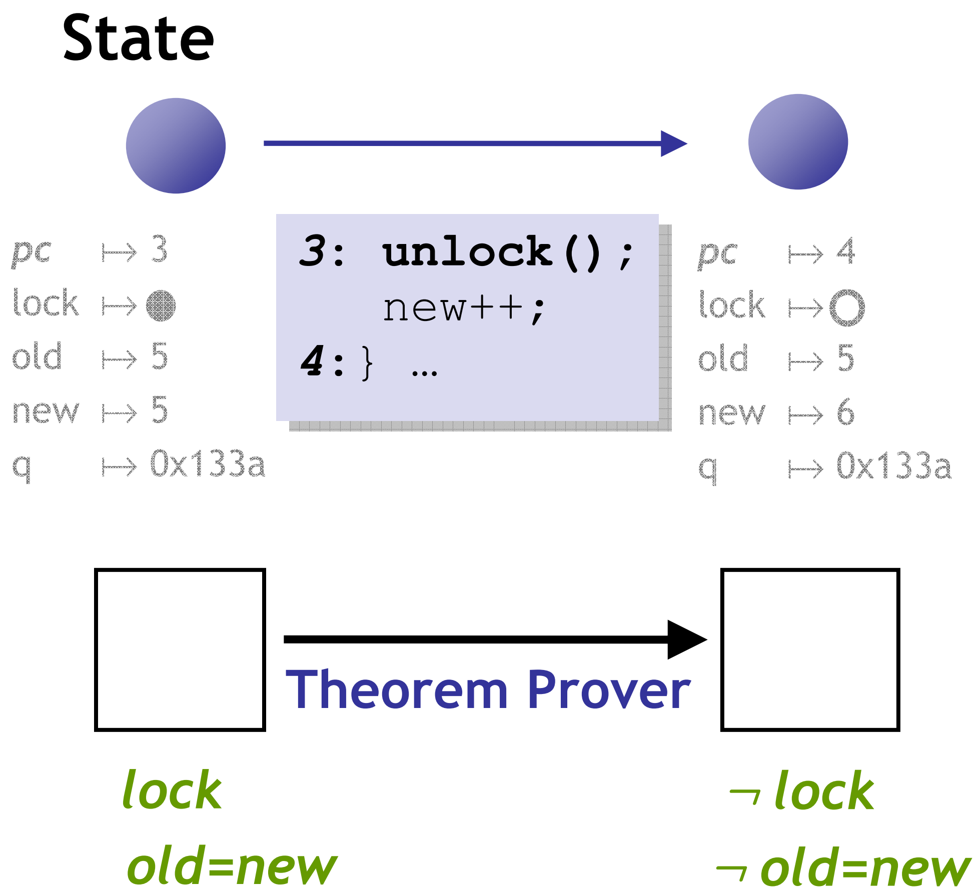
Idea 1: Predicate Abstraction



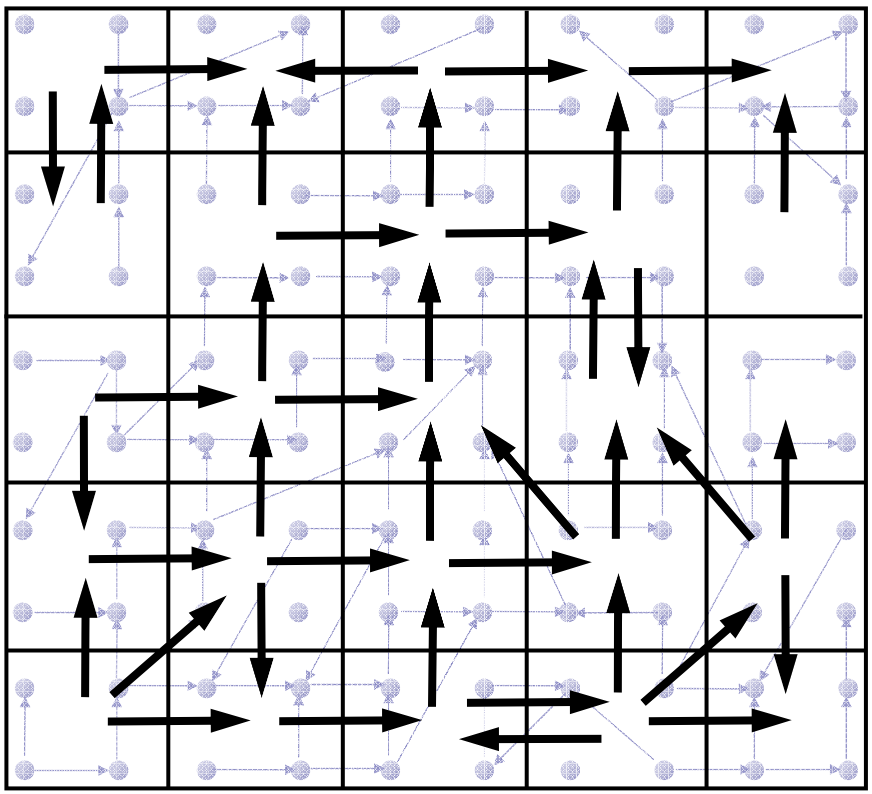
- **Predicates** on program state:
  lock
  old = new
- States satisfying **same** predicates are **equivalent**
  - **Merged** into one **abstract state**
- #abstract states is **finite**

Abstract States and Transitions

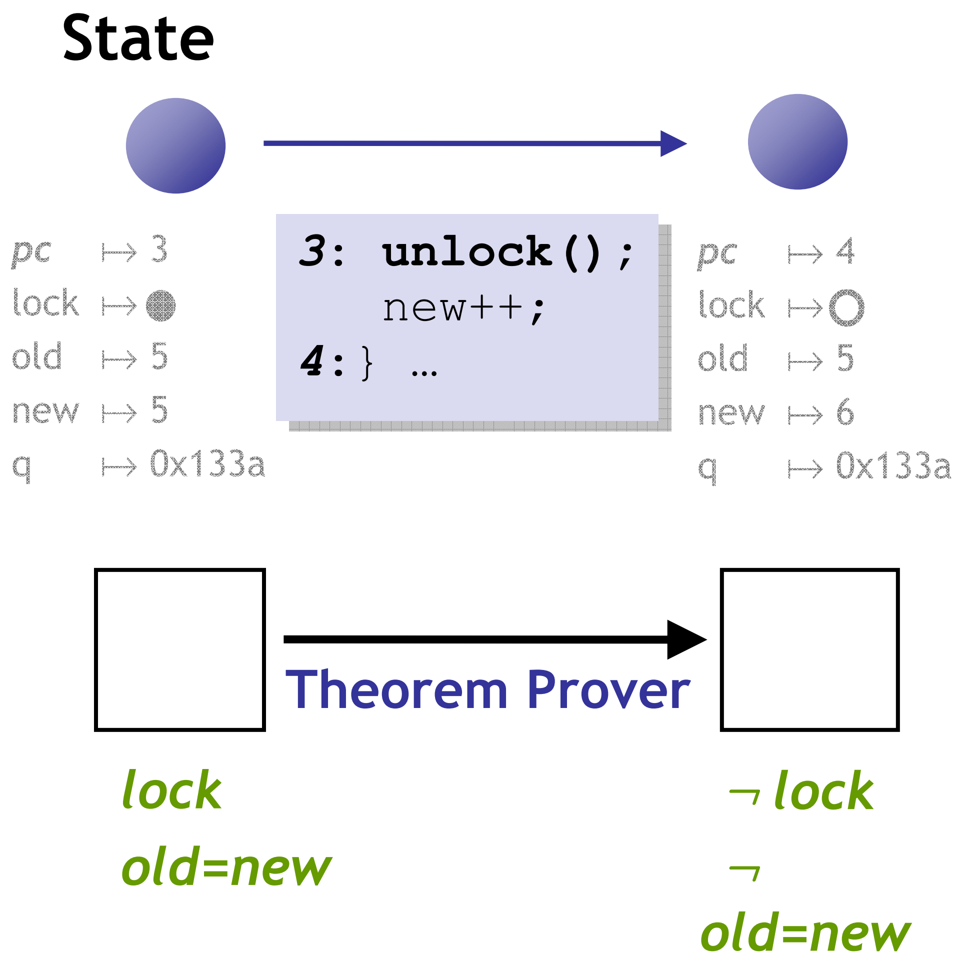




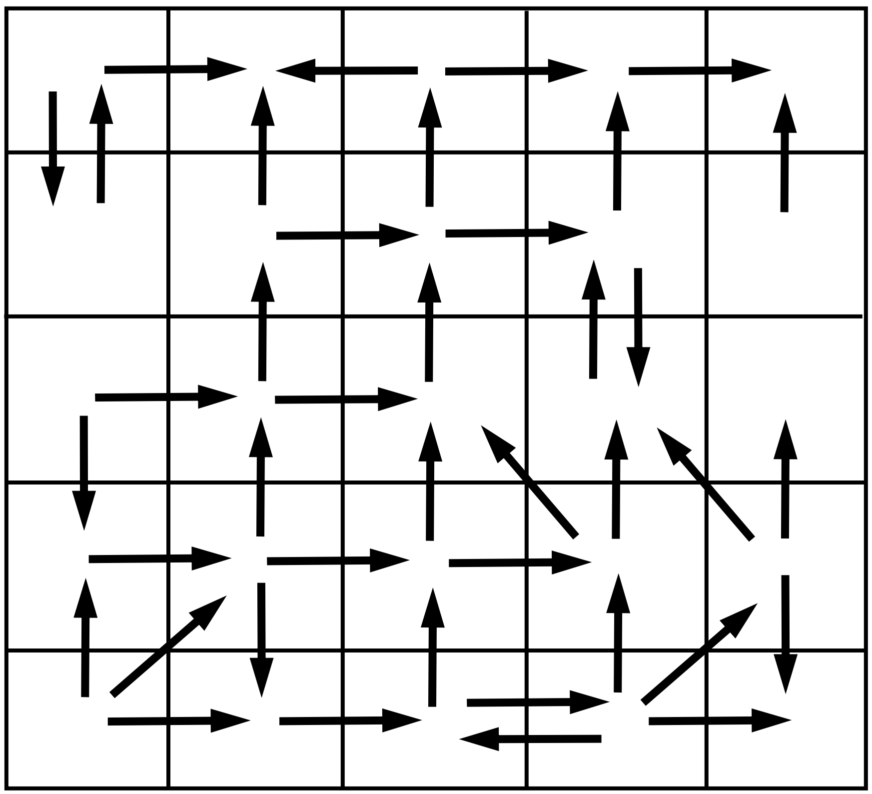
Abstraction

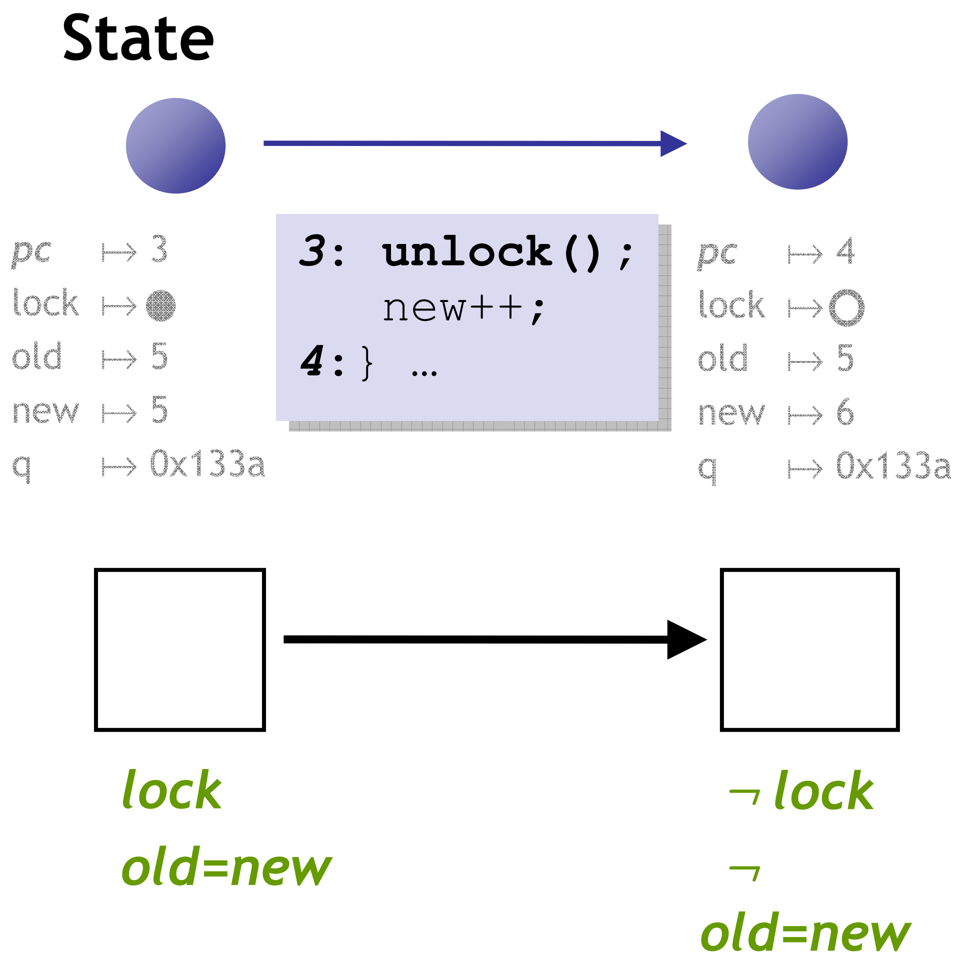


Existential Lifting

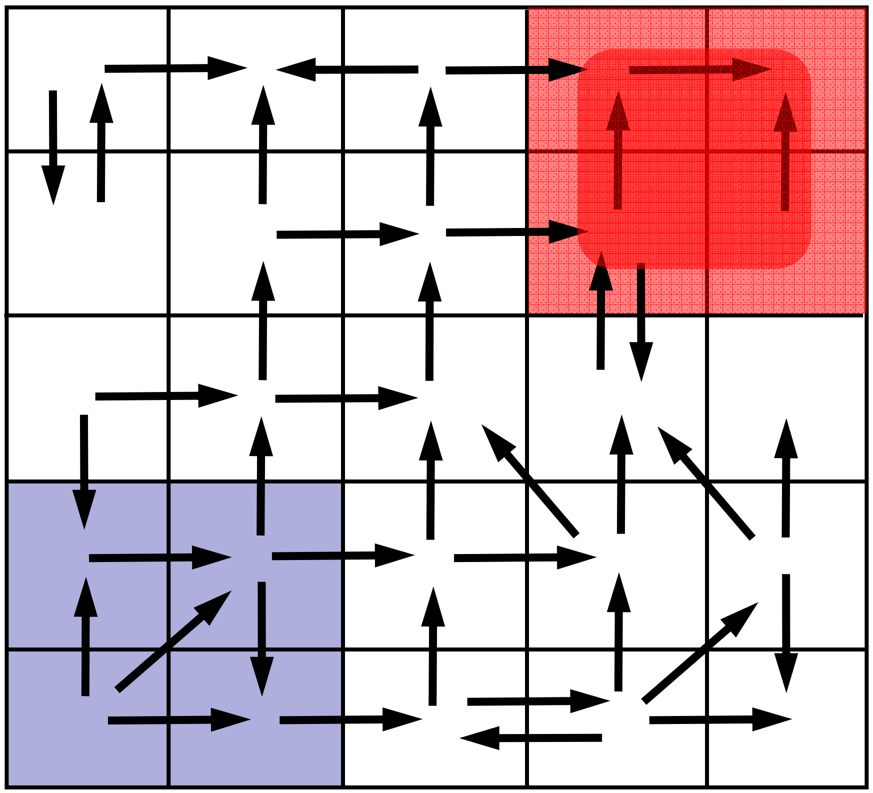


Abstraction





Analyze Abstraction



Analyze finite graph

Over Approximate:

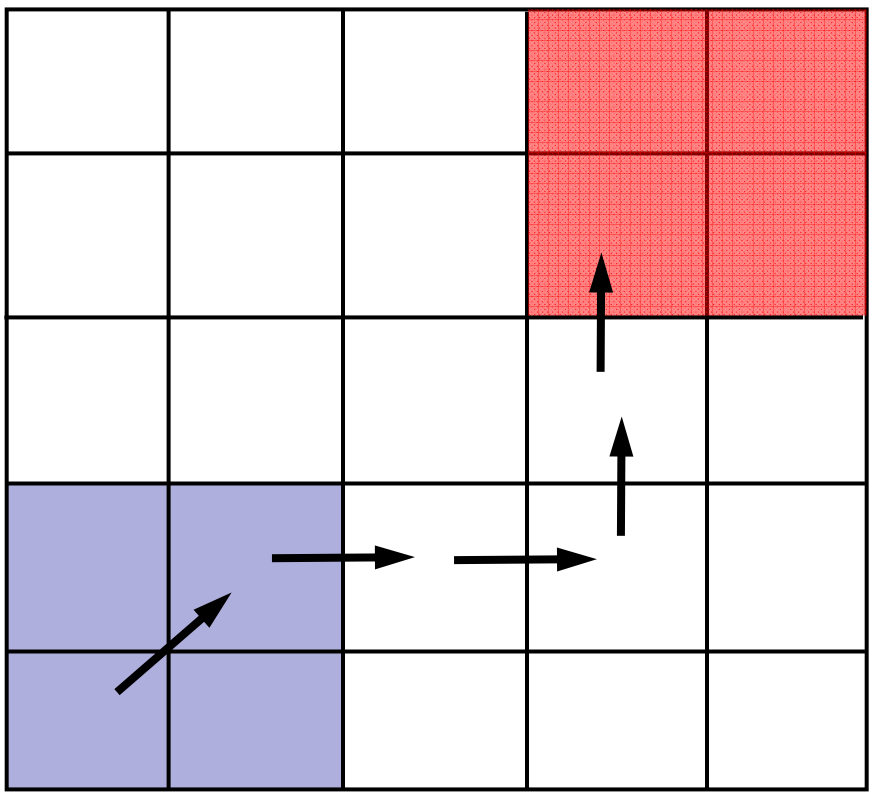
Safe $\Rightarrow$ System Safe

No **false negatives**

Problem

Spurious **counterexamples**

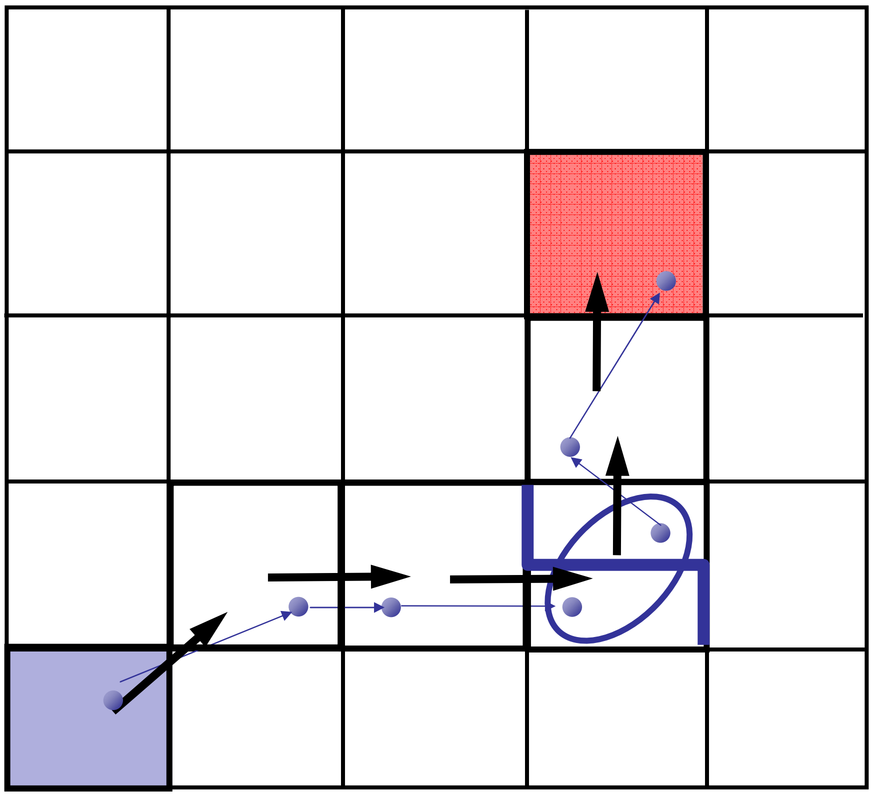
Idea 2: Counterex.-Guided Refinement



Solution

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

Idea 2: Counterex.-Guided Refinement



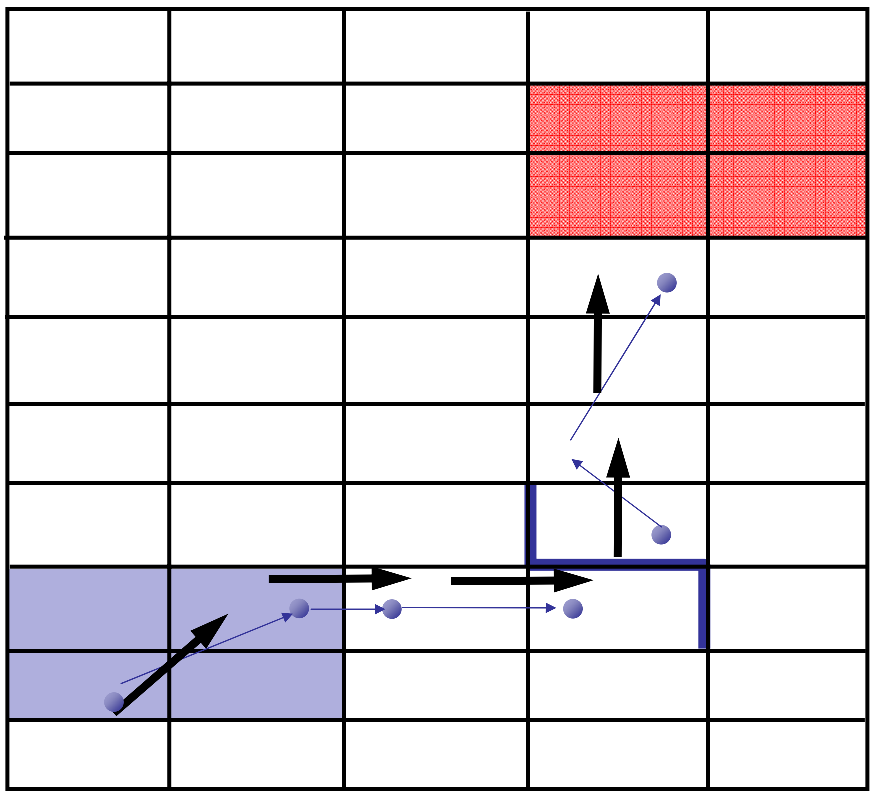
Solution

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



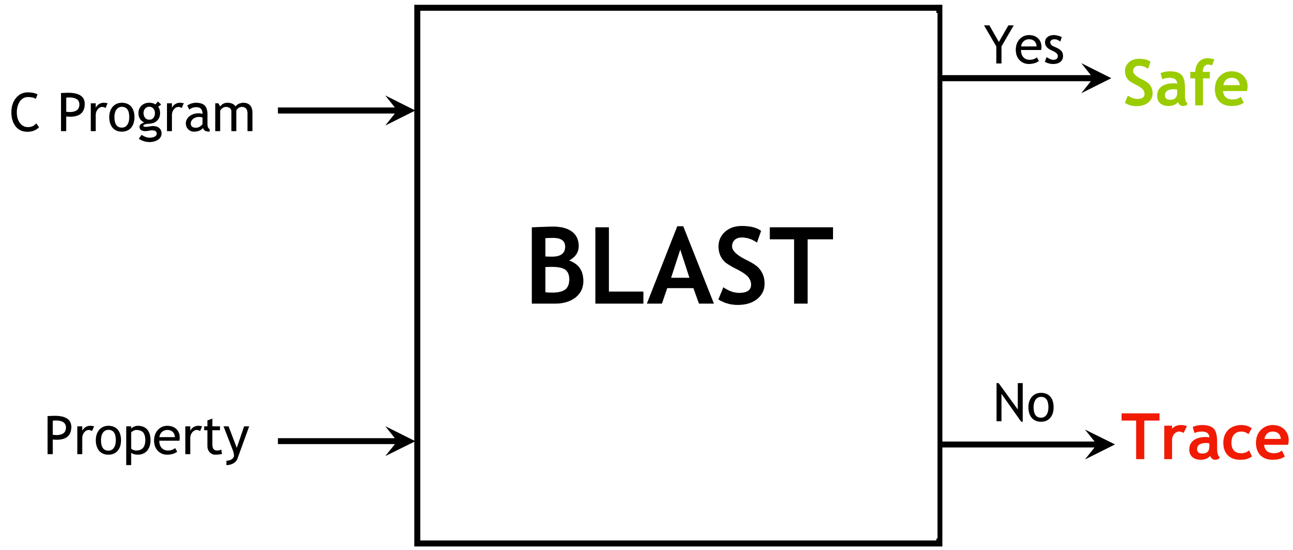
[Kurshan et al 93] [Clarke et al 00]
[Ball-Rajamani 01]

Solution

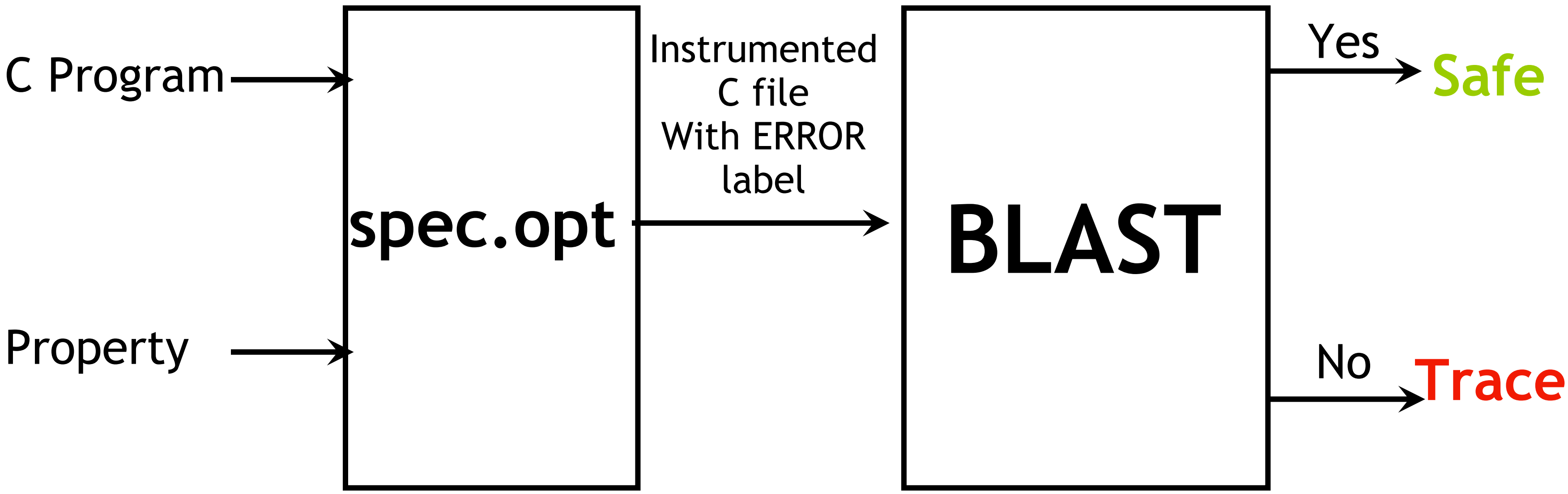
Use spurious **counterexamples** to **refine** abstraction

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

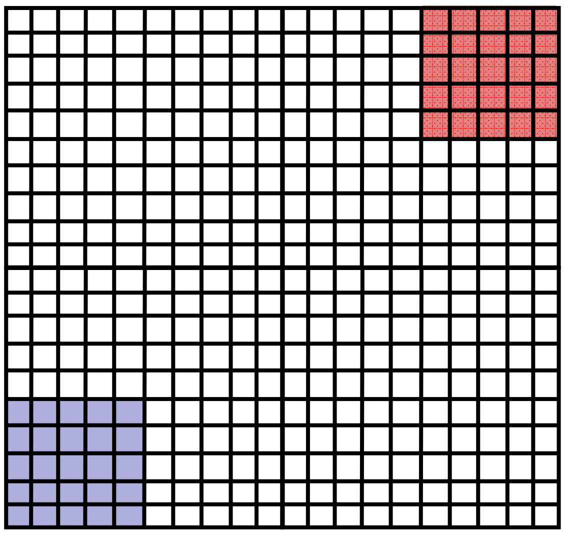
Software Model Checking

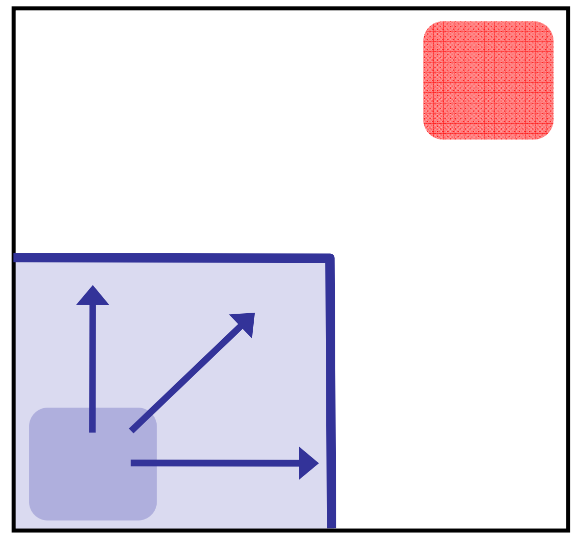


Lazy Abstraction



Problem: Abstraction is Expensive





Reachable

Problem

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

Exponential Thm. Prover queries

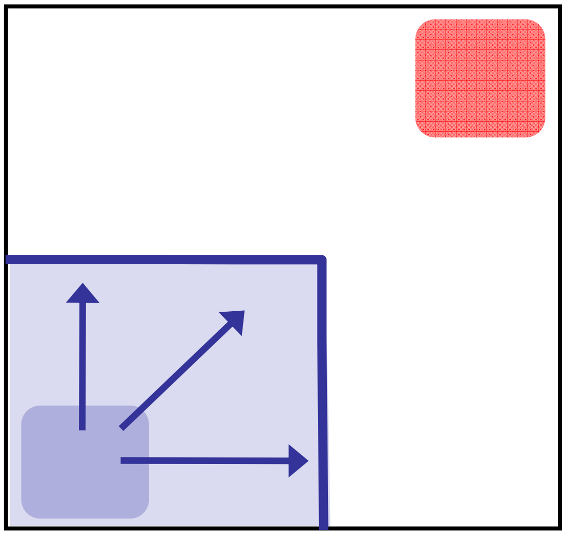
Observe

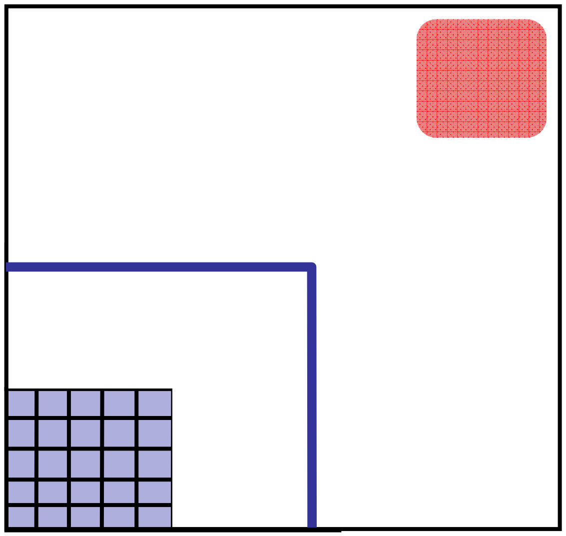
Fraction of state space reachable

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

#Reach ~ 1000's

Solution1: Only Abstract Reachable States





Problem

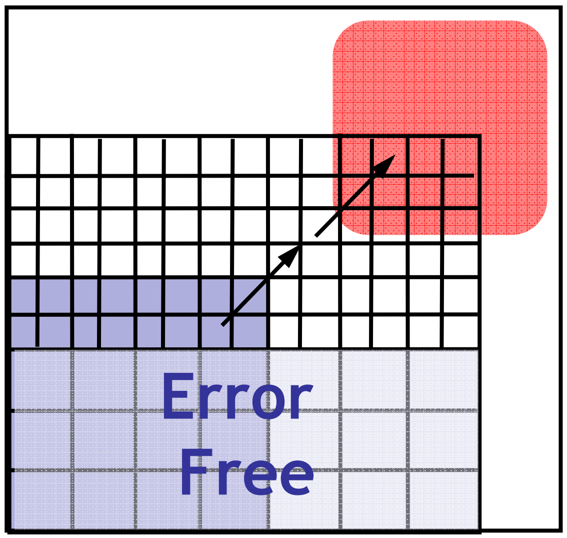
#abstract states = $2^{\#predicates}$

Exponential Thm. Prover queries

Solution

Build abstraction **during** search

Solution2: Don’t Refine Error-Free Regions



Problem

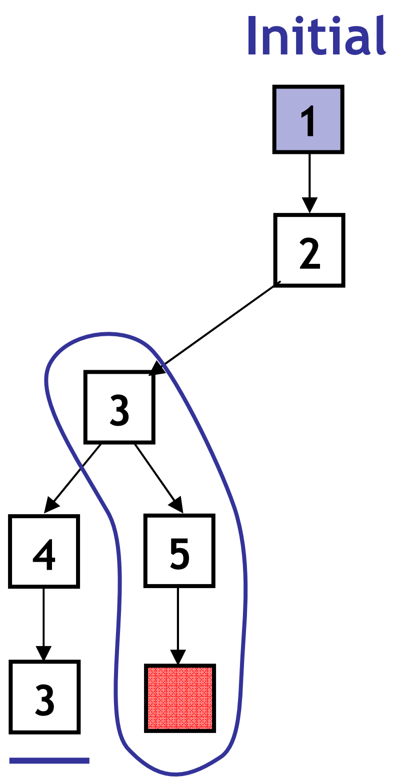
#abstract states = $2^{\#predicates}$

Exponential Thm. Prover queries

Solution

Don’t refine error-free regions

Key Idea: Reachability Tree



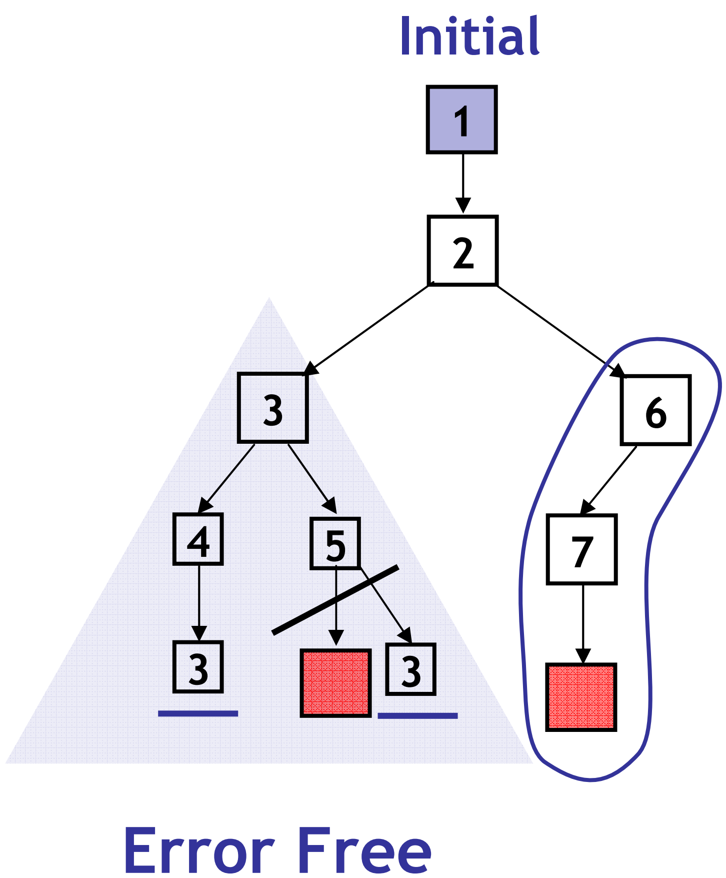
Unroll Abstraction

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



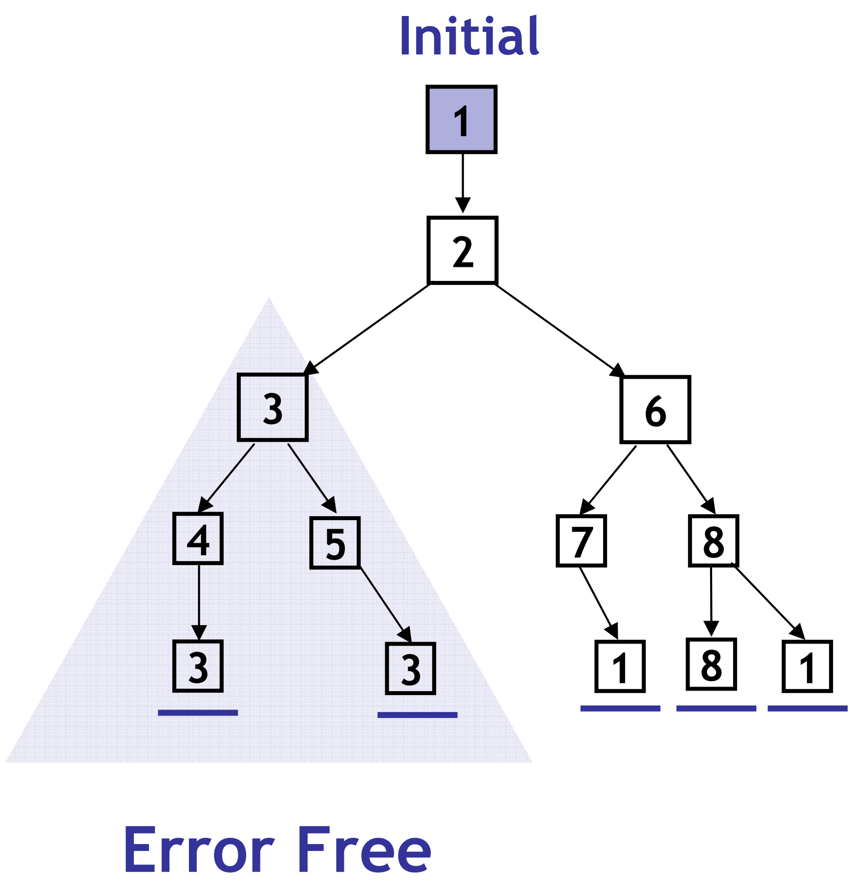
Unroll Abstraction

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



SAFE

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

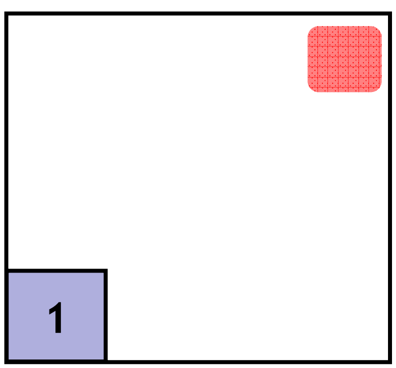
- Learn new predicates
- Rebuild subtree with new preds.

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 ();
}
```

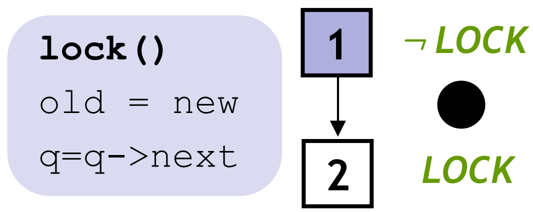


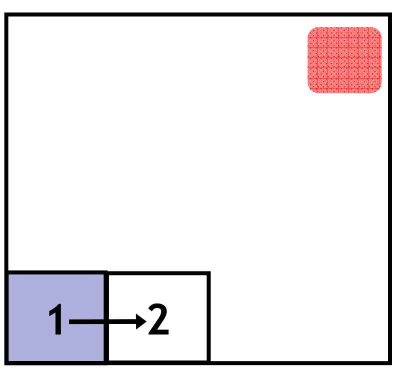
Predicates: *LOCK*

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 ();
}
```



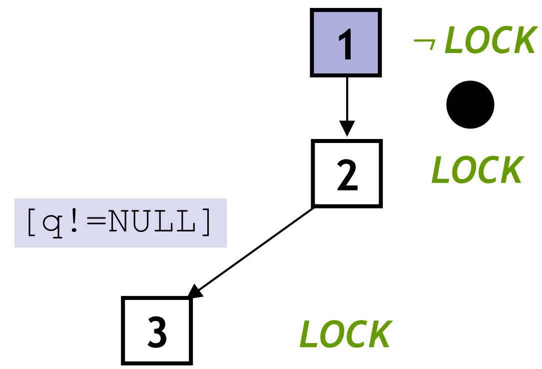


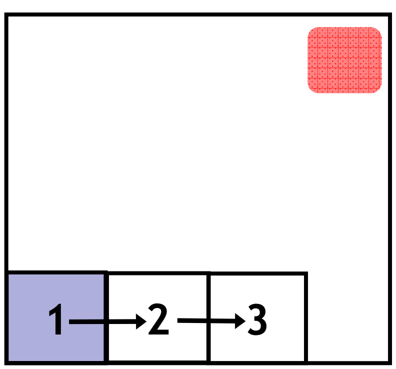
Predicates: *LOCK*

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 ();
}
```



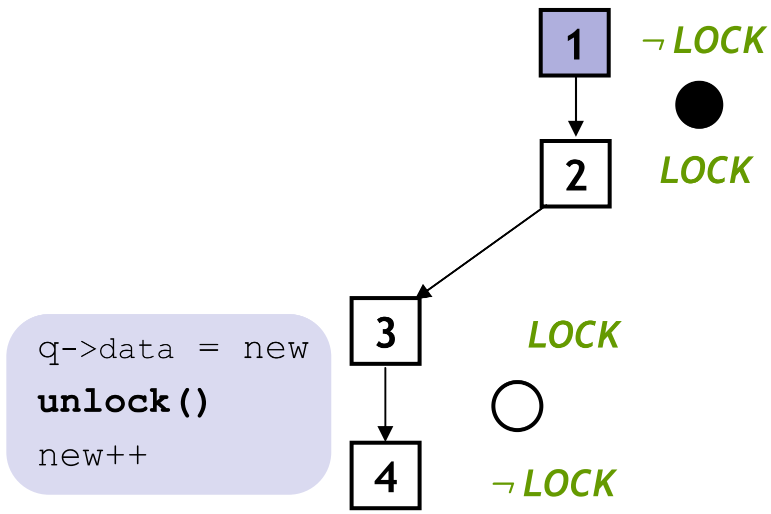


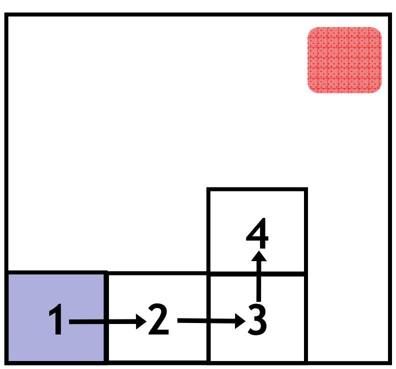
Predicates: *LOCK*

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 ();
}
```



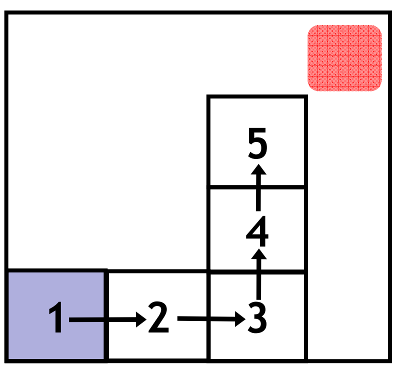


Predicates: *LOCK*

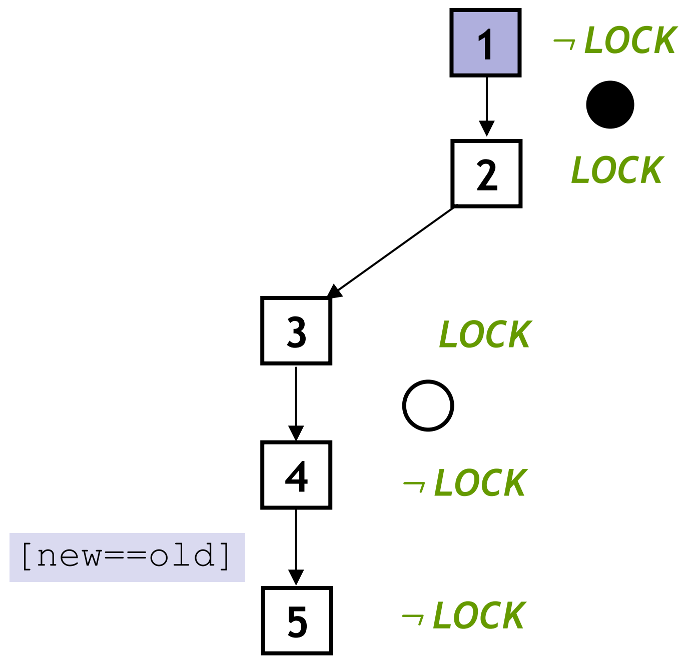
Reachability Tree

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 ();
}
```



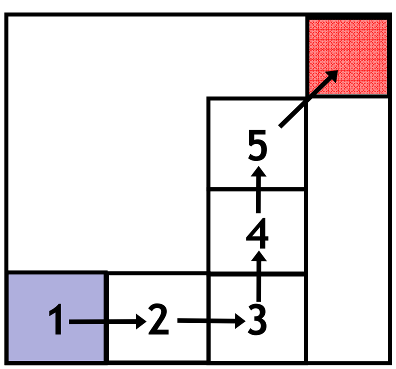
Predicates: *LOCK*



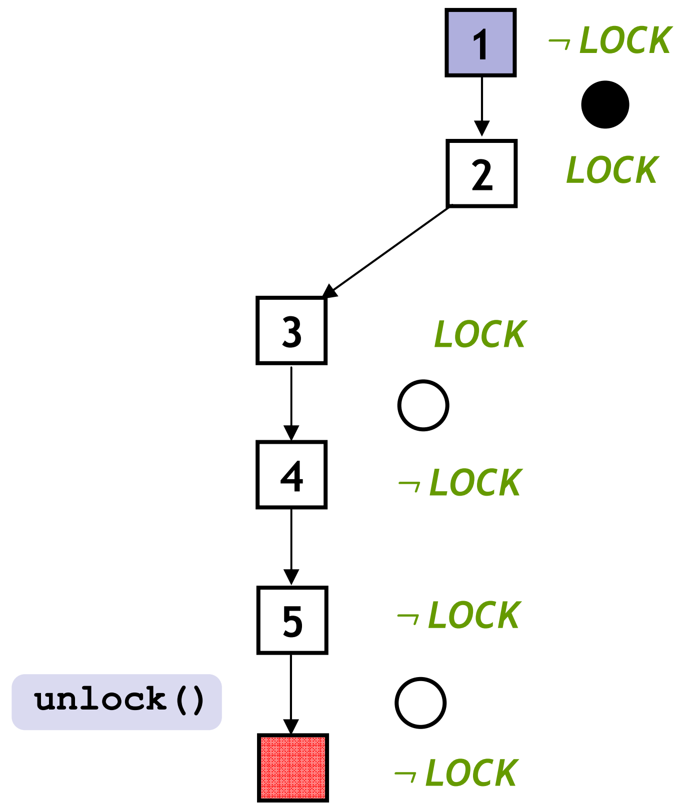
Reachability Tree

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 ();
}
```



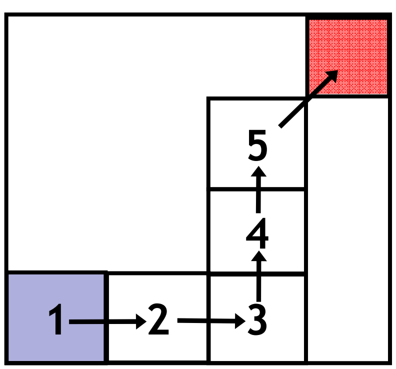
Predicates: *LOCK*



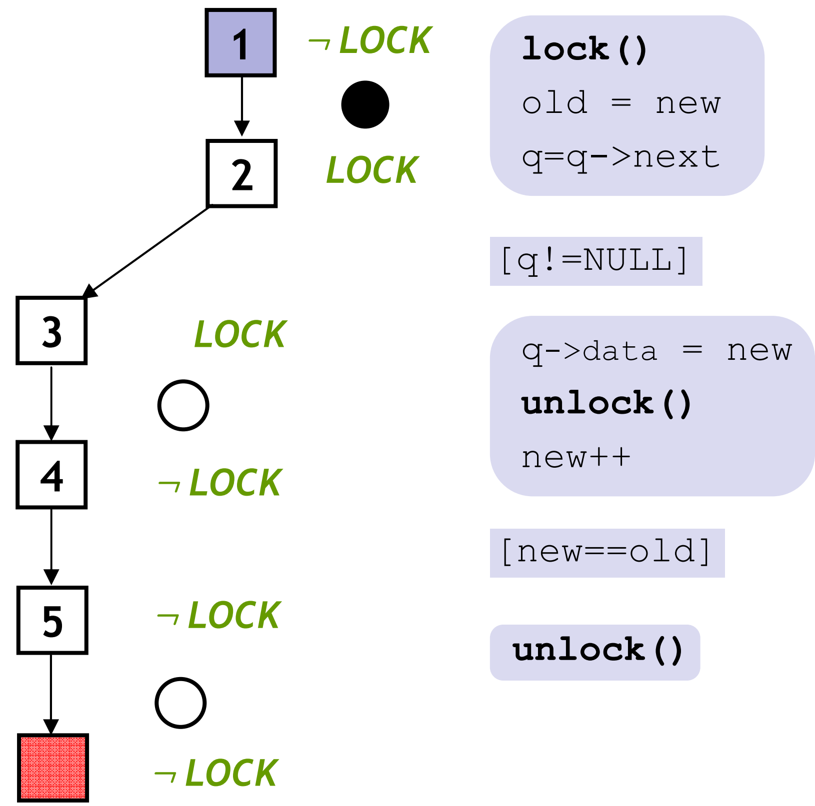
Reachability Tree

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 ();
}
```



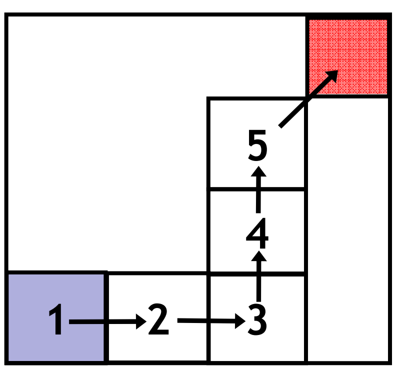
Predicates: *LOCK*



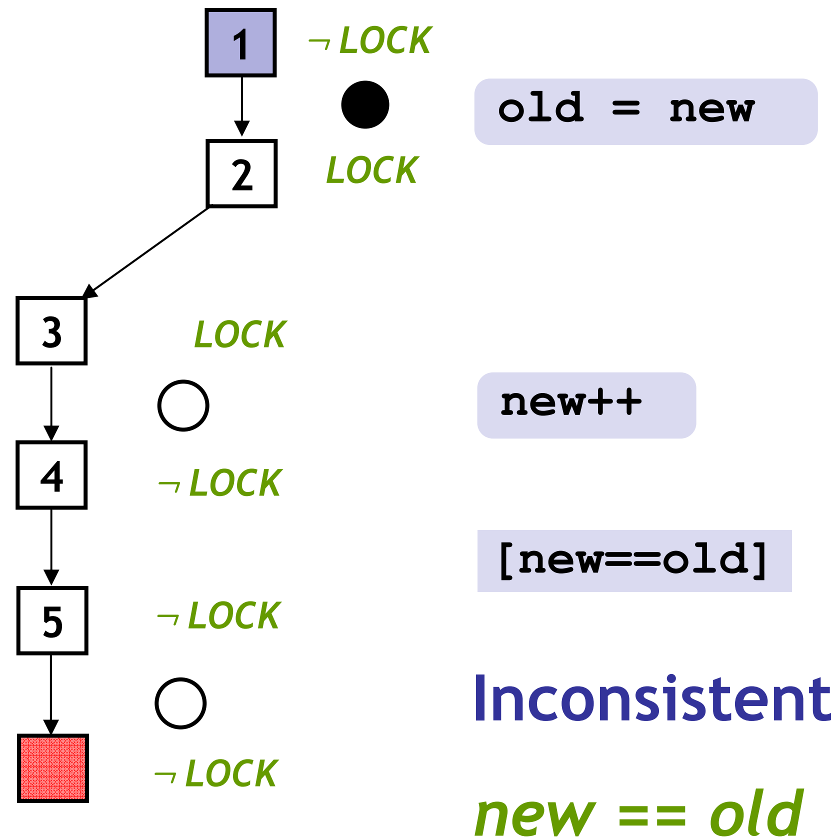
Reachability Tree

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 ();
}
```



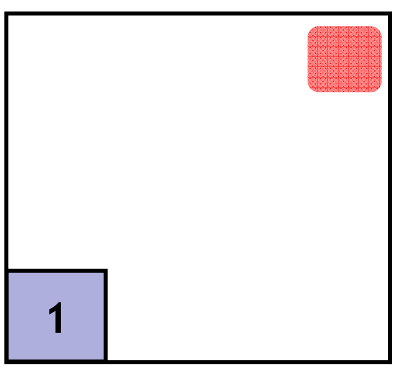
Predicates: *LOCK*



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 ();
}
```

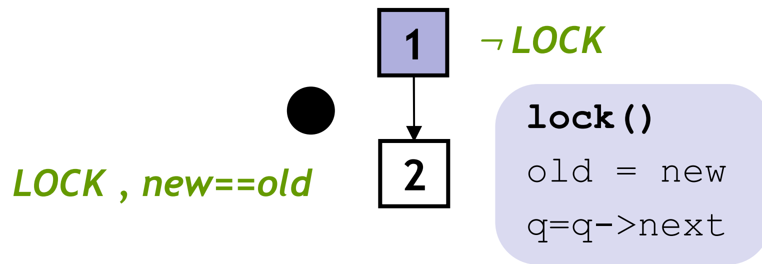


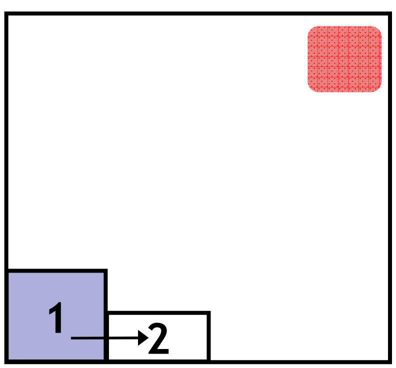
Predicates: *LOCK, new==old*

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 ();
}
```



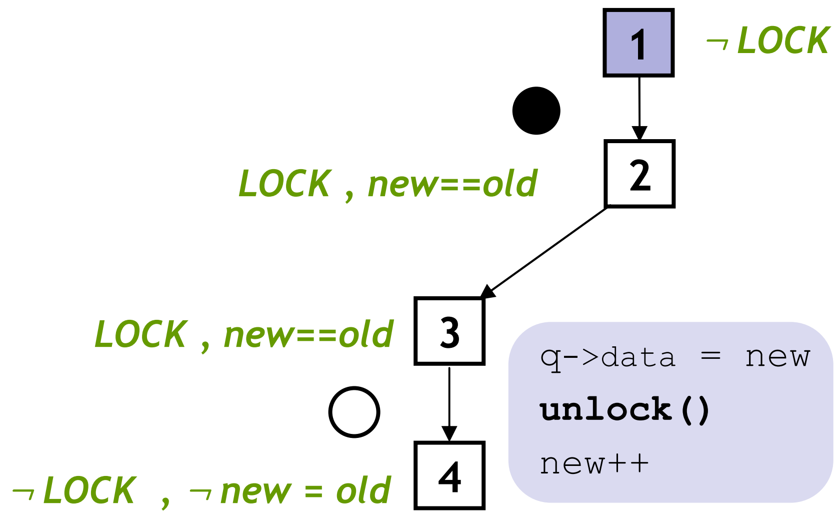


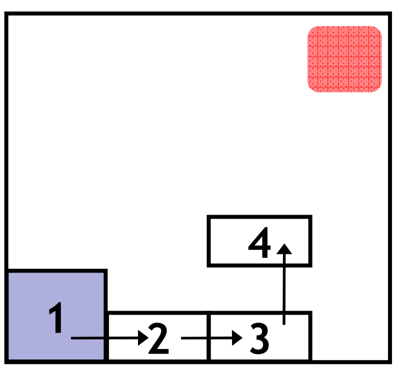
Predicates: *LOCK, new==old*

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 ();
}
```



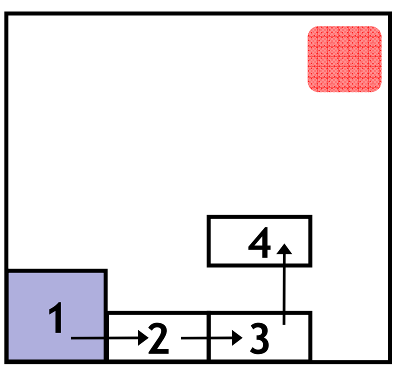


Predicates: *LOCK, new==old*

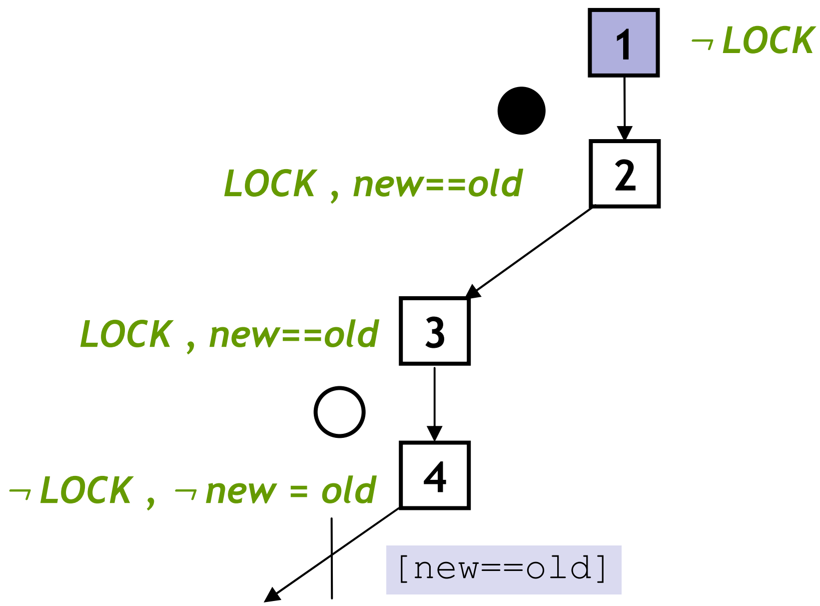
Reachability Tree

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 ();
}
```



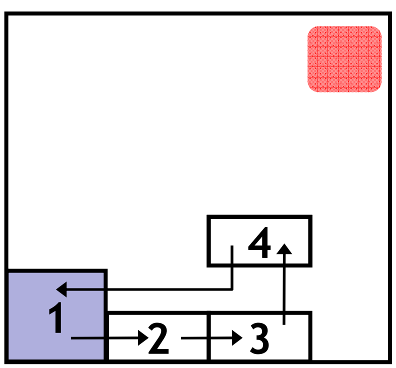
Predicates: *LOCK, new==old*



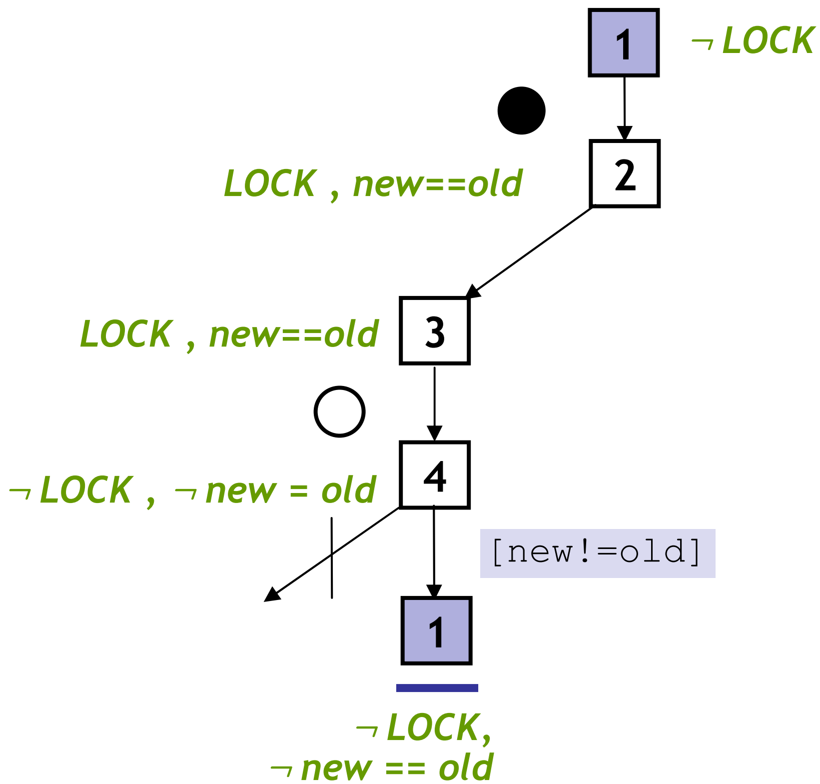
Reachability Tree

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 ();
}
```



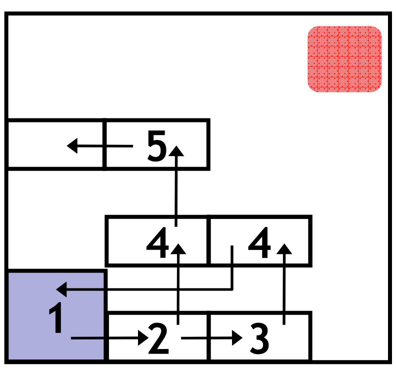
Predicates: *LOCK, new==old*



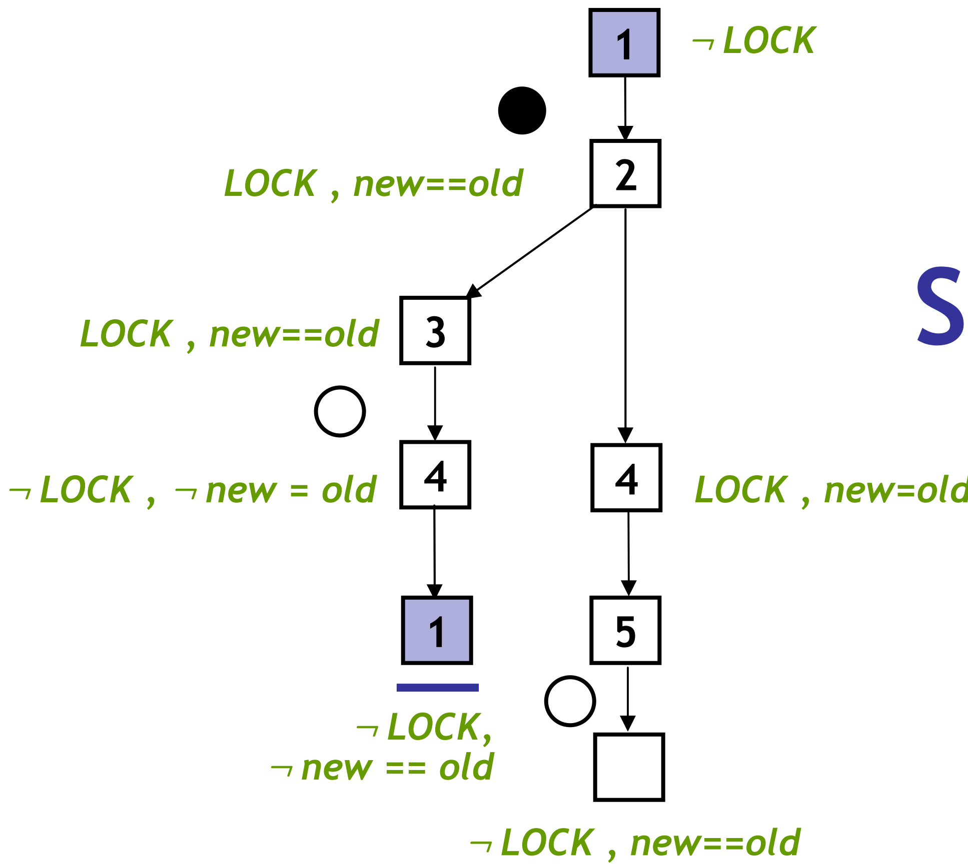
Reachability Tree

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 ();
}
```



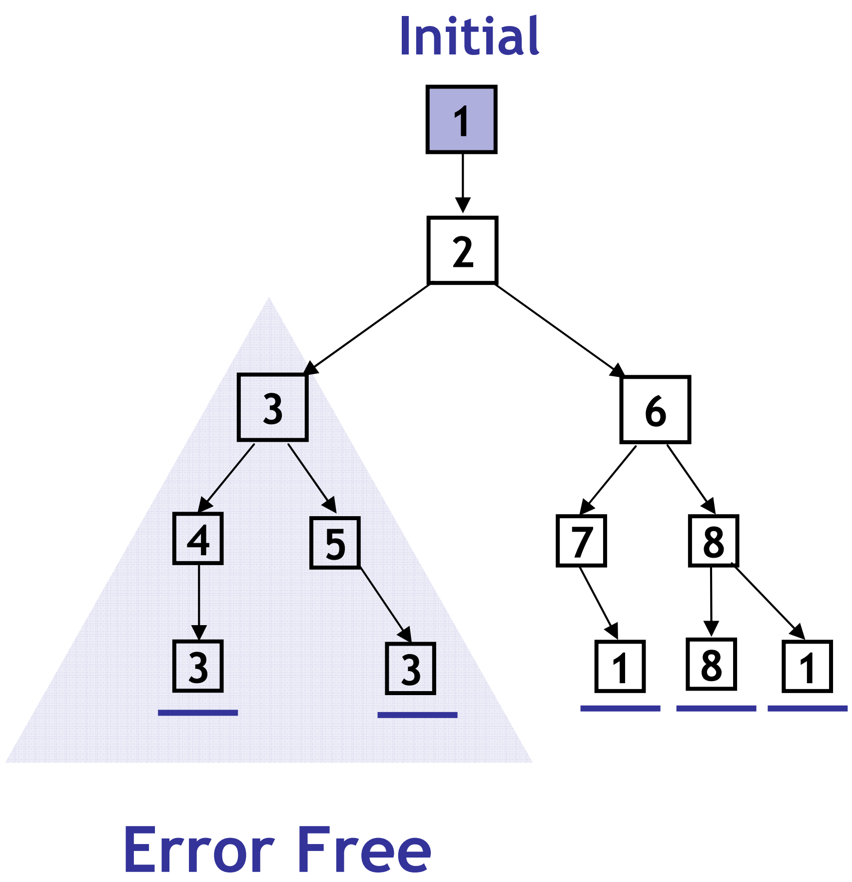
Predicates: *LOCK, new==old*



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

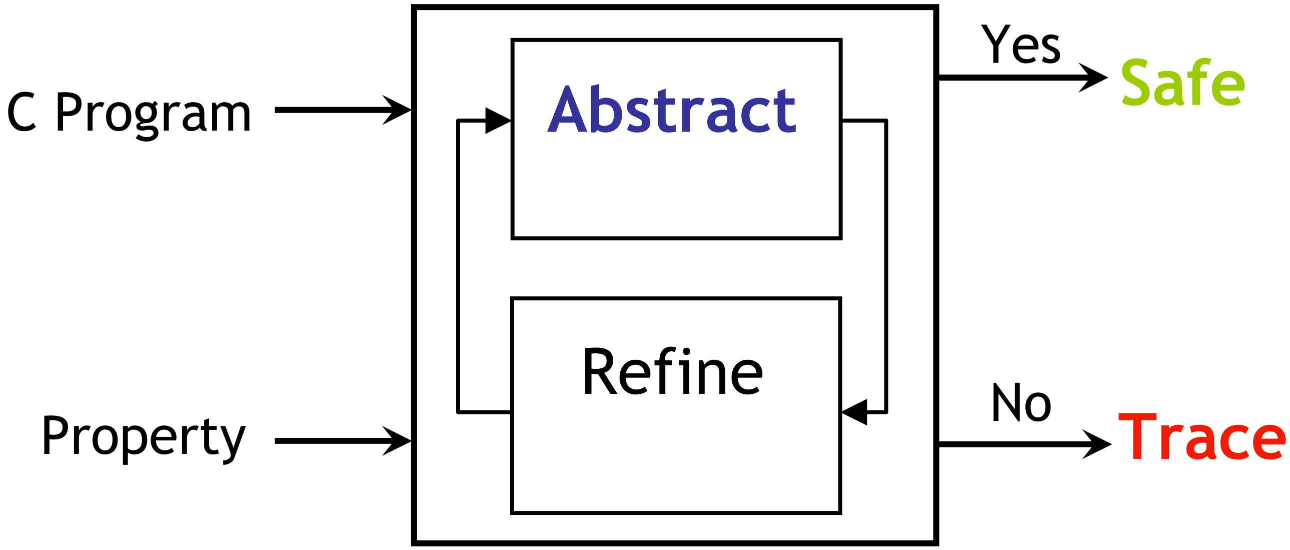
- Learn new predicates
- Rebuild subtree with new preds.

SAFE

S1: Only Abstract Reachable States

S2: Don't refine error-free regions

Lazy Abstraction



Problem: Abstraction is Expensive

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

Key Idea: Reachability Tree

Technical Details

Q. How to find predicates ?

#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

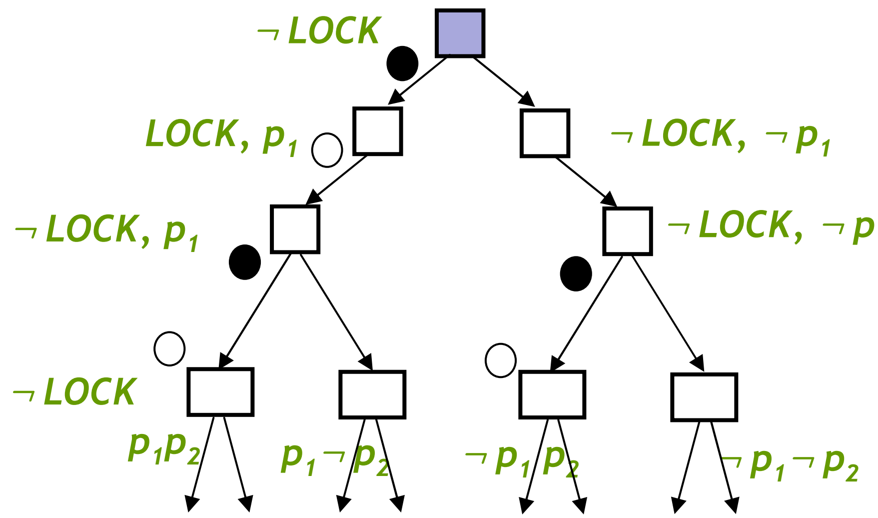
Problem:

$p_1,\ldots,p_n$ needed for verification

Exponential reachable abstract states

#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**

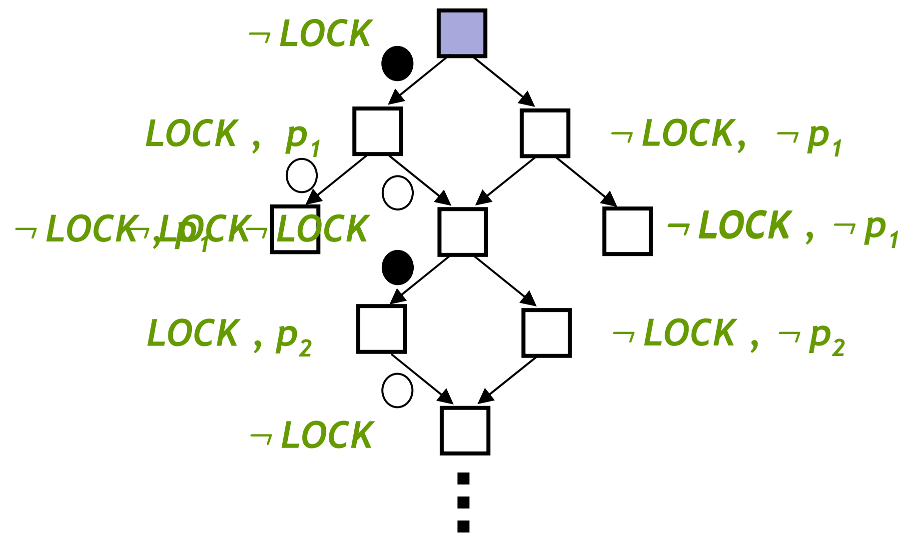
Problem:

$p_1,\ldots,p_n$ needed for verification

Exponential reachable abstract states

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

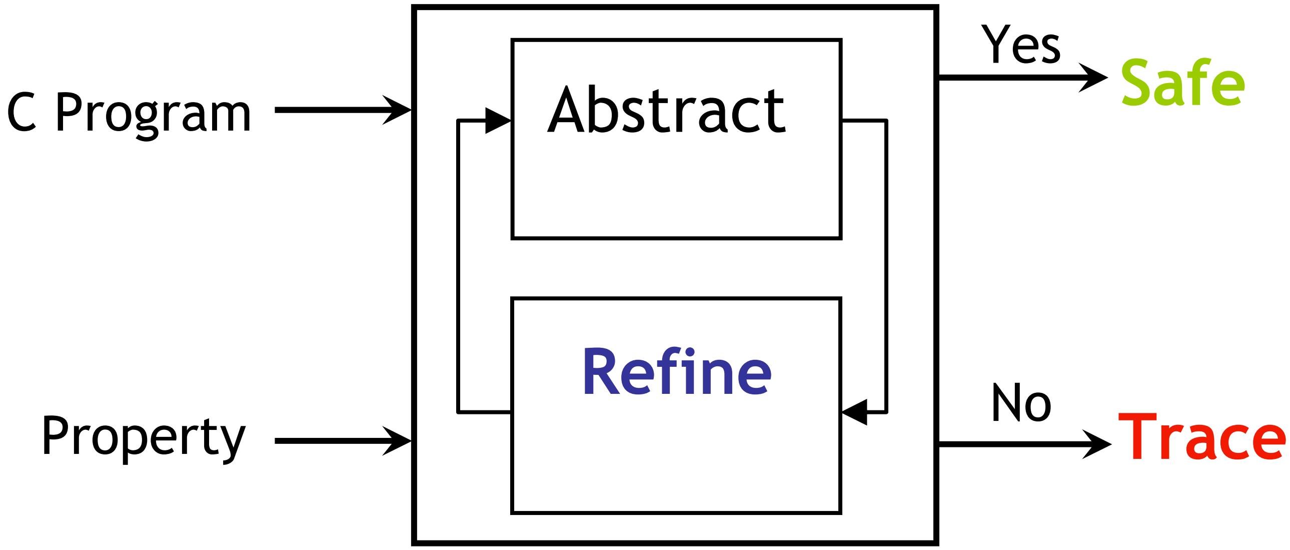
Solution: Use predicates **only where needed**

Using **Counterexamples:**

Q1. Find **predicates**

Q2. Find **where** predicates are needed

Lazy Abstraction



Problem: #Preds grows w/ Program Size

Solution: Localize pred. use, find where preds. needed

Counterexample Traces

```
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)
```

$y = x + 1$

Trace Formulas

Trace	SSA Trace	Trace Feasibility 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)	$\wedge\ y_1 \neq i_0$

Thm: Trace is **feasible** $\Leftrightarrow$ TF is **satisfiable**

The Present State…

Trace

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

… is all the information the executing program has ***here***

State…

1. … after executing trace ***past (prefix)***
2. … knows ***present values*** of variables
3. … makes trace ***future (suffix)*** infeasible

At $\boldsymbol{pc_4}$, which predicate on ***present state*** shows infeasibility of ***future*** ?

What 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)`	$\wedge \quad y_1 \neq i_0$

What 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**

What 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)`	$\wedge\ y_1 \neq i_0$

Relevant Information

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

Predicate …

… implied by TF **prefix**

… on **common** variables

What 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

… & TF **suffix** is **unsatisfiable**

What 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

… & TF **suffix** is **unsatisfiable**

Interpolant = Predicate !

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^+ $\xrightarrow{\text{Interpolate}}$ Φ

$y_1 = x_1 + 1$

Predicate at 4:
$y = x + 1$

Craig Interpolant
[Craig 57]

Computable from Proof of Unsat
[Krajicek 97] [Pudlak 97]

Predicate …

… implied by TF prefix

… on common variables

… & TF suffix is unsatisfiable

Interpolant = Predicate !

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)`	$\wedge \quad y_1 \neq i_0$

Predicate at 4:
$y = x+1$

ψ^-

Interpolate

Φ

ψ^+

$y_1 = x_1 + 1$

Q. How to compute interpolants ? ...

Craig Interpolant
[Craig 57]

Computable from
Proof of Unsat
[Krajicek 97] [Pudlak 97]

Predicate ...

... implied by TF prefix

... on common variables

... & 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 $x_1 = ctr_0$

- Cut + Interpolate at **each** point
- Pred. Map: $pc_i \mapsto$ Interpolant from cut i

Building Predicate Maps

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

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

Interpolate $\Rightarrow$ $x_1 = ctr_1 - 1$

- Cut + Interpolate at **each** point
- Pred. Map: $pc_i \mapsto$ 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^-

Interpolate

ψ^+

$y_1 = x_1 + 1$

- Cut + Interpolate at **each** point
- Pred. Map: $pc_i \mapsto$ 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^-

ψ^+

Interpolate

$y_1 = i_0$

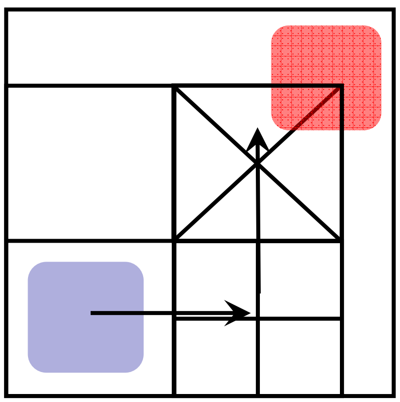
- Cut + Interpolate at **each** point
- Pred. Map: $\mathbf{pc_i} \mapsto$ Interpolant from cut i

Local Predicate Use

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

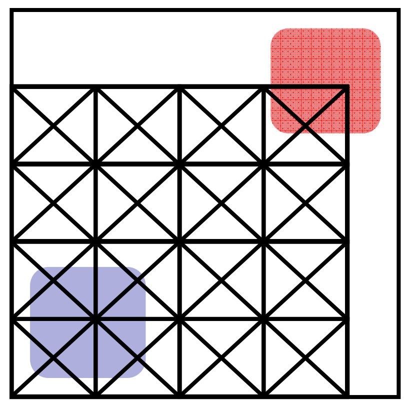
- #Preds. grows with program size
- #**Preds per location** small

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



Local Predicate use

Ex: 2n states



Global Predicate use

Ex: 2^n states

Property3:

IRP Handler

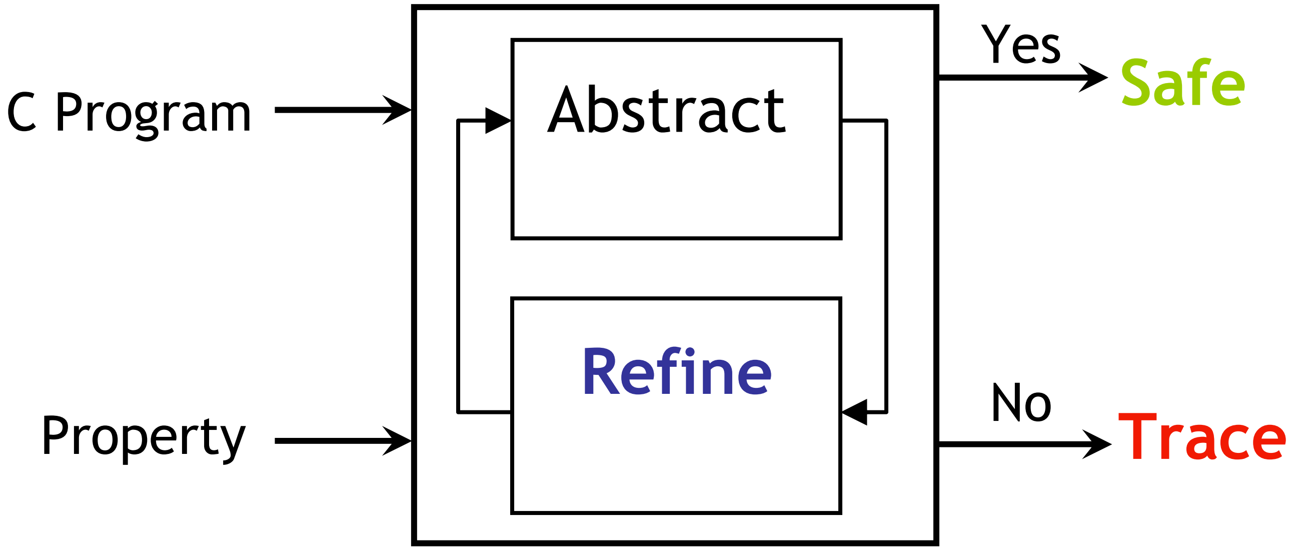
Win NT DDK

Localizing

Program	*Lines**	*Previous Time(mins)*	*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*

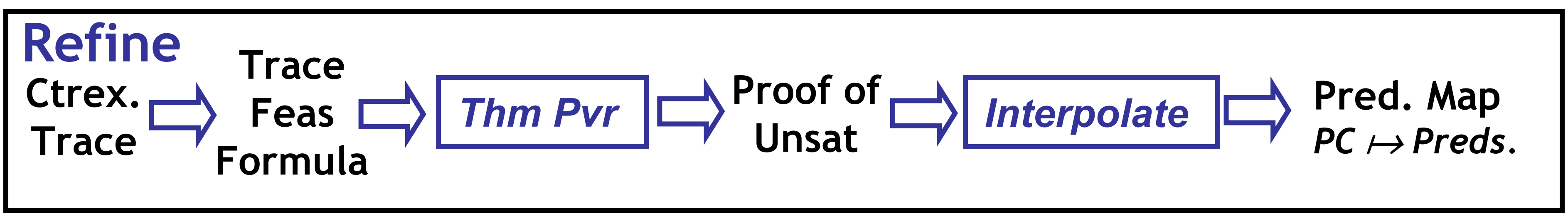
* Pre-processed

Lazy Abstraction



Problem: #Preds grows w/ Program Size

Solution: Localize pred. use, find where preds. needed



Lazy Abstraction: Summary

- Predicates:
  - Abstract infinite program states
- Counterexample-guided Refinement:
  - Find predicates tailored to prog, property

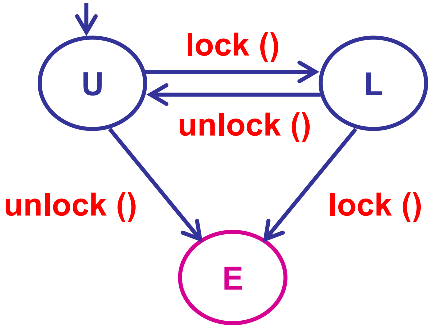
1. **Abstraction** : Expensive
   Reachability Tree
2. **Refinement** : Find predicates, use locations
   Proof of unsat of TF + Interpolation

The BLAST Query Language

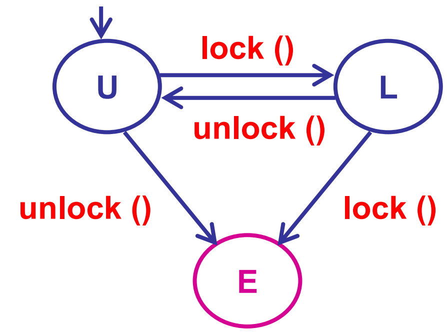
1. (Possibly Infinite-State) ***Monitor Automata*** for Reachability Queries over Program Locations
2. First-Order Imperative ***Scripting Language*** for Combining Relations over Program Locations

Two-State Locking Monitor

```
GLOBAL int locked;

EVENT {
        PATTERN { init() }
        ACTION { locked = 0; }
}

EVENT {
        PATTERN { lock() }
        ASSERT { locked == 0 }
        ACTION { locked = 1; }
}

EVENT {
        PATTERN { unlock() }
        ASSERT { locked == 1 }
        ACTION { locked = 0; }
}
```



Two-State Locking Monitor




```
GLOBAL int locked;

EVENT {
        PATTERN { init() }
        ACTION { locked = 0; }
}

EVENT {
        PATTERN { lock() }
        ASSERT { locked == 0 }
        ACTION { locked = 1; }
}

EVENT {
        PATTERN { unlock() }
        ASSERT { locked == 1 }
        ACTION { locked = 0; }
}
```

else REJECT

else REJECT

Single-Lock Safety Analysis

```
source(l1) := LOC_LABEL(l1,"START");

target(l2) := TRUE(l2);

error-traces(l1,l2) := REJECT(source,target,monitor);

error-locs(l2) := EXISTS(l1,error-traces(l1,l2));

PRINT "The following locations are reachable and cause
      a locking error:";

PRINT error-locs(l2);
```

`REJECT(l1,l2,monitor)` is the set of all location pairs **`(l1,l2)`** such that there is a feasible program trace from **`l1`** to **`l2`** which is rejected by the automaton **`monitor`**.

Type-State Locking Monitor

```
SHADOW lock_t { int locked; }

EVENT {
      PATTERN { init($1) }
      ACTION { $1->locked = 0; }
}

EVENT {
      PATTERN { lock($1) }
      ASSERT { $1->locked == 0 }
      ACTION { $1->locked = 1; }
         }

EVENT {
      PATTERN { unlock($1) }
      ASSERT { $1->locked == 1 }
      ACTION { $1->locked = 0; }
}
```

Dead-Code Analysis

```
source(l1) := LOC_LABEL(l1,"START");
target(l2) := TRUE(l2);
feasible-traces(l1,l2) := ACCEPT(source,target,EMPTY);
reached-locs(l2) := feasible-traces(_,l2));
PRINT "The following locations are not reachable:";
PRINT !reached-locs(l2);
```

Impact Analysis

```
GLOBAL int defined;

INITIAL { defined = 0; }

EVENT {
      PATTERN { j = $1; }
      ACTION { defined ++ ; }
}

FINAL { defined == 1 }
```

else REJECT

```
affected(l1,l2) :=
ACCEPT(LOC_LHS(l1,"j"),LOC_RHS(l2,"j"),monitor);

PRINT affected(l1,l2);
```

Benefits of Two-Level Specifications

1. Separates properties from programs, while keeping a familiar syntax for writing properties
2. Treats a program as a database of facts that can be queried, and supports macros for traditional temporal-logic specifications
3. Supports the formulation of decomposition strategies for verification tasks
4. Supports the incremental maintenance of properties during program development

The BLAST Two-Level Query Language

1. (Possibly Infinite-State) ***Monitor Automata*** for Reachability Queries over Program Locations:

   checked by the BLAST model checking engine

2. First-Order Imperative ***Scripting Language*** for Combining Relations over Program Locations:

   checked by the CrocoPat relational query engine [Beyer, Noack, Lewerentz]