

Counterexample Guided Abstraction Refinement in Blast

Optional reading: ***Checking Memory
Safety with Blast***

17-355/17-665/17-819: Program Analysis
Jonathan Aldrich



How would you analyze this?



```
Example() {  
1:   if (*) {  
7:       do {  
           got_lock = 0;  
8:           if (*) {  
9:               lock();  
               got_lock++;  
           }  
10:          if (got_lock) {  
11:              unlock();  
          }  
12:      } while (*)  
}
```

- * means something we can't analyze (user input, random value)
- Line 10: the lock is held if and only if got_lock = 1

How would you analyze this?



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

- * means something we can't analyze (user input, random value)
- Line 5: the lock is held if and only if old = new

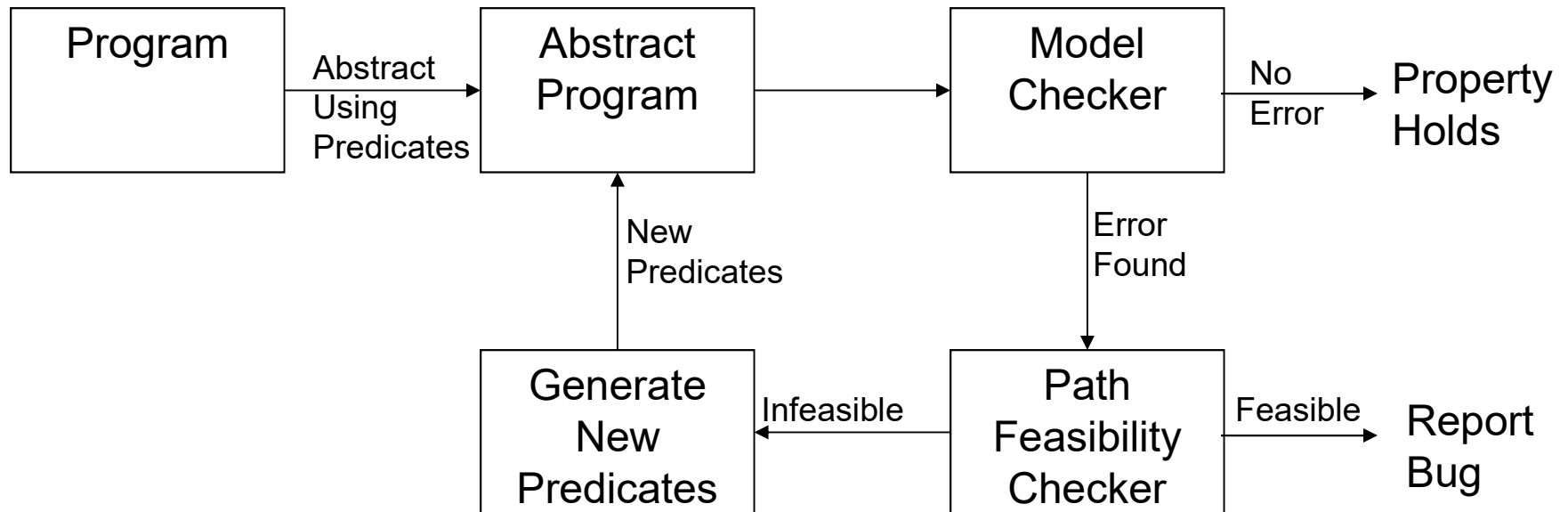


Motivation

- Dataflow analysis uses fixed abstraction
 - e.g. zero/nonzero, locked/unlocked
 - Model checking version of DFA similar
- Symbolic execution shows need to eliminate infeasible paths
 - E.g. lock/unlock on correlated branches
 - Requires extending abstraction with branch predicates
- It's hard to make symbolic execution sound
 - Infeasible to cover all paths
 - Although we can merge paths with similar analysis info, the information is too detailed to assure finitely many explored paths
- Can we get both soundness and the precision to eliminate infeasible paths?
 - **In general: of course not! That's undecidable.**
 - But in many situations we can solve it with *abstraction refinement*; it's just that this technique may not always terminate

CEGAR:

Counterexample Guided Abstraction Refinement



CEGAR:

Counterexample Guided Abstraction Refinement

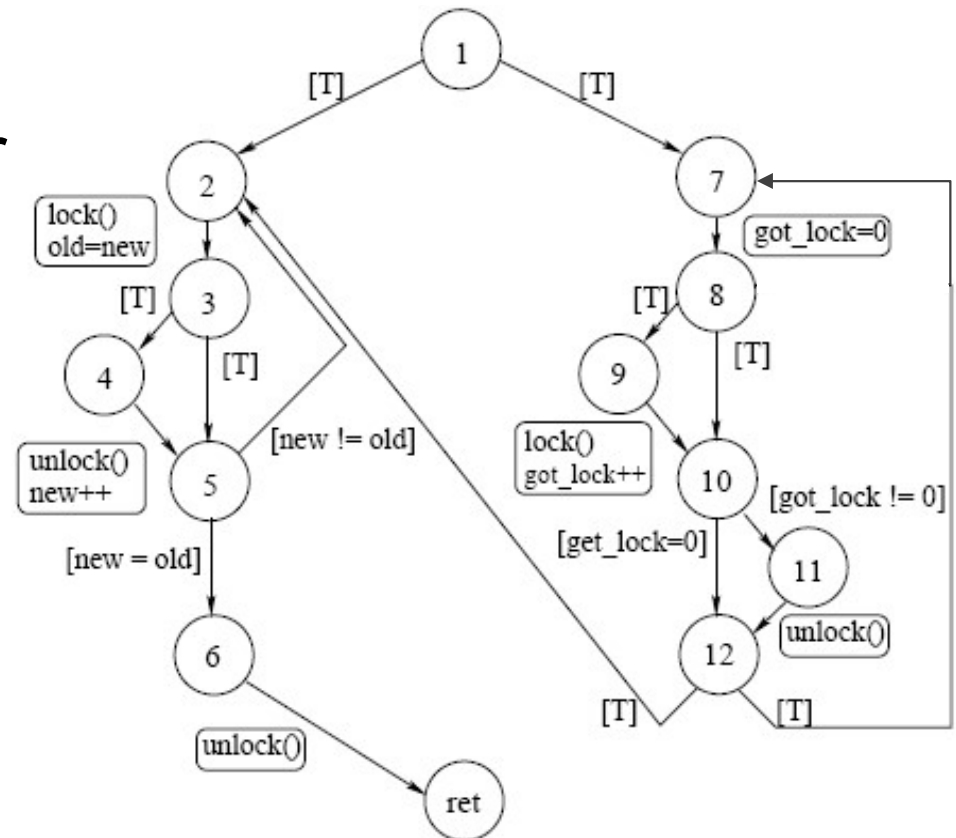


- Begin with control flow graph (CFG) abstraction
- Check reachability of error nodes
 - For conditional errors, surround with a branch
- If error node is reachable, check if path is feasible
 - Can use weakest preconditions; if you get false, the path is impossible
- For feasible paths, report an error
- For infeasible paths, figure out why
 - e.g. correlation between lock and got_lock
- Add reason for infeasible paths to abstraction and try again!
 - Cross product of CFG and predicate abstractions
 - This time the analysis won't consider that path
 - But it might consider other infeasible paths, so you may have to repeat the process multiple times



Control Flow Automaton

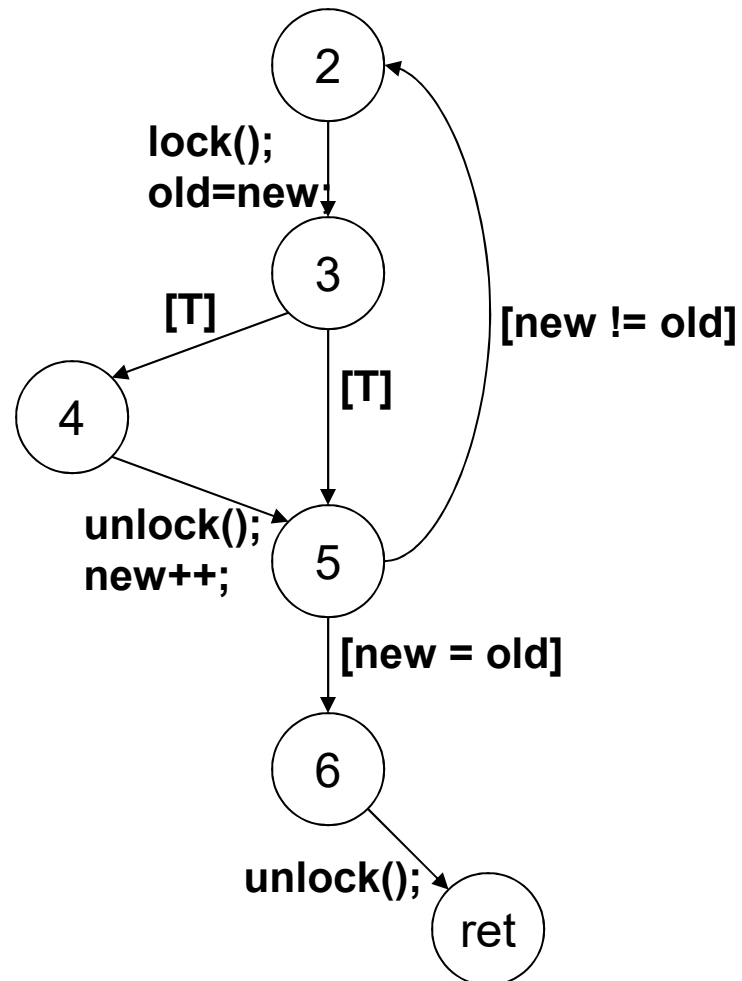
- One node for each location (before/after a statement)
- Edges
 - Blocks of statements
 - Assume clauses model if and loops
 - some predicate must be true to take the edge



Control Flow Automaton Example



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

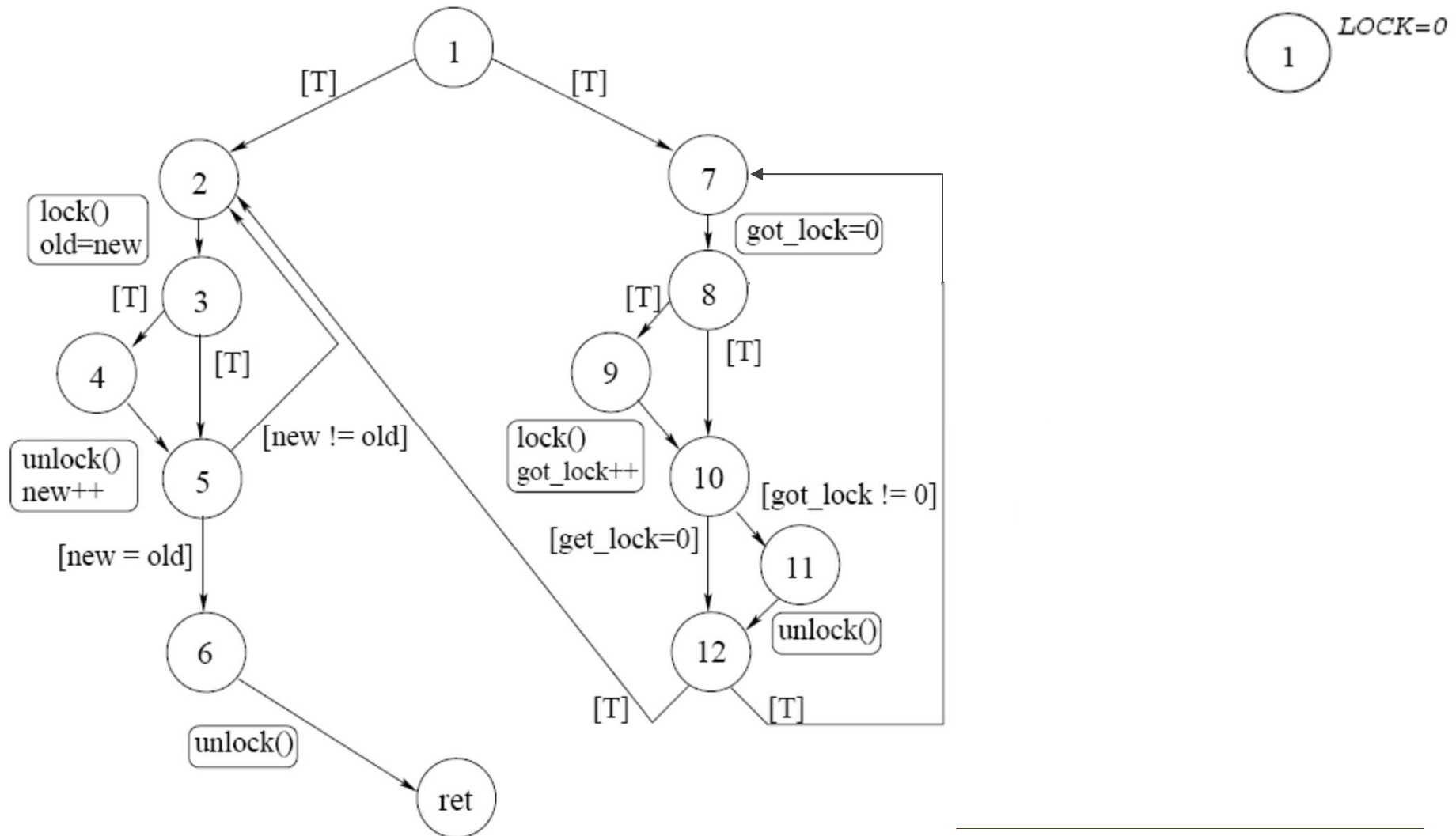




Checking for Reachability

- Generate Abstract Reachability Tree
 - Contains all reachable nodes
 - Annotates each node with state
 - Initially LOCK = 0 or LOCK = 1
 - Cross product of CFA and data flow abstraction
- Algorithm: depth-first search
 - Generate nodes one by one
 - If you come to a node that's already in the tree, stop
 - This state has already been explored through a different control flow path
 - If you come to an error node, stop
 - The error is reachable

Depth First Search Example





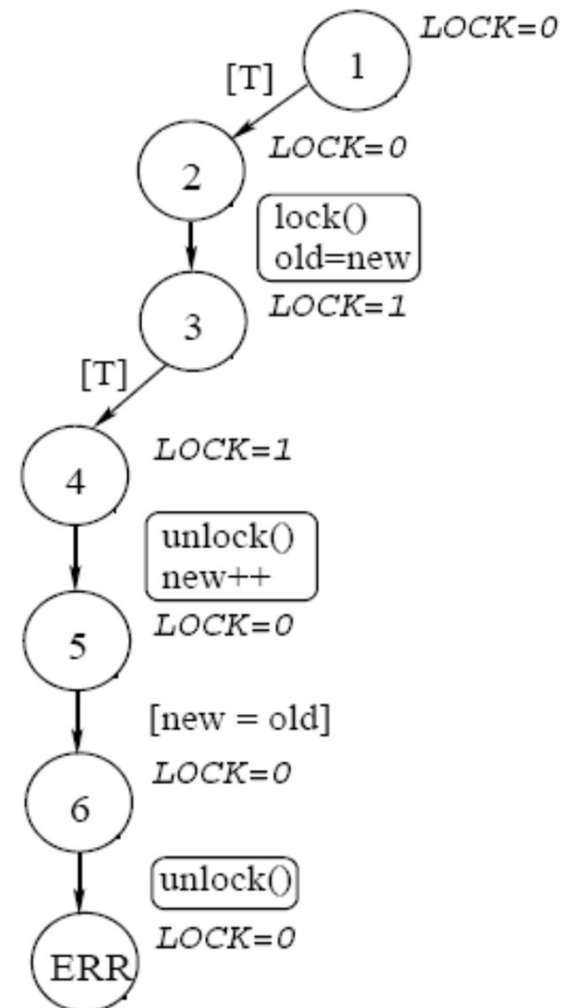
Is the Error Real?

- Use weakest preconditions to find out the weakest precondition that leads to the error
 - If the weakest precondition is false, there is no initial program condition that can lead to the error
 - Therefore the error is spurious
- Blast uses a variant of weakest preconditions
 - creates a new variable for each assignment before using weakest preconditions
 - Instead of substituting on assignment, adds new constraint
 - Helps isolate the reason for the spurious error more effectively



Is the Error Real?

- assume True;
- lock();
- old = new;
- assume True;
- unlock();
- new++;
- assume new==old
- error (lock==0)





Model Locking as Assignment

- `assume True;`
- `lock = 1;`
- `old = new;`
- `assume True;`
- `lock = 0;`
- `new = new + 1;`
- `assume new==old`
- `error (lock==0)`



Index the Variables

- `assume True;`
- `lock1 = 1`
- `old1 = new1;`
- `assume True;`
- `lock2 = 0`
- `new2 = new1 + 1`
- `assume new2==old1`
- `error (lock2==0)`

Generate Weakest Preconditions



- | | | |
|-----------------------|---|--|
| • assume True; | $\wedge \text{True}$ | |
| • lock1 = 1 | $\wedge \text{lock1} == 1$ | |
| • old1 = new1; | $\wedge \text{old1} == \text{new1}$ | |
| • assume True; | $\wedge \text{True}$ | |
| • lock2 = 0 | $\wedge \text{lock2} == 0$ | |
| • new2 = new1 + 1 | $\wedge \text{new2} == \text{new1} + 1$ | |
| • assume new2 == old1 | $\wedge \text{new2} == \text{old1}$ | |
| • error (lock2 == 0) | $\text{lock2} == 0$ | |
- Contradictory!**
- Diagram showing arrows pointing from the text "Contradictory!" to the expressions $\wedge \text{old1} == \text{new1}$ and $\wedge \text{new2} == \text{old1}$, indicating a logical contradiction between these two conditions.



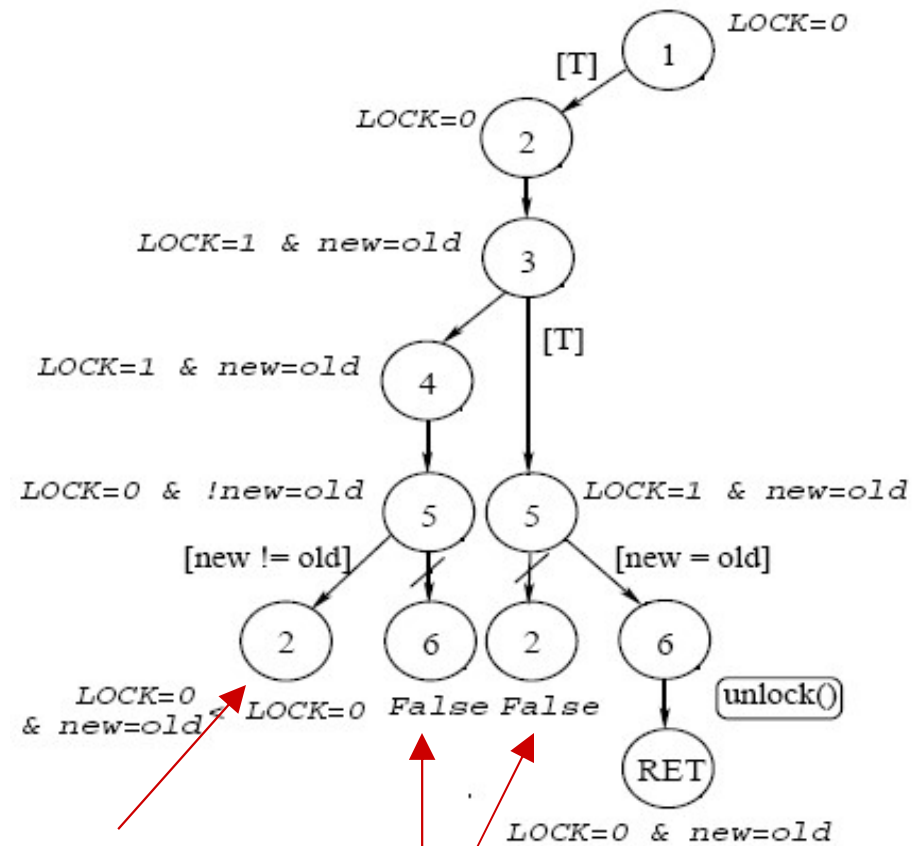
Why is the Error Spurious?

- More precisely, what predicate could we track that would eliminate the spurious error message?
 - Consider, for each node, the constraints generated before that node (c1) and after that node (c2)
 - Find a condition I such that
 - $c1 \Rightarrow I$
 - I is true at the node
 - I only contains variables mentioned in both c1 and c2
 - I mentions only variables in scope (not old or future copies)
 - $I \wedge c2 = \text{false}$
 - I is enough to show that the rest of the path is infeasible
 - I is guaranteed to exist
 - See Craig Interpolation
- $\wedge \text{True}$
 - $\wedge \text{lock1} == 1$
 - $\wedge \text{old1} == \text{new1}$ ← Interpolant: $\text{old} == \text{new}$
 - $\wedge \text{True}$
 - $\wedge \text{lock2} == 0$
 - $\wedge \text{new2} == \text{new1} + 1$
 - $\wedge \text{new2} == \text{old1}$
 - $\text{lock2} == 0$



Reanalyzing the Program

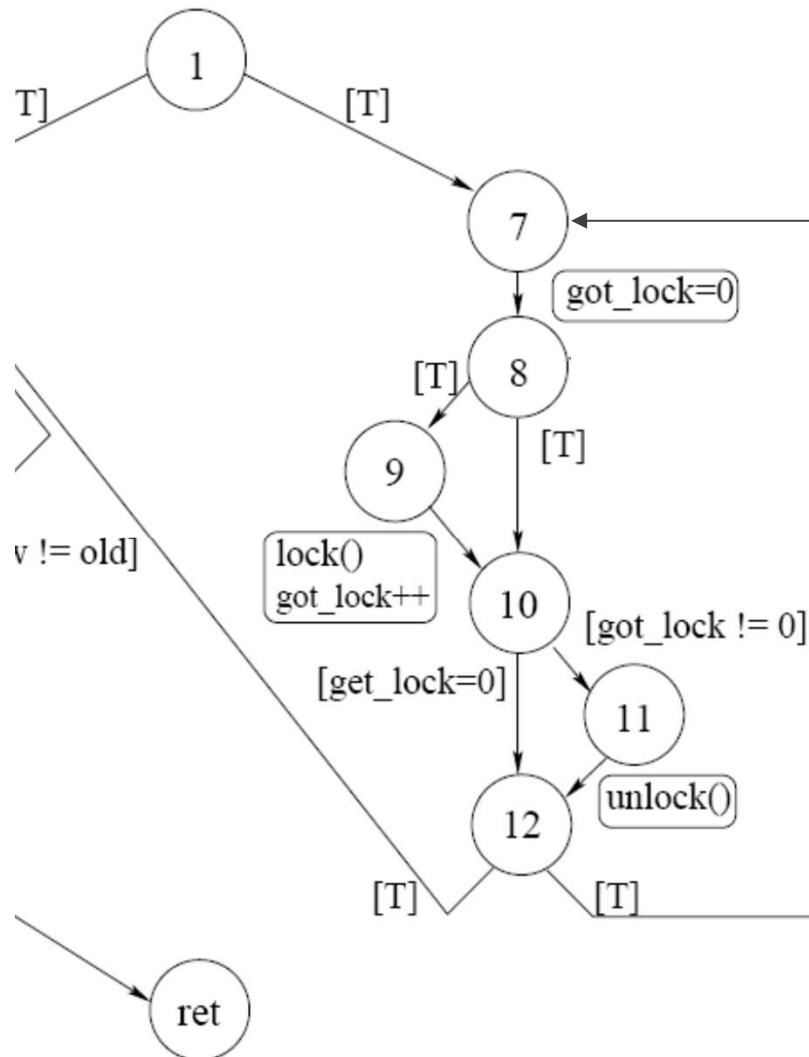
- Explore a subtree again
 - Start where new predicates were discovered
 - This time, track the new predicates
 - If the conjunction of the predicates on a node is false, stop exploring—this node is unreachable



Unreachable



Analyzing the Right Hand Side



Exercise: run weakest preconditions from the unlock() at the end of the path 1-7-8-10-11-12.

Recall that we model locking with a variable *lock*, so unlock() is an error if *lock*==0

Generate Weakest Preconditions



- `assume True;`
- `got_lock = 0;`
- `assume True;`
- `assume got_lock != 0;`
- `error (lock==0)`

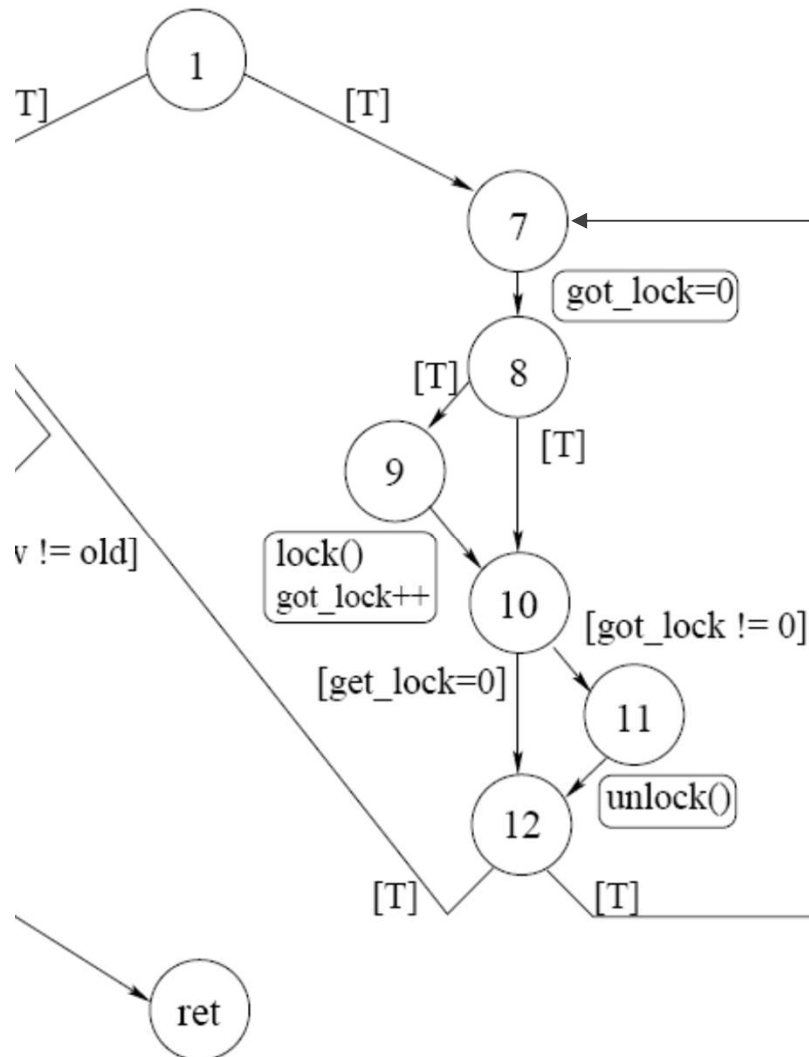


Why is the Error Spurious?

- More precisely, what predicate could we track that would eliminate the spurious error message?
- Consider, for each node, the constraints generated before that node ($c1$) and after that node ($c2$)
- Find a condition I such that
 - $c1 \Rightarrow I$
 - I is true at the node
 - I only contains variables mentioned in both $c1$ and $c2$
 - I mentions only variables in scope (not old or future copies)
 - $I \wedge c2 = \text{false}$
 - I is enough to show that the rest of the path is infeasible
 - I is guaranteed to exist
 - See Craig Interpolation
- $\wedge \text{True}$
- $\wedge \text{got_lock} == 0$
- $\wedge \text{True}$
- $\wedge \text{got_lock} != 0$
- $\text{lock} == 0$

Exercise: now find the Craig interpolant

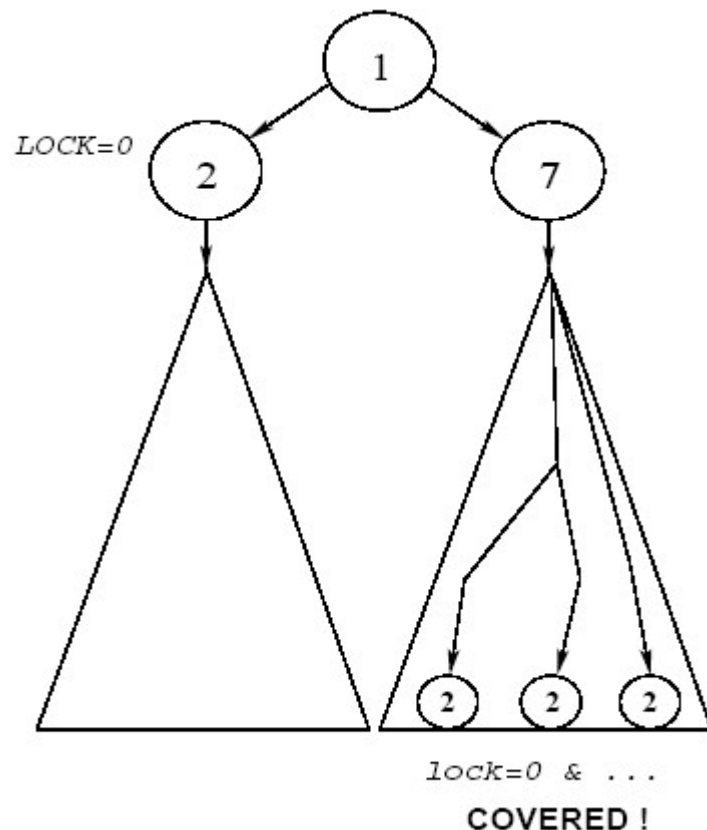
Reanalysis



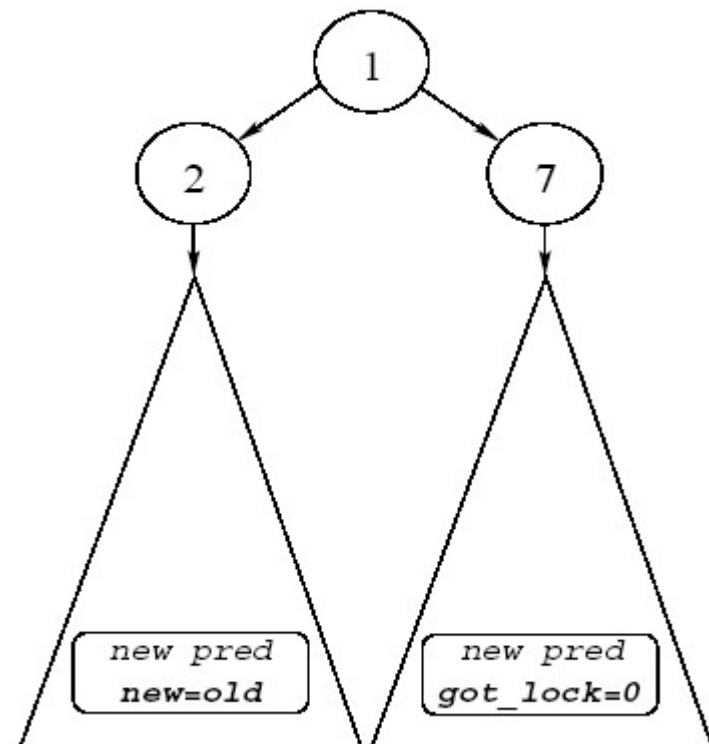
Blast Techniques, Graphically



- Explores reachable state, not all paths
 - Stops when state already seen on another path



- Lazy Abstraction
 - Uses predicates on demand
 - Only applies predicate to relevant part of tree





Termination

- Not guaranteed
 - The system could go on generating predicates forever
- Can guarantee termination
 - Restrict the set of possible predicates to a finite subset
 - Finite height lattices in data flow analysis!
 - Those predicates are enough to predict observable behavior of program
 - E.g. the ordering of lock and unlock statements
 - Predicates are restricted in practice
 - E.g. likely can't handle arbitrary quantification as in Dafny
 - Model checking is hard if properties depend on heap data, for example
 - Can't prove arbitrary properties in this case
- In practice
 - Terminate abstraction refinement after a time bound



Key Points of CEGAR

- To prove a property, may need to strengthen it
 - Just like strengthening induction hypothesis
- CEGAR figures out strengthening automatically
 - From analyzing why errors are spurious
- Blast uses *lazy abstraction*
 - Only uses an abstraction in the parts of the program where it is needed
 - Only builds the part of the abstract state that is reached
 - Explored state space is ***much*** smaller than potential state space

Experimental Results



Program	Postprocessed LOC	Predicates		BLAST Time (sec)	Ctrex analysis (sec)	Proof Size (bytes)
		Total	Active			
qpmouse.c	23539	2	2	0.50	0.00	175
ide.c	18131	5	5	4.59	0.01	253
aha152x.c	17736	2	2	20.93	0.00	
tlan.c	16506	5	4	428.63	403.33	405
cdaudio.c	17798	85	45	1398.62	540.96	156787
floppy.c	17386	62	37	2086.35	1565.34	
[fixed]		93	44	395.97	17.46	60129
kbfiltr.c	12131	54	40	64.16	5.89	
		48	35	256.92	165.25	
[fixed]		37	34	10.00	0.38	7619
mouclass.c	17372	57	46	54.46	3.34	
parport.c	61781	193	50	1980.09	519.69	102967



Blast in Practice

- Has scaled past 100,000 lines of code
 - Realistically starts producing worse results after a few 10K lines
- Sound up to certain limitations
 - Assumes restricted (“safe”) use of C
 - No aliases of different types; how realistic?
 - No recursion, no function pointers
 - Need models for library functions
- Has also been used to find memory safety errors, race conditions, generate test cases