BLAST-A Model Checker for C

Developed by
Thomas A. Henzinger (EPFL)
Rupak Majumdar (UC Los Angeles)
Ranjit Jhala (UC San Diego)
Dirk Beyer (Simon Fraser University)

Presented by Sowmya Venkateswaran

BLAST Installation

- Currently version 2.0 src files available.
- http://mtc.epfl.ch/software-tools/blast/
- Installation
  - Download Simplify theorem prover http://www.cs.virginia.edu/~weimer/615/hw.html
  - Either build from src files or use Linux binaries.
  - A “working” example configuration for compiling Blast 2.0 is OCaml 3.08.3 and gcc (GCC) 3.4.4 20050721 (Red Hat 3.4.4-2).

Features

- “On the Fly” Abstraction
- Automatic abstraction
- Smarter predicate discovery
- Verify safety properties, assertion violations
- Finding reachable program locations
- Detecting dead code
- Reuse saved abstractions

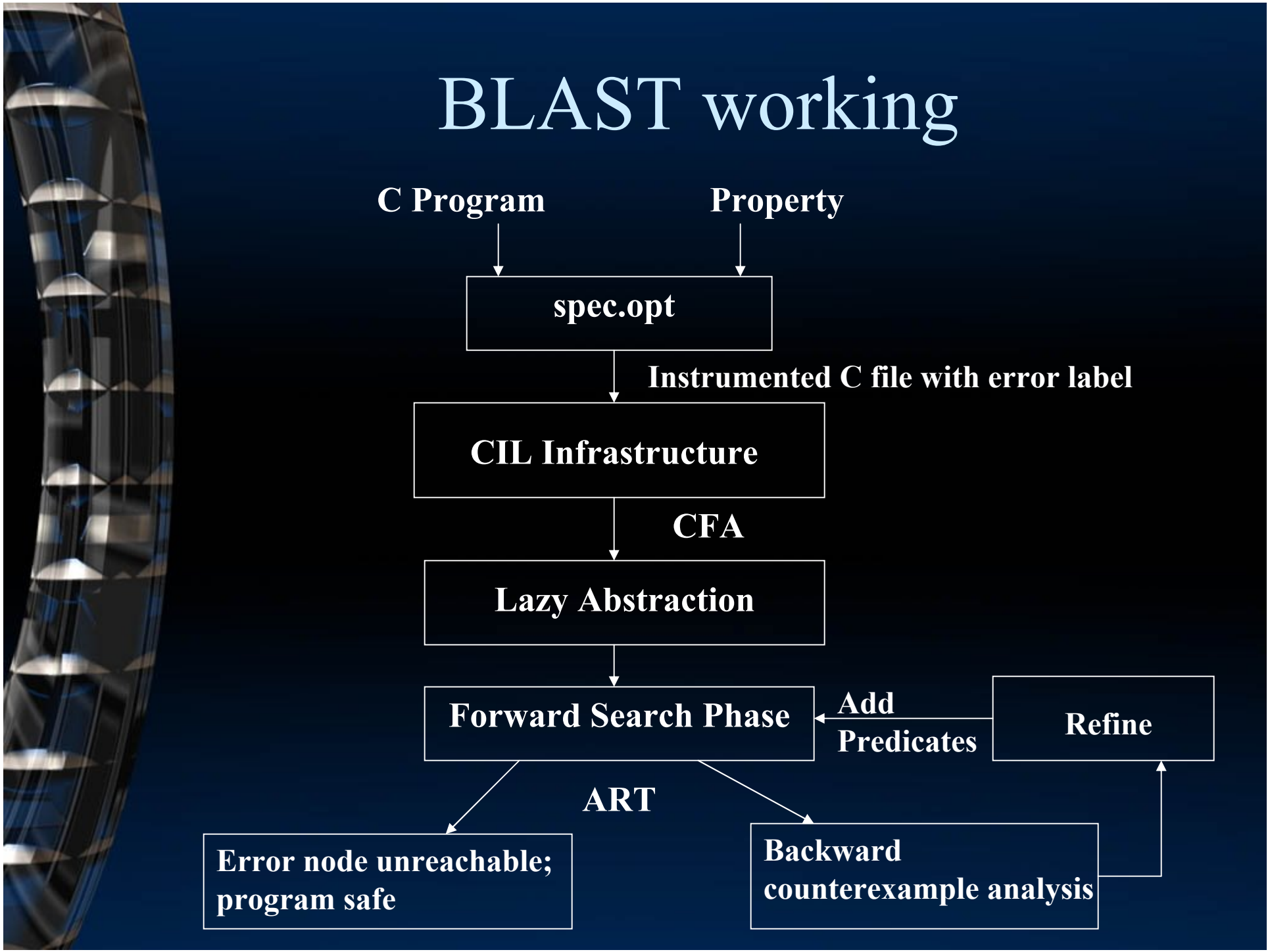
Problems

- Installing and making it work
- Predicate discovery not good enough.
- Checking concurrent programs
- Eclipse plugin
- Checking recursive functions

BLAST working

- Build an abstract model using predicate abstraction.
- Check for reachability of a specified label using the abstract model.
- If no path to ERR node-system safe.
- If path is feasible, output error trace.
- Else use infeasibility of path to refine abstract model.

BLAST working



Problem: Abstraction is expensive

- # of abstract states=$2^{\#\text{ of predicates}}$
- Solution 1: Only abstract reachable states
- Solution 2: Don't refine any error free states
- Advantages:
  - State space only refined as much as required.
  - Reuse previously defined error free states.

Lazy Abstraction

- Integrate the following
  - Abstraction
  - Verification
  - Counterexample-driven refinement
- Find pivot state.
- Construct, verify and refine abstract model “on the fly” from pivot state on.
- Forward Search Phase and Backward Counterexample analysis.
- Stop when either real counterexample found or system found safe

Locking example

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

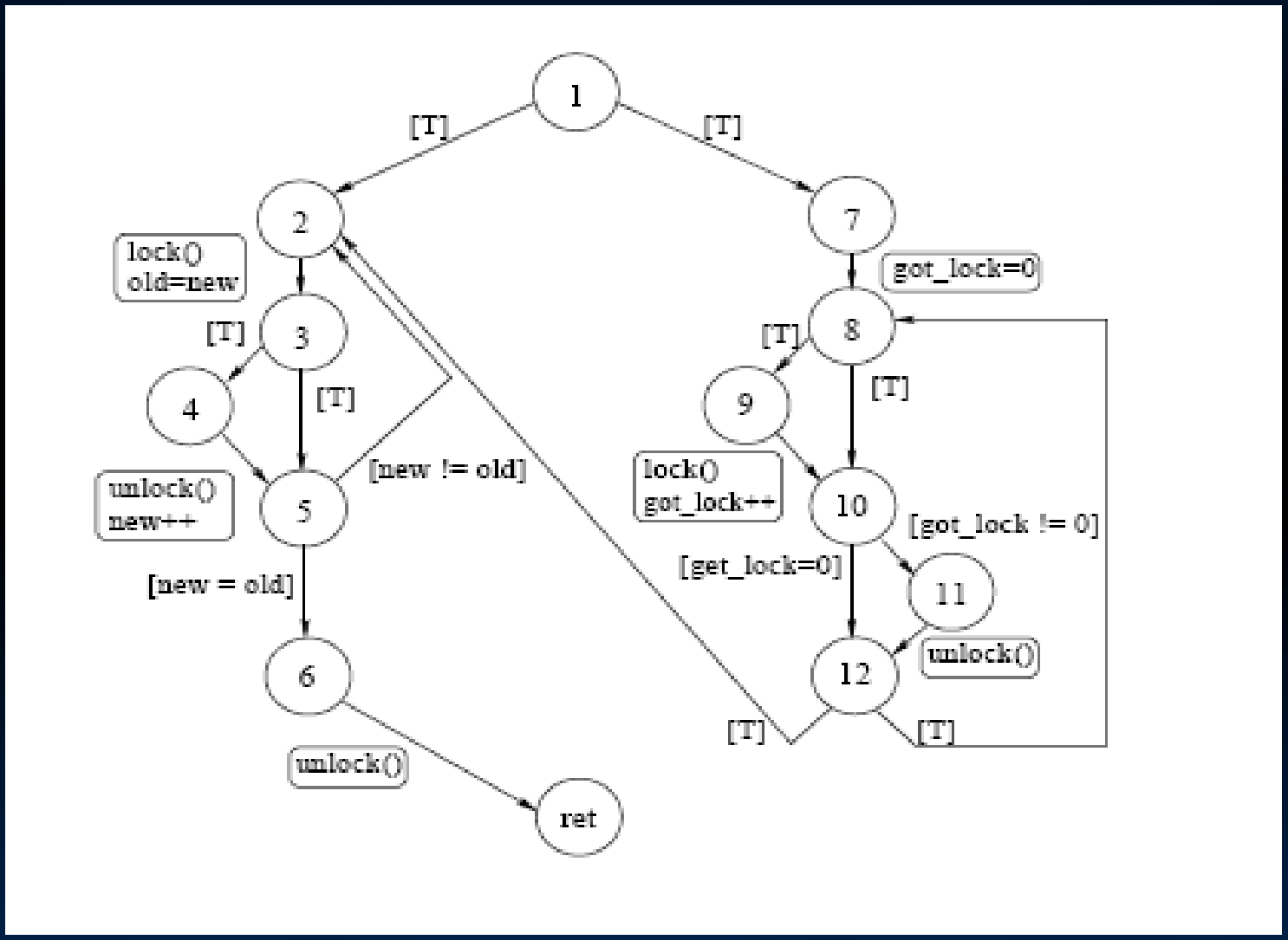
```
lock(){
   if (LOCK == 0){
      LOCK = 1;
   } else {
      ERROR
   }
}

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

Control Flow Automaton

- Local and global variables of C program
- Vertices: control locations of a function.
- Labeled directed edges
  - Basic block of instructions.
  - Assume predicate for branch condition.
- Formally, CFA is a tuple $<Q,q_0,X,Ops,\rightarrow>$
  - Q- finite set of control locations
  - q_0-initial control location
  - X- set of variables
  - Ops- set of operations on X (lval=exp or [p])
  - $\rightarrow \in (Q \times Ops \times Q)$

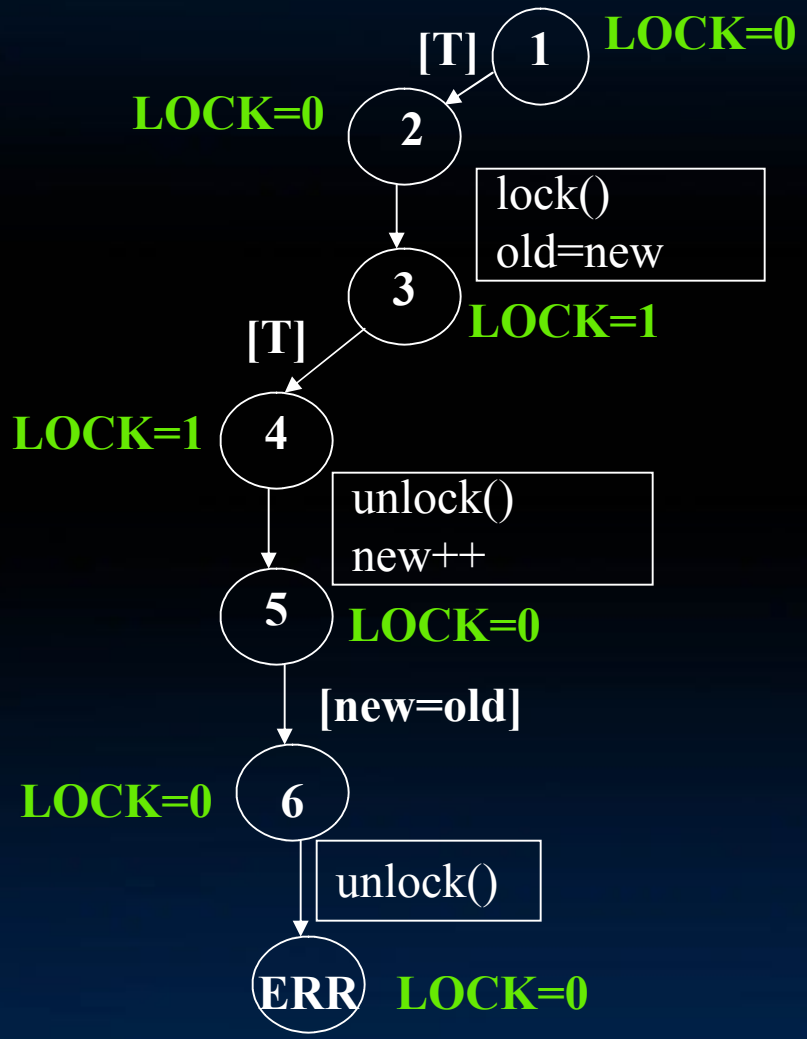
Control Flow Automaton



Forward Search Phase

- Abstract reachability tree in dfs order.
- Constructed from CFA.
- Vertices in CFA are nodes in ART.
- Labels of nodes are reachable regions.
- Reachable region obtained from parent's reachable region and instructions on the edge between them.
- Finite set of predicates per node.
- Reachable region is a boolean combination of set of predicates

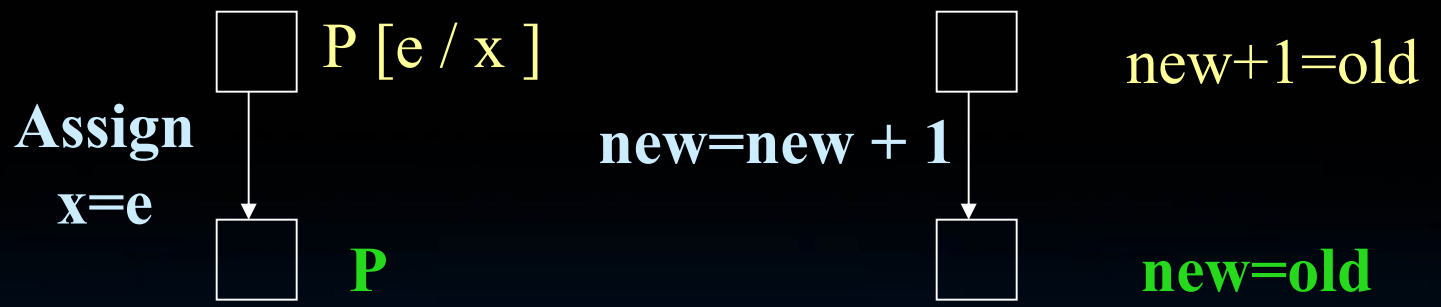
Forward Search for locking example



Is this a valid counterexample??

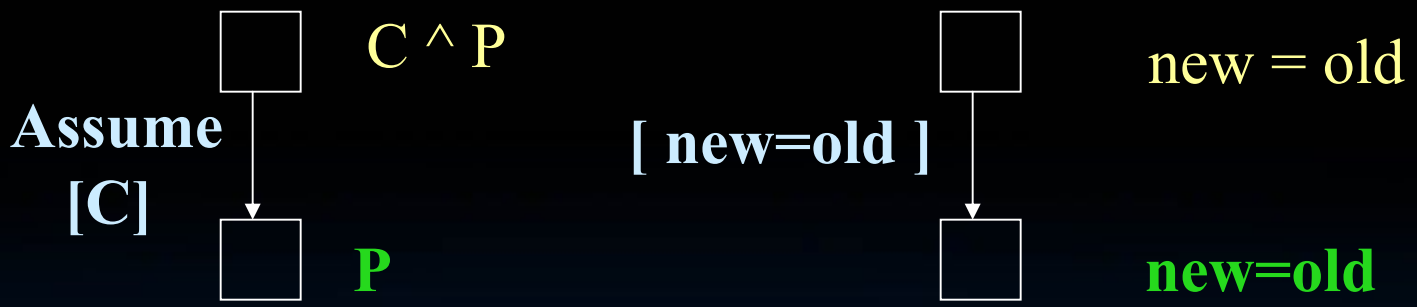
Weakest Precondition

- WP(P,Op) –weakest formula P` s.t. if P` is T before Op, then P is T after Op



Weakest Precondition

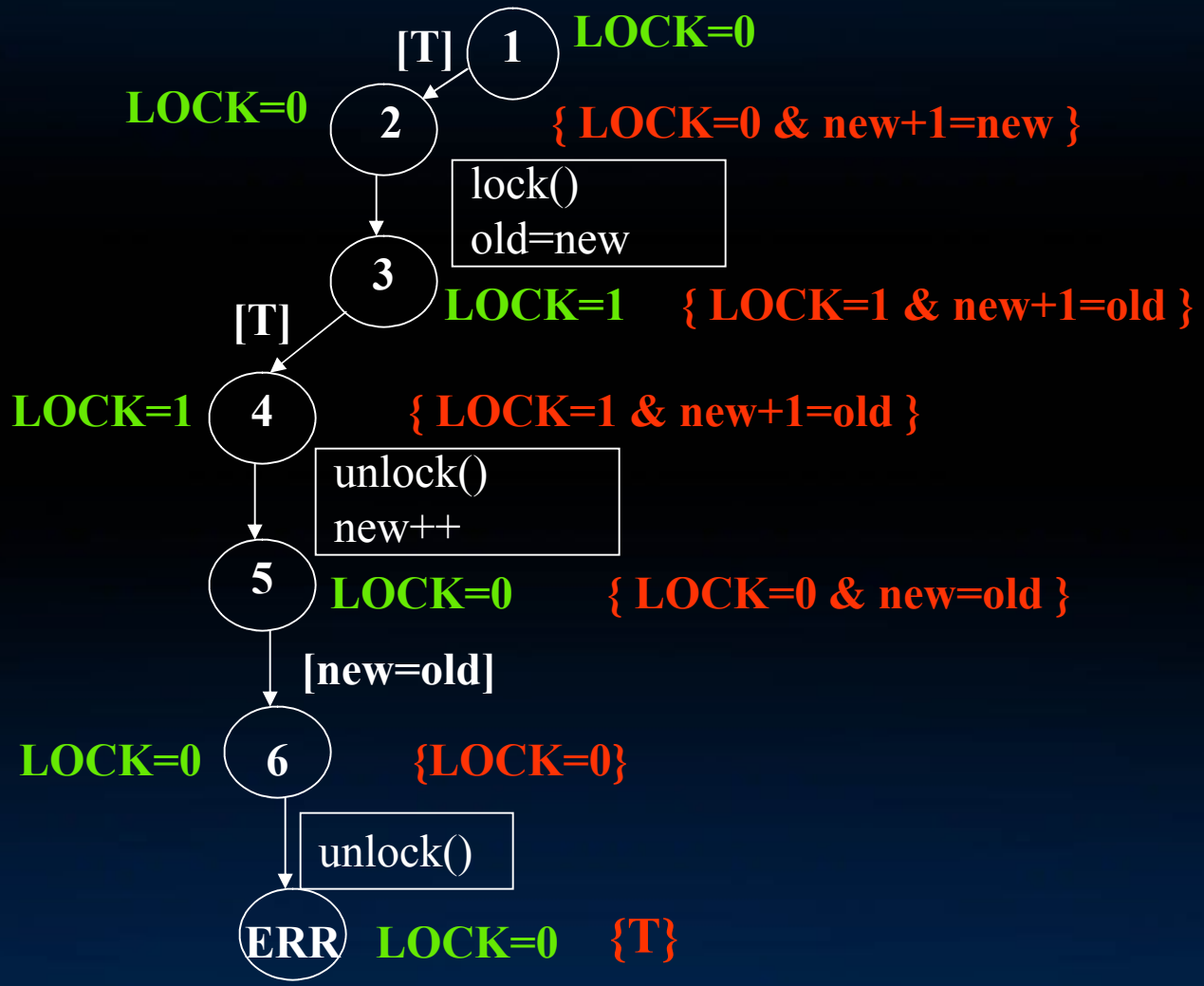
- WP(P,Op) –weakest formula P` s.t. if P` is T before Op, then P is T after Op



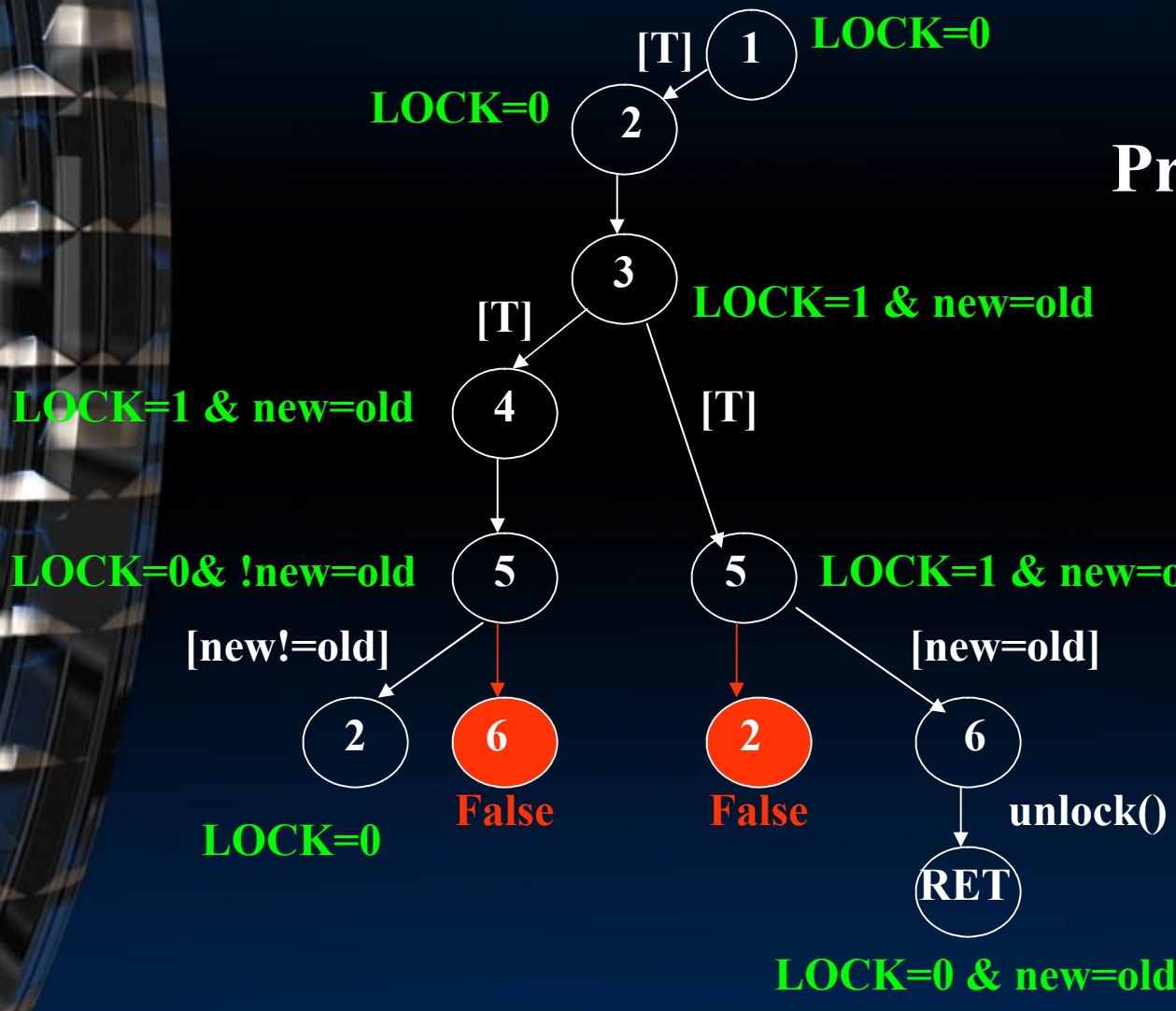
Backward Counterexample Analysis

- For each tree node, find a bad region.
- Bad region of ERR node=T
- Other nodes=WP of bad region of child w.r.t instructions on edge between the 2.
- Start from ERR node
- Pivot node - First node in the tree where Bad region ∩ Reachable region=φ
- Refine abstraction from pivot node onwards

Counter example analysis for locking program



Searching with new predicate new=old

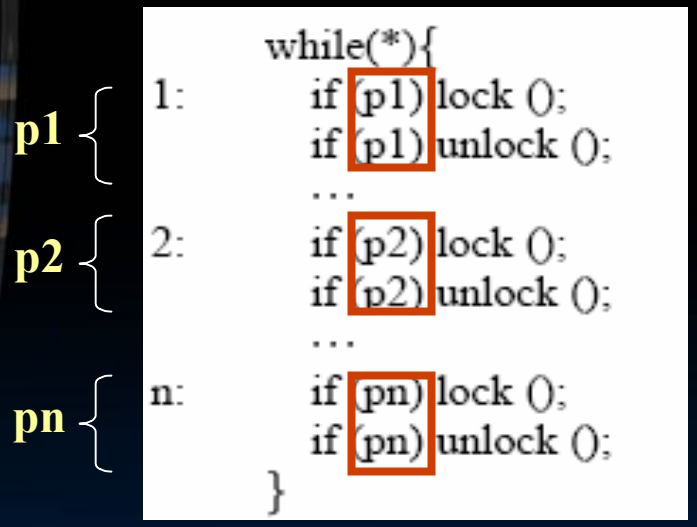


Program Safe!!

Finding Predicates

- Problem: How many predicates to find?

  # of predicates grows with program size



Solution: Use predicates only where needed

2^n abstract states!!

2n abstract states

Counter example Traces

Theorem: Trace formula is satisfiable iff trace is feasible.
Trace formula is a conjunction of constraints, one per instruction in the trace.

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

Sample program

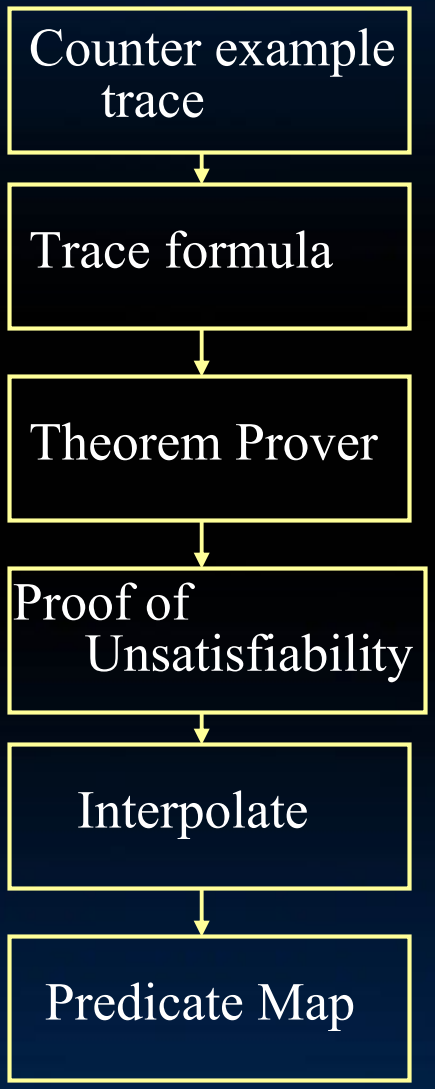
```
1: x=ctr;
2: ctr=ctr+1;
3: y=ctr;
4: assume (x=i-1)
5: assume (y!=i)
```

Counter example trace

$1: x_1 = ctr_0$
$2: ctr_1 = ctr_0 + 1$
$3: y_1 = ctr_1$
$4: x_1 = i_0 - 1$
$5: y_1 != i_0$

Trace Formula φ

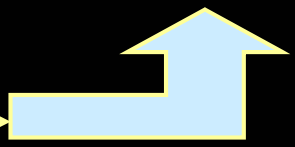
Steps in Refine Stage



Finding what predicates are needed

Trace	Trace Formula
1: x=ctr;	1:$x_1=ctr_0$
2: ctr=ctr+1;	2:$ctr_1=ctr_0+1$
3: y=ctr;	3:$y_1=ctr_1$
4: assume (x=i-1)	4:$x_1=i_0-1$
5: assume (y!=i)	5:$y_1!=i_0$

What predicate is needed for trace to become infeasible

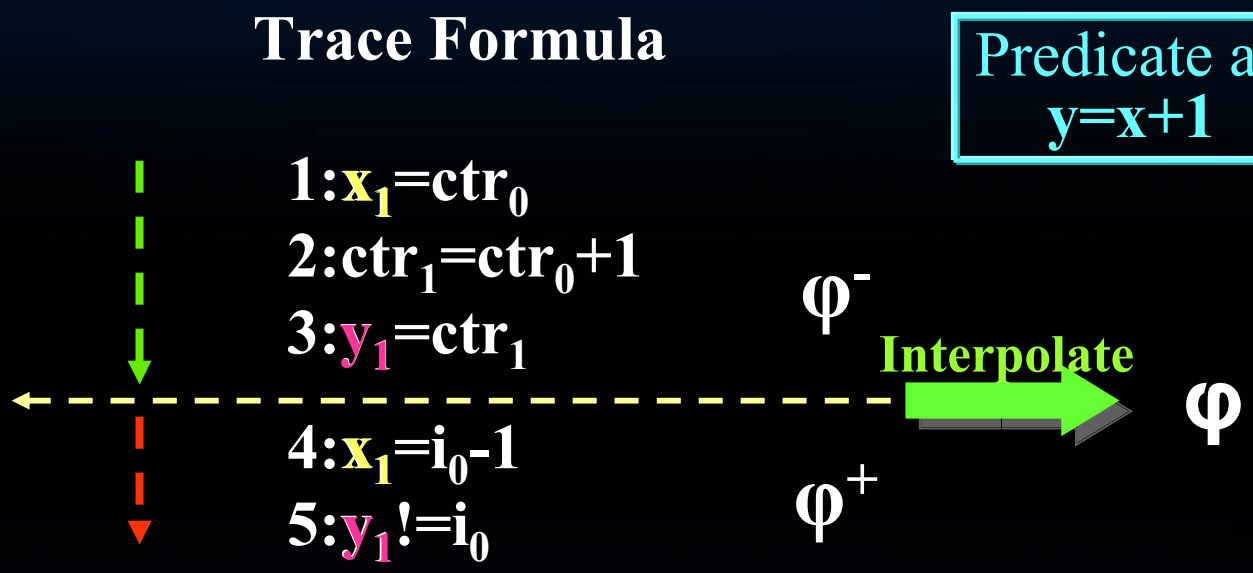


Given an infeasible trace t, find a set of predicates P, such that t is abstractly infeasible w.r.t P.

Finding what predicates are needed

- **Partition φ into φ^- (trace prefix) and φ^+ (trace suffix)**
- **Find an interpolant Ψ s.t**
  - **φ^- implies Ψ**
  - **Ψ ^ φ^+ is unsatisfiable.**
  - **The variables of Ψ are common to both φ^- and φ^+**
- **Use interpolant to construct predicate map.**

Interpolant = Predicate



Predicate is

..implied by Trace formula prefix

..on common variables

..makes Trace Formula suffix unfeasible

Finding predicate map

- Partition at each point
- Interpolate at each partition
- Construct predicate map $pc_i \mapsto$ Interpolant from partition i

Trace	Trace Formula
1: x=ctr;	1: $x_1 = ctr_0$
2: ctr=ctr+1;	2: $ctr_1 = ctr_0 + 1$
3: y=ctr;	3: $y_1 = ctr_1$
4: assume (x=i-1)	4: $x_1 = i_0 - 1$
5: assume (y!=i)	5: $y_1 != i_0$

φ^- | φ^+

Interpolate → $x_1 = ctr_0$

Predicate Map
2: $x_1 = ctr_0$

Finding predicate map

- Partition at each point
- Interpolate at each partition
- Construct predicate map $pc_i \mapsto$ Interpolant from partition i

Trace	Trace Formula
1: x=ctr;	1:$x_1=ctr_0$
2: ctr=ctr+1;	2:$ctr_1=ctr_0+1$
3: y=ctr;	3:$y_1=ctr_1$
4: assume (x=i-1)	4:$x_1=i_0-1$
5: assume (y!=i)	5:$y_1!=i_0$

φ^- Interpolate $x_1=ctr_1-1$

φ^+

Predicate Map
2: $x_1=ctr_0$
3: $x_1=ctr_1-1$

Finding predicate map

- Partition at each point
- Interpolate at each partition
- Construct predicate map $pc_i \mapsto$ Interpolant from partition i

Trace	Trace Formula
1: x=ctr;	1:$x_1=ctr_0$
2: ctr=ctr+1;	2:$ctr_1=ctr_0+1$
3: y=ctr;	3:$y_1=ctr_1$
4: assume (x=i-1)	4:$x_1=i_0-1$
5: assume (y!=i)	5:$y_1!=i_0$

φ^- → Interpolate → $y_1=x_1+1$

φ^+

Predicate Map
2: $x_1=ctr_0$
3: $x_1=ctr_1-1$
4: $y_1=x_1+1$

Finding predicate map

- Partition at each point
- Interpolate at each partition
- Construct predicate map $pc_i \mapsto$ Interpolant from partition i

Trace	Trace Formula
1: x=ctr;	1:$x_1=ctr_0$
2: ctr=ctr+1;	2:$ctr_1=ctr_0+1$
3: y=ctr;	3:$y_1=ctr_1$
4: assume (x=i-1)	4:$x_1=i_0-1$
5: assume (y!=i)	5:$y_1!=i_0$

φ^-

Interpolate $y_1=i_0$

φ^+

Predicate Map
2: $x_1=ctr_0$
3: $x_1=ctr_1-1$
4: $y_1=x_1+1$
5: $y_1=i_0$

BLAST Specification language

- Include directives
- Global variables
- Shadowed types
- Events
  - Pattern
  - Guard
  - Action/Repair
  - Before/After

References

- “Abstractions from Proofs”-Thomas .H et al.
- “The Blast query language for software verification”- Dirk Beyer et al.
- “Lazy Abstraction”-Gregoire Sutre et al.