The software model checker BLAST

Dirk Beyer, Thomas A. Henzinger,

Ranjit Jhala, Rupak Majumdar

Presented by Yunho Kim

Overview

- Predicate abstraction is successfully applied to software model checking
  - Infinite concrete states → finite abstract states
  - Tools: SLAM(MSR), BLAST(UCB), SATABS(CMU)
- Cost for abstraction is still too high
  - O($2^{\# \text{ preds}}$) abstract states
  - We need to abstract and refine locally, not globally
- Blast proposed
  - Lazy abstraction
  - Craig interpolation-based refinement

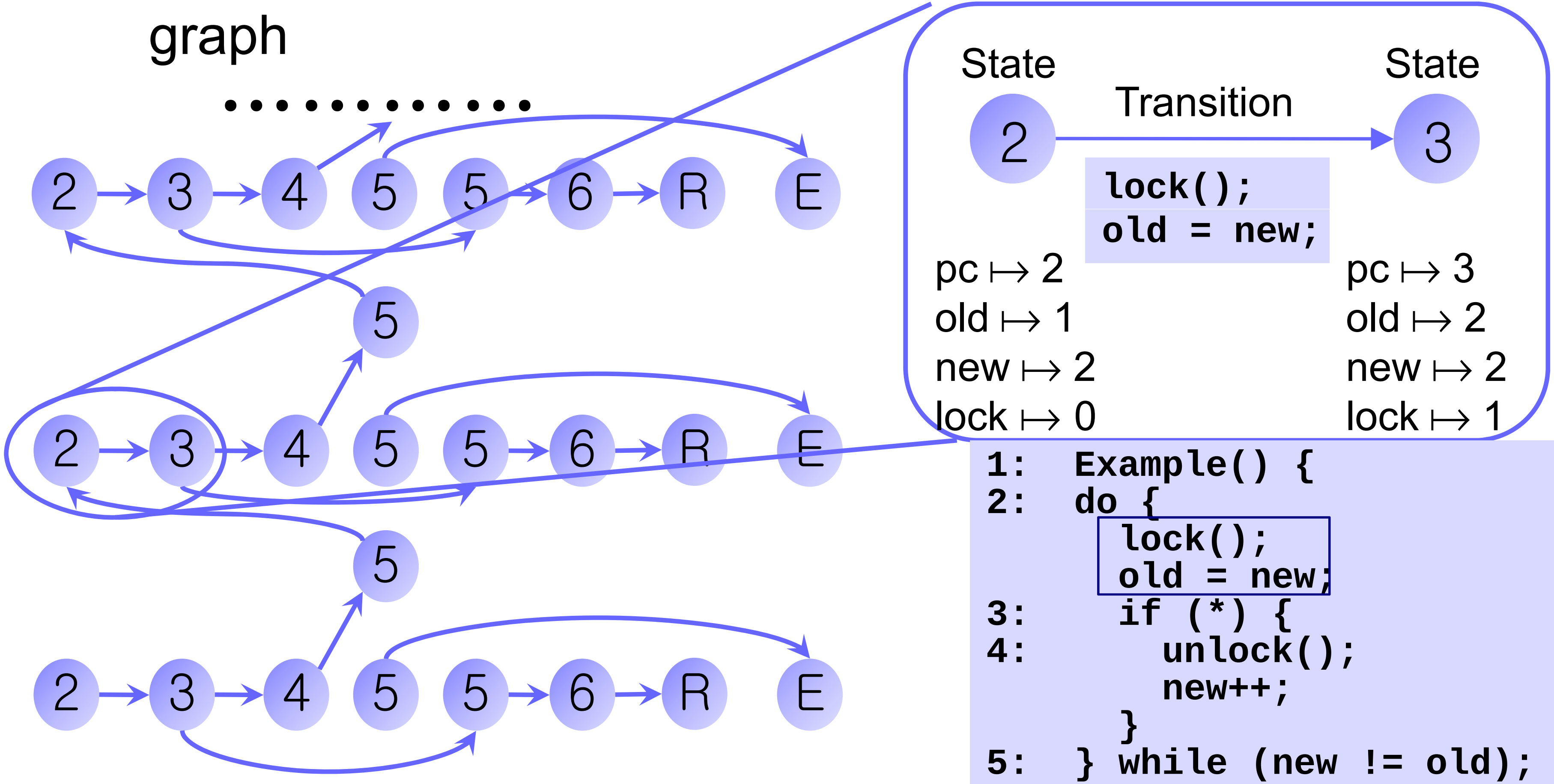
Contents

- Part I. Software Model Checking
  - Program behavior
  - Predicate abstraction
  - Counterexample-guided abstraction refinement

- Part II. BLAST
  - Abstraction and model checking
  - Craig interpolation-based refinement

Behavior of program

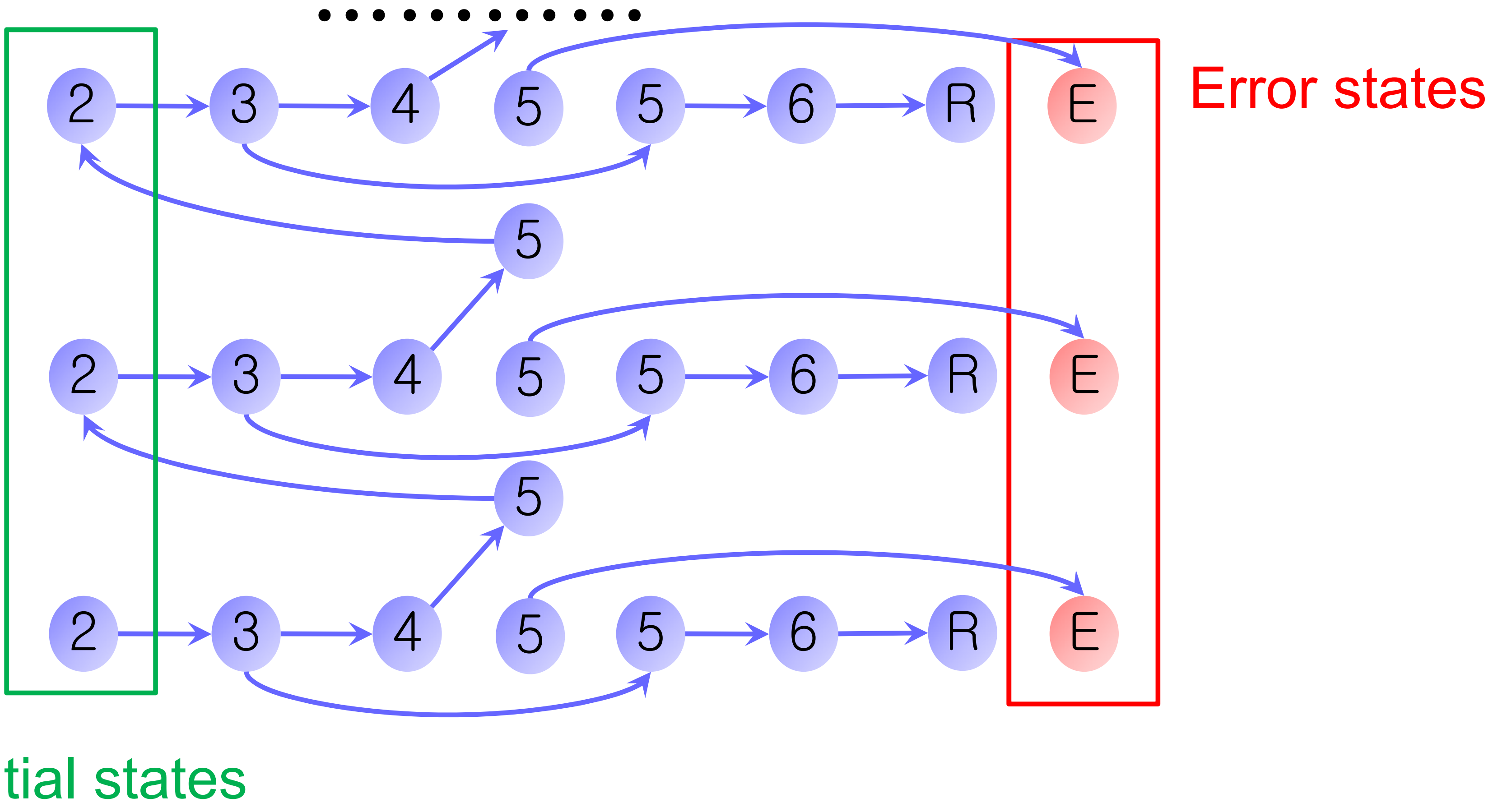
- Behavior of program can be modeled as a state transition graph



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

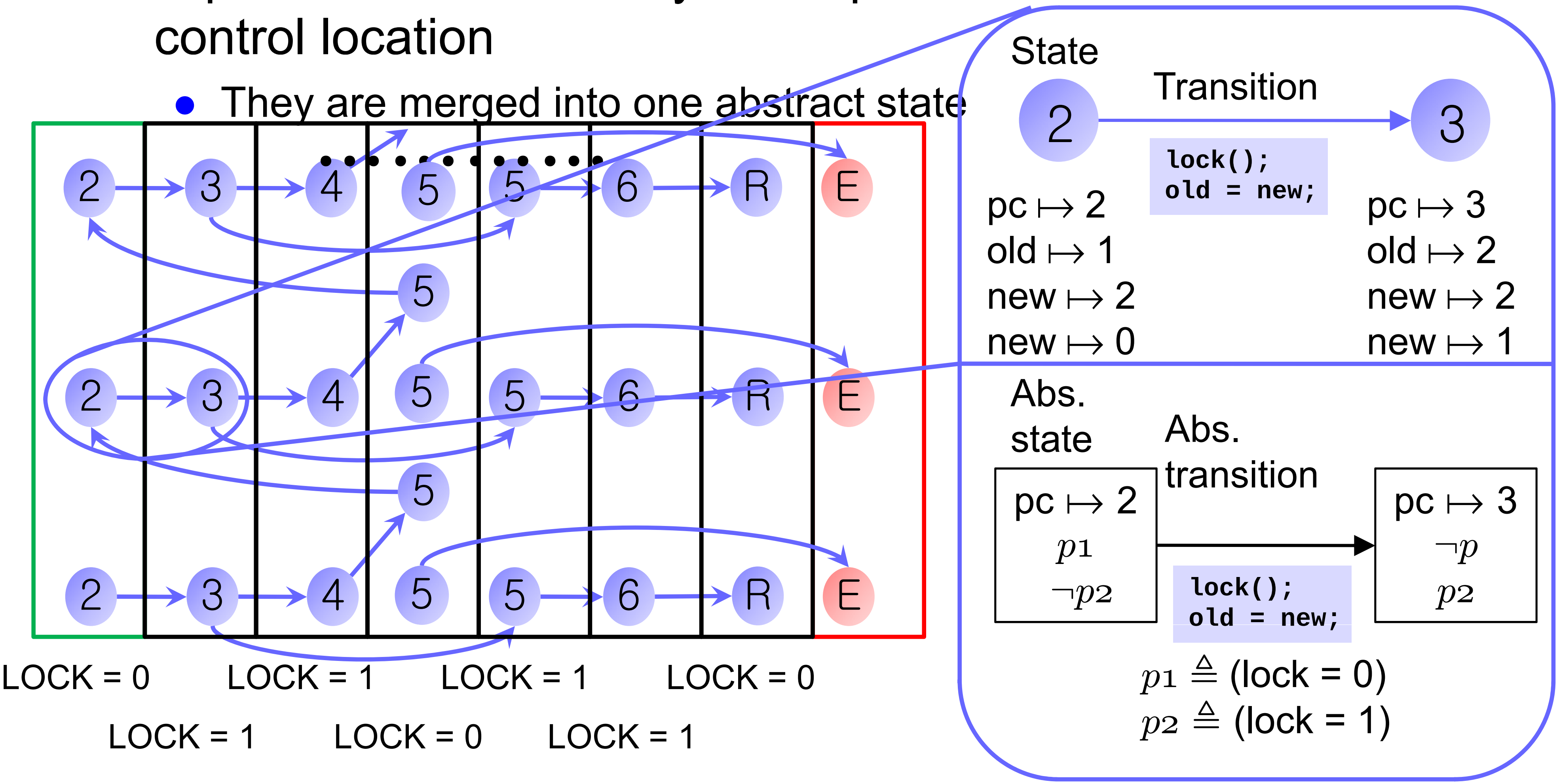
The safety verification

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



Abstract behavior of program

- Equivalent states satisfy same predicates and have same control location
  - They are merged into one abstract state

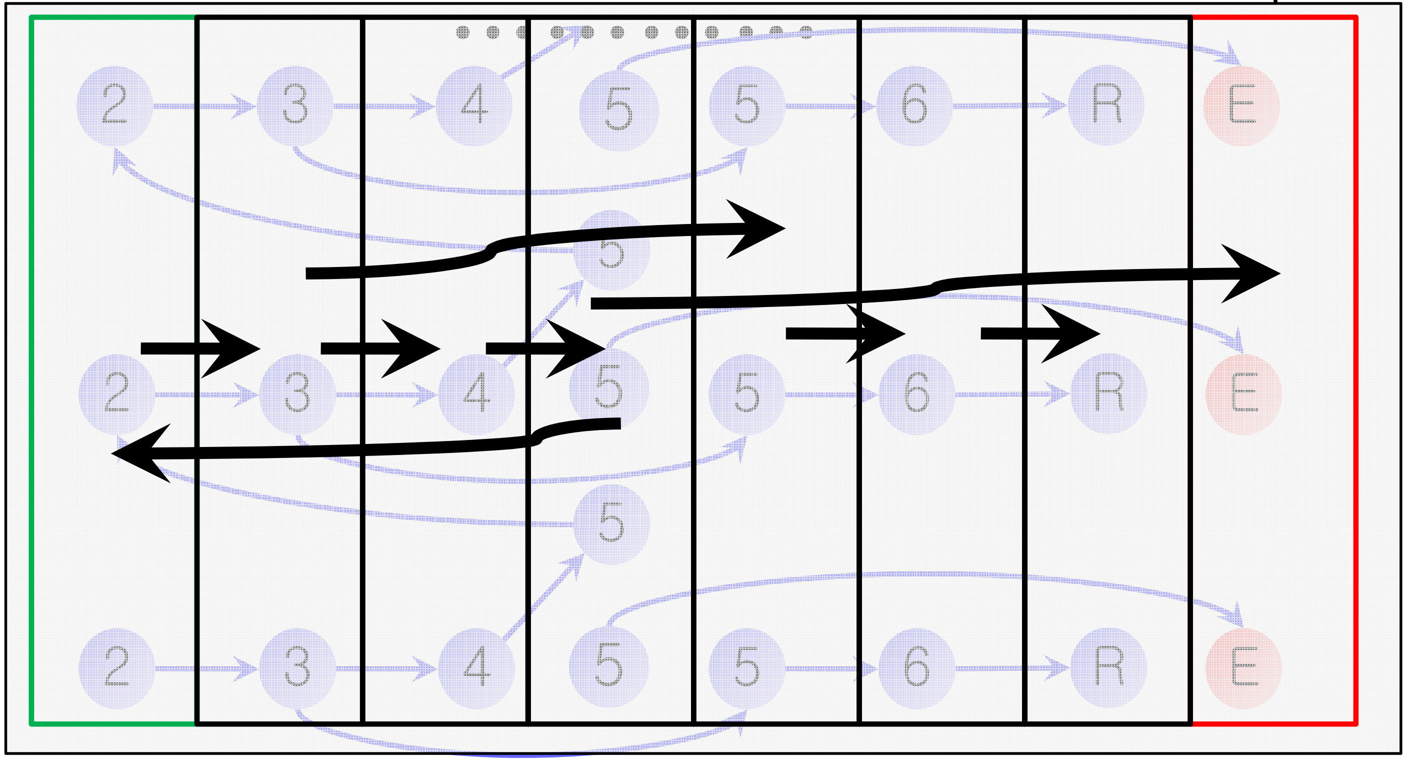


```
lock();
old = new;
```

```
lock();
old = new;
```

Over-approximation

- If there exists a transition between s_1 and s_2, then also there exists a transition between abstract state of s_1 and s_2

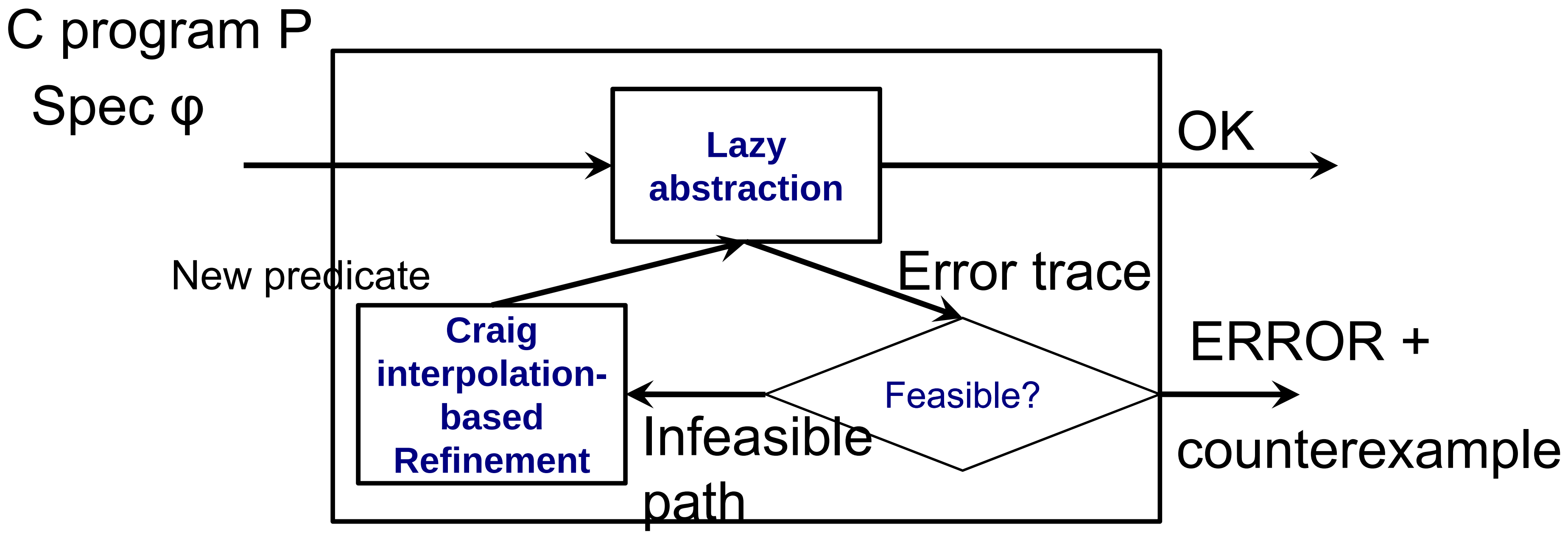


LOCK = 0 LOCK = 1 LOCK = 1 LOCK = 0

LOCK = 1 LOCK = 0 LOCK = 1

CEGAR

- CounterExample-Guided Abstraction Refinement



Part II. BLAST

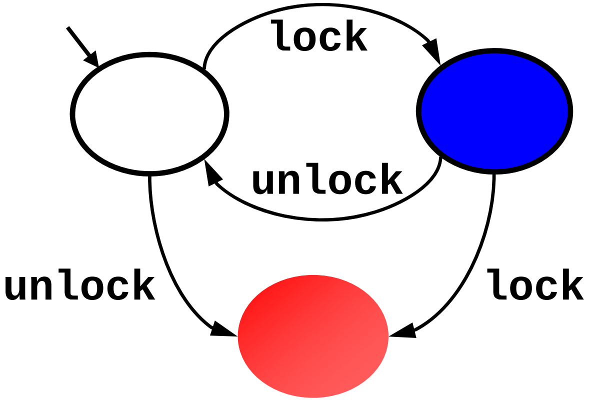
- Abstraction and model checking
- Craig interpolation-based refinement

A locking example

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

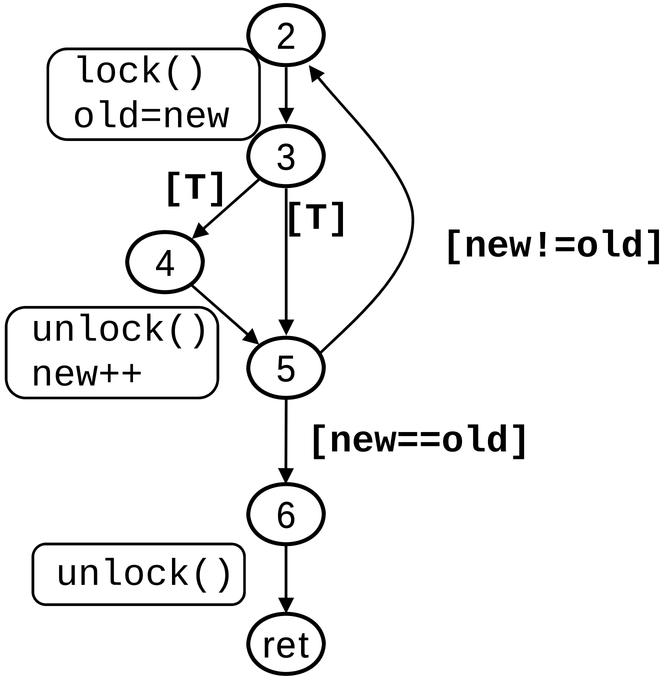
```
lock() {
 if (LOCK == 0) {
  LOCK = 1;
 } else {
  ERROR
 }
}

unlock() {
 if (LOCK == 1) {
  LOCK = 0;
 } else {
  ERROR
 }
}
```



Control Flow Automata for C programs

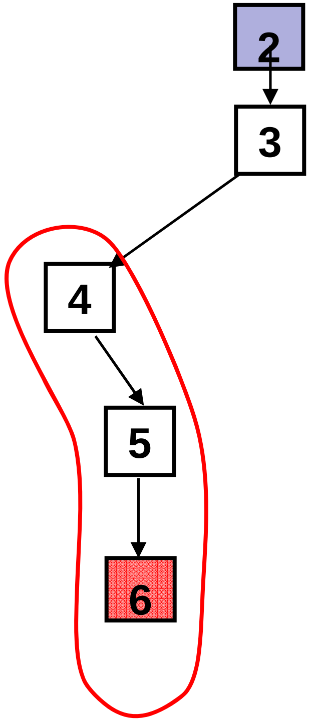
```
1:   Example() {
2:   do {
       lock();
       old = new;
3:     if (*) {
4:       unlock();
         new++;
       }
5:   } while (new != old);
6:   unlock();
     return;
}
```



- Node corresponds to control location
- Edge corresponds to either a basic block or an assume predicate

Reachability tree

Initial



Unroll Abstraction

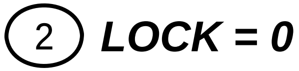
1. Pick tree-node **(=abs. state)**
2. Add children **(=abs. successors)**
3. On **re-visiting** abs. state, **cut-off**

Find infeasible trace

- Learn new predicates
- Rebuild subtree with new preds.

Forward search(1/4)

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



Map P from Loc to set of predicates

Location	Predicates
2	LOCK = 0, LOCK = 1
3	LOCK = 0, LOCK = 1
4	LOCK = 0, LOCK = 1
5	LOCK = 0, LOCK = 1
6	LOCK = 0, LOCK = 1

- Each tree node corresponds to control location and labeled with reachable region
- Edge corresponds to either a basic block or an assume predicate

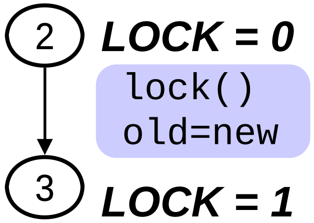
Reachability Tree

Forward search(2/4)

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

Map P from Loc to set of predicates

Location	Predicates
2	LOCK = 0, LOCK = 1
3	LOCK = 0, LOCK = 1
4	LOCK = 0, LOCK = 1
5	LOCK = 0, LOCK = 1
6	LOCK = 0, LOCK = 1



Compute successors where op = ‘x:=e’ and

Loc is successors’ program counter

SP(ϕ, x:=e) = ϕ [x’/x] $\wedge$ (x = e[x’/x])

SP(ϕ, x:=e) w.r.t. P(Loc) = ψ s.t.

(1) SP(ϕ, x:=e) $\Rightarrow$ ψ

(2) ψ is a boolean combination of P(Loc)

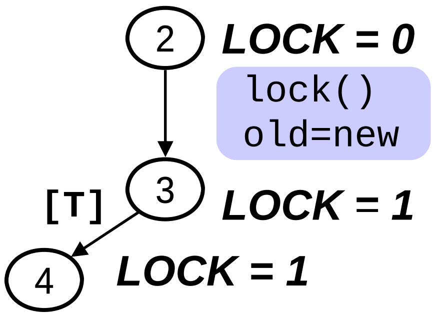
Reachability Tree

Forward search(3/4)

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

Map P from Loc to set of predicates

Location	Predicates
2	LOCK = 0, LOCK = 1
3	LOCK = 0, LOCK = 1
4	LOCK = 0, LOCK = 1
5	LOCK = 0, LOCK = 1
6	LOCK = 0, LOCK = 1



Compute successors where op = '[pred]' and

Loc is successors' program counter

SP(ϕ, [pred]) = $\phi \wedge$ [pred]

SP(ϕ, [pred]) w.r.t. P(Loc) = ψ s.t.

(1) SP(ϕ, [pred]) $\Rightarrow \psi$

(2) ψ is a boolean combination of P(Loc)

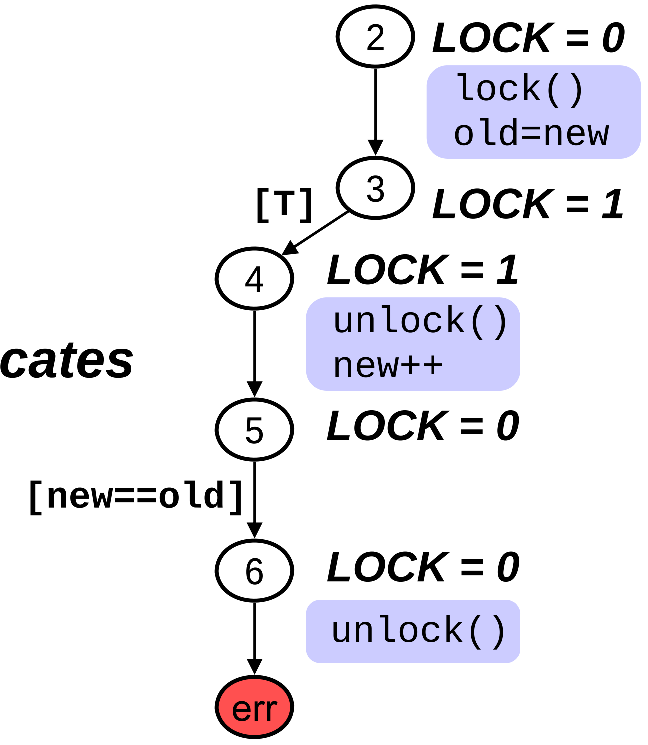
Reachability Tree

Forward search(4/4)

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

Map P from Loc to set of predicates

Location	Predicates
2	LOCK = 0, LOCK = 1
3	LOCK = 0, LOCK = 1
4	LOCK = 0, LOCK = 1
5	LOCK = 0, LOCK = 1
6	LOCK = 0, LOCK = 1



```
Counterexample trace
1: assume(true);
2: lock = 1;
   old = new;
3: assume(true);
4: lock = 0;
   new++;
5: assume(new==old);
6: assume(LOCK!=1);
```

Reachability Tree

Feasibility checking

```
Counterexample trace
1: assume(true);
2: LOCK = 1;
   old = new;
3: assume(true);
4: LOCK = 0;
   new++;
5: assume(new==old);
6: assume(LOCK!=1);
```

SSA form

1: assume(true);

2: $\texttt{LOCK}_0 = 1$;

   $\texttt{old}_0 = \texttt{new}_0$;

3: assume(true);

4: $\texttt{LOCK}_1 = 0$;

   $\texttt{new}_1 = \texttt{new}_0 + 1$;

5: assume($\texttt{new}_1 == \texttt{old}_0$);

6: assume($\texttt{LOCK}_1 != 1$);

Trace formula

1: true

2: $\wedge\ \texttt{LOCK}_0 = 1$;

   $\wedge\ \texttt{old}_0 = \texttt{new}_0$;

3: $\wedge$ true;

4: $\wedge\ \texttt{LOCK}_1 = 0$;

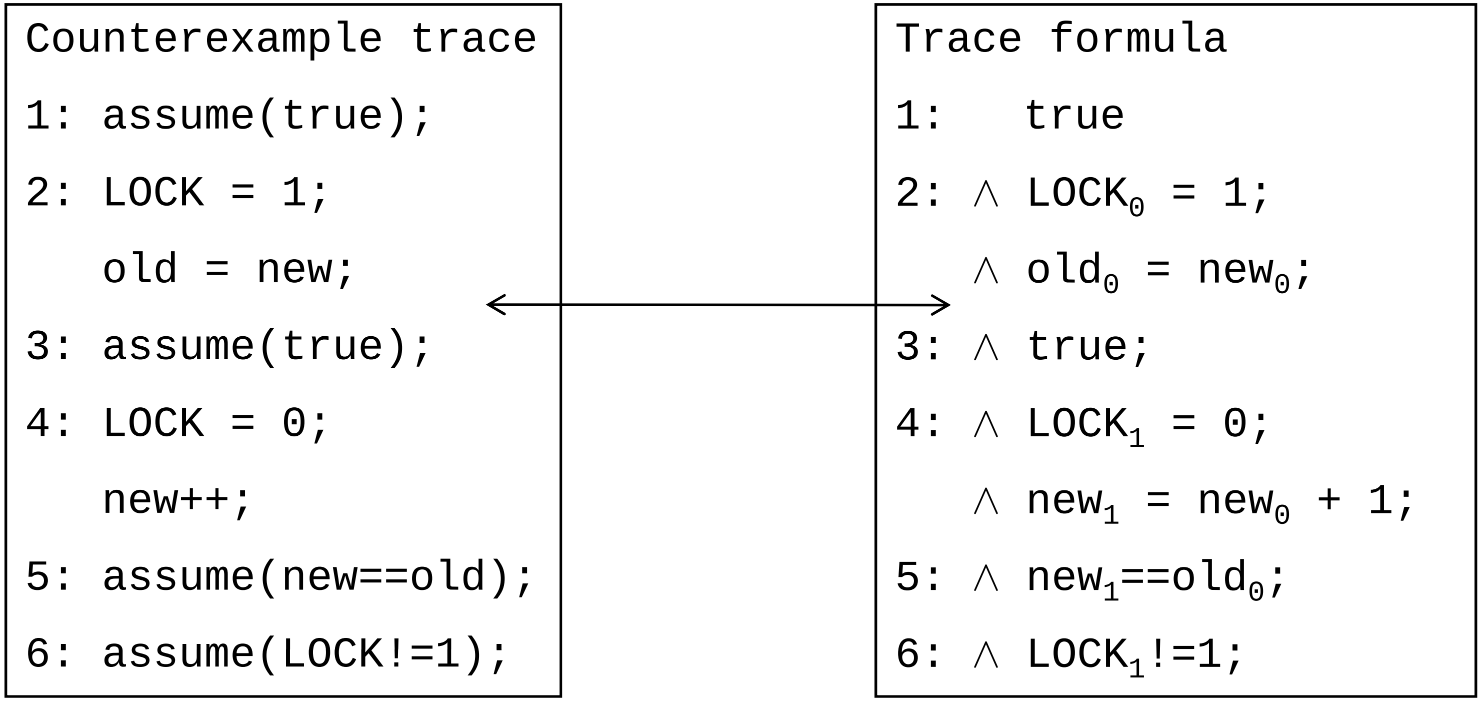
   $\wedge\ \texttt{new}_1 = \texttt{new}_0 + 1$;

5: $\wedge\ \texttt{new}_1 == \texttt{old}_0$;

6: $\wedge\ \texttt{LOCK}_1 != 1$;

Trace is feasible ⇔ Trace formula is satisfiable

Which predicate is needed?



```
Counterexample trace
1: assume(true);
2: LOCK = 1;
   old = new;
3: assume(true);
4: LOCK = 0;
   new++;
5: assume(new==old);
6: assume(LOCK!=1);
```

Trace formula

1: $\quad$ true

2: $\land$ $LOCK_0 = 1;$

$\quad\land$ $old_0 = new_0;$

3: $\land$ true;

4: $\land$ $LOCK_1 = 0;$

$\quad\land$ $new_1 = new_0 + 1;$

5: $\land$ $new_1{=}{=}old_0;$

6: $\land$ $LOCK_1 != 1;$

Relevant information
1. Can be obtained after executing trace
2. has present values of variables
3. Makes trace suffix infeasible

Relevant predicate
1. Implied by TF preffix
2. On common variables
3. TF suffix is unsatisfiable

Craig interpolant

- Given a pair (ϕ^-, ϕ^+) of formulas, an interpolant for (ϕ^-, ϕ^+) is a formula ψ such that

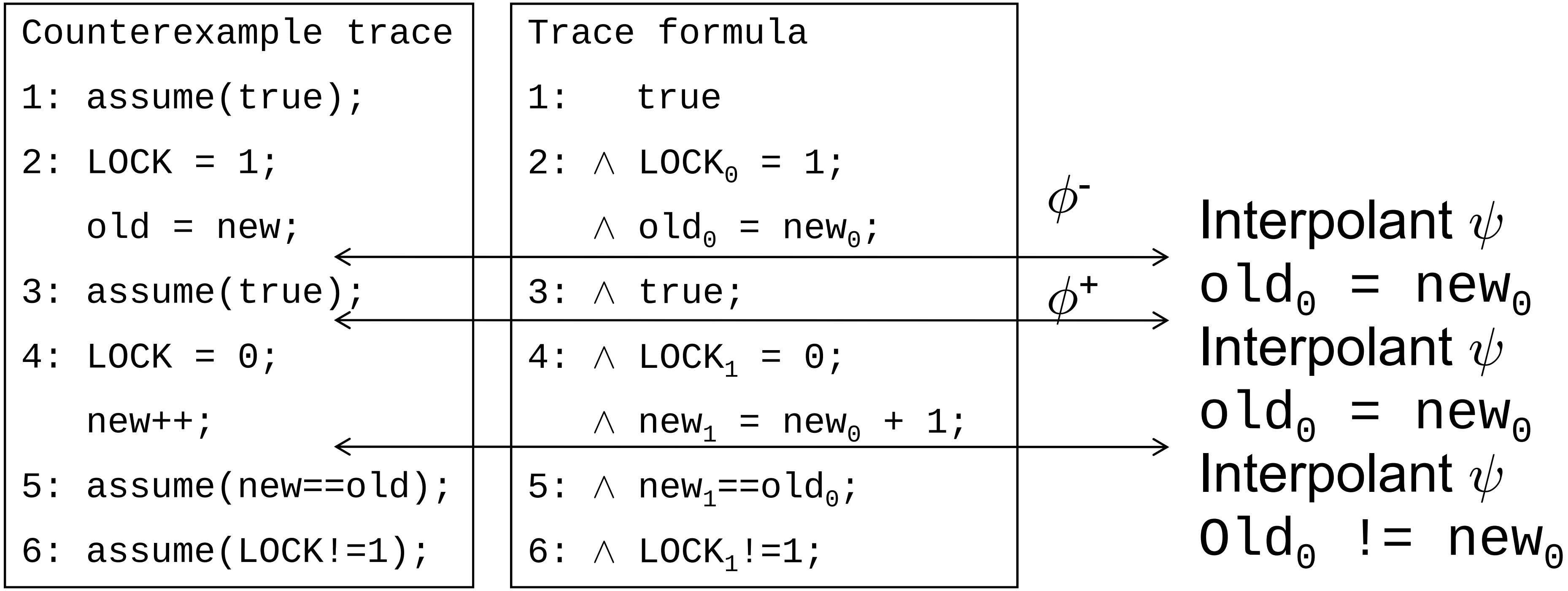
  (i) $\phi^- \Rightarrow \psi$

  (ii) $\psi \wedge \phi^+$ is unsatisfiable

  (iii) the variables of ψ are common to both ϕ^- and ϕ^+

- If $\phi^- \wedge \phi^+$ is unsatisfiable, then an interpolant always exists, and can be computed from a proof of unsatisfiability of $\phi^- \wedge \phi^+$

Craig interpolant



```
Counterexample trace
1: assume(true);
2: LOCK = 1;
   old = new;
3: assume(true);
4: LOCK = 0;
   new++;
5: assume(new==old);
6: assume(LOCK!=1);
```

Trace formula

1: $\quad \texttt{true}$

2: $\wedge\ \texttt{LOCK}_0 = 1;$

$\quad \wedge\ \texttt{old}_0 = \texttt{new}_0;$

3: $\wedge\ \texttt{true};$

4: $\wedge\ \texttt{LOCK}_1 = 0;$

$\quad \wedge\ \texttt{new}_1 = \texttt{new}_0 + 1;$

5: $\wedge\ \texttt{new}_1\texttt{==}\texttt{old}_0;$

6: $\wedge\ \texttt{LOCK}_1\texttt{!=}1;$

ϕ^-

ϕ^+

Interpolant ψ

$\texttt{old}_0 = \texttt{new}_0$

Interpolant ψ

$\texttt{old}_0 = \texttt{new}_0$

Interpolant ψ

$\texttt{Old}_0 \texttt{ != } \texttt{new}_0$

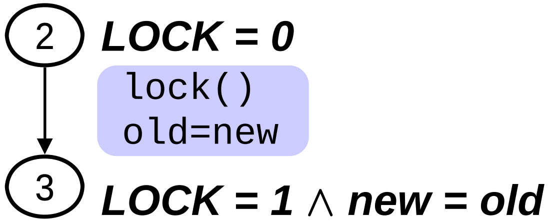
Relevant predicate
1. Implied by TF suffix
2. On common variables

3. $\wedge$ TF suffix is unsatisfiable

ψ is a formula such that
1. $\phi^- \Rightarrow \psi$
2. ψ only contains common variables of ϕ^- and ϕ^+
3. $\psi \wedge \phi^+$ is unsatisfiable

Search with new predicates(1/3)

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



Map P’ from loc to set of predicates

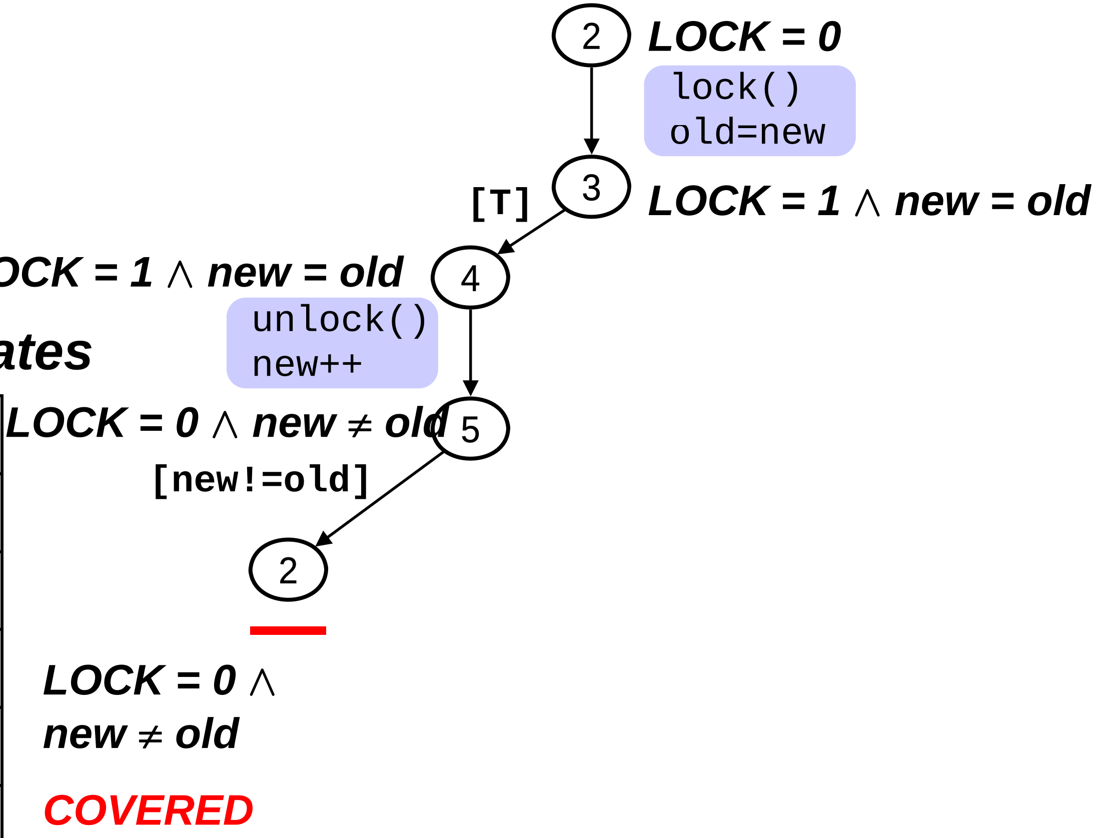
Location	Predicates
2	LOCK = 0, LOCK = 1,
3	LOCK = 0, LOCK = 1, old = new
4	LOCK = 0, LOCK = 1, old = new
5	LOCK = 0, LOCK = 1, old != new
6	LOCK = 0, LOCK = 1

Search with new predicates(2/3)

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

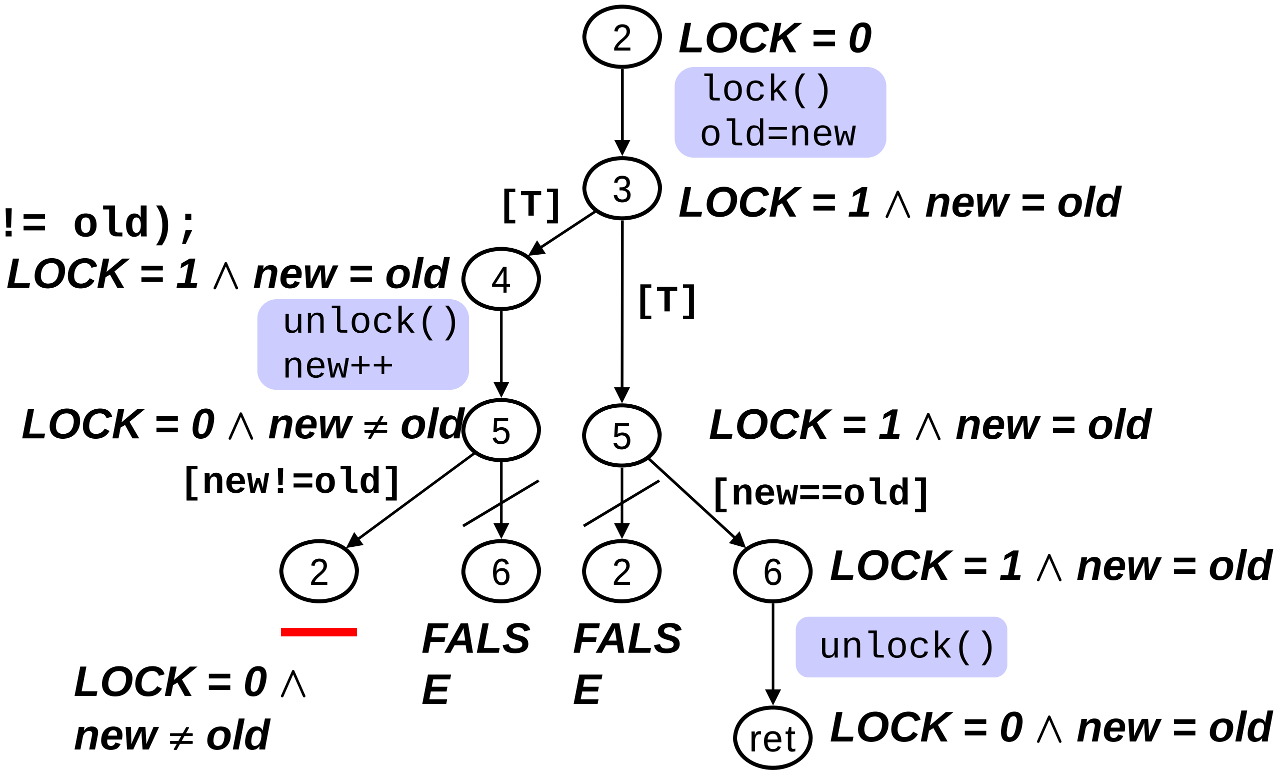
Map P' from loc to set of predicates

Location	Predicates
2	LOCK = 0, LOCK = 1,
3	LOCK = 0, LOCK = 1, old = new
4	LOCK = 0, LOCK = 1, old = new
5	LOCK = 0, LOCK = 1, old != new
6	LOCK = 0, LOCK = 1



Search with new predicates(3/3)

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



2 $LOCK = 0$

lock()
old=new

3 $LOCK = 1 \wedge new = old$

[T]

4 $LOCK = 1 \wedge new = old$

unlock()
new++

5 $LOCK = 0 \wedge new \neq old$

[new!=old]

2 $LOCK = 0 \wedge new \neq old$

6 FALSE

[T]

5 $LOCK = 1 \wedge new = old$

2 FALSE

[new==old]

6 $LOCK = 1 \wedge new = old$

unlock()

ret $LOCK = 0 \wedge new = old$

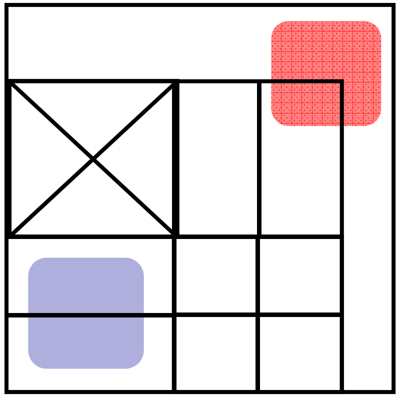
COVERED

Safe!

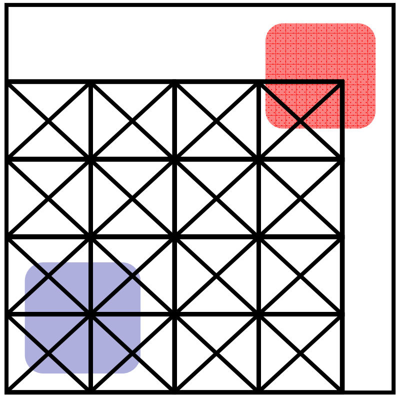
Local predicate use

- Use predicates needed at location

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

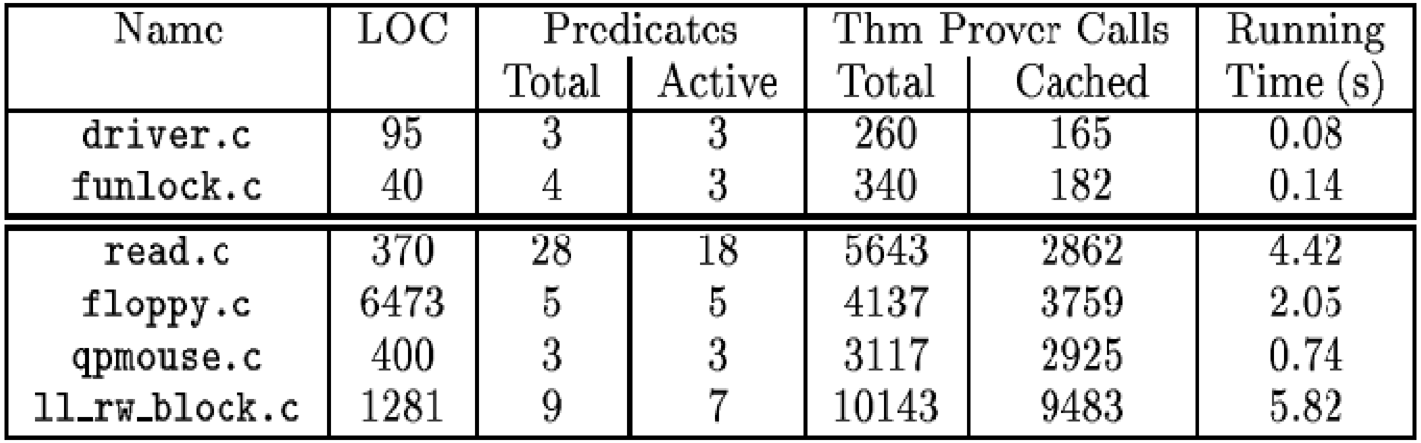


Local Predicate use
Ex: 2n states



Global Predicate use
Ex: 2^n states

Experiments



Name	LOC	Predicates		Thm Prover Calls		Running Time (s)
		Total	Active	Total	Cached	
driver.c	95	3	3	260	165	0.08
funlock.c	40	4	3	340	182	0.14
read.c	370	28	18	5643	2862	4.42
floppy.c	6473	5	5	4137	3759	2.05
qpmouse.c	400	3	3	3117	2925	0.74
ll_rw_block.c	1281	9	7	10143	9483	5.82

- funlock.c is an example we covered
- driver.c is a Windows driver for verifying locking discipline
- read.c, floppy.c are drivers from Windows DDK
- qpmouse.c and llrw_block.c are drivers from Linux
- Experiments ran on 800MHz PIII with 256M RAM

Conclusions

- BLAST is a software model checker for verifying program written in C language

- BLAST improves the scheme of CEGAR by implementing lazy abstraction
  - avoids redundant abstraction and checking
  - Predicates are locally applied and states are locally abstracted

Reference

- The Software Model Checker Blast: Applications to Software Engineering.

  by Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala,

  and Rupak Majumdar

  in Int. Journal on Software Tools for Technology Transfer, 2007
- Lazy abstraction

  by Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre

  in ACM SIGPLAN – SIGACT Conference on

  Principle Of Programming Language 2000
- Abstractions from Proofs

  by Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar and Kenneth L. McMillan

  in ACM SIGPLAN-SIGACT Conference on

  Principles of Programming Languages, 2004

Reference

- Lazy Abstraction slides
  by Jinseong Jeon
  in CS750 class, Fall, 2006
- Software Verification with BLAST slides
  by Tom Henzinger, Ranjit , Rupak Majumdar
  in SPIN workshop 2005 tutoria ϕ ψ