Counterexample Guided Abstraction Refinement in Blast

Optional reading: *Checking Memory Safety with Blast*

17-355/17-665: Program Analysis
Jonathan Aldrich
How would you analyze this?

Example() {
  if (*){
    do {
      got_lock = 0;
      if (*){
        lock();
        got_lock++;
      }
      if (got_lock){
        unlock();
      }
    } while (*)
  }
}
How would you analyze this?

```c
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

Program → Abstract Program → Model Checker

- Abstract Using Predicates
- New Predicates
- Error Found
- Infeasible
- Feasible
- No Error
- Property Holds
- Report Bug
- Generate New Predicates
CEGAR:
Counterexample Guided Abstraction Refinement

- Begin with control flow graph abstraction
- Check reachability of error nodes
  - Typically take cross product of dataflow abstraction and CFG
  - However, can encode dataflow abstraction in CFG through error nodes—assert(false)
- 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!
  - 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;

Program Analysis - Spring 2017
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; \(\land\) True
- lock1 = 1 \(\land\) lock1==1
- old1 = new1; \(\land\) old1==new1
- assume True; \(\land\) True
- lock2 = 0 \(\land\) lock2==0
- new2 = new1 + 1 \(\land\) new2==new1+1
- assume new2==old1 \(\land\) new2==old1
- error (lock2==0) \(\land\) lock2==0

Contradictory!
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)
  • \( I \land c2 = \text{false} \)
    • \( I \) is enough to show that the rest of the path is infeasible
  • \( I \) is guaranteed to exist
    • See Craig Interpolation

\[
\land \text{True} \\
\land \text{lock1==1} \\
\land \text{old1==new1} \\
\land \text{True} \\
\land \text{lock2==0} \\
\land \text{new2==new1+1} \\
\land \text{new2==old1} \\
\text{lock2==0}
\]

Interpolant: \( \text{old == new} \)
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
Reanalysis Example

Already Covered

Unreachable
Analyzing the Right Hand Side
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 => 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 ^ c2 = false
    - I is enough to show that the rest of the path is infeasible
  - I is guaranteed to exist
    - See Craig Interpolation

- ^ True
- ^ got_lock==0
- ^ True
- ^ got_lock!=0
- lock==0
Reanalysis

Key: L = locked=1
Z = got_lock=0
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

<table>
<thead>
<tr>
<th>Program</th>
<th>Postprocessed LOC</th>
<th>Predicates</th>
<th>BLAST Time (sec)</th>
<th>Ctrex analysis (sec)</th>
<th>Proof Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>qpmouse.c</td>
<td>23539</td>
<td>2</td>
<td>0.50</td>
<td>0.00</td>
<td>175</td>
</tr>
<tr>
<td>ide.c</td>
<td>18131</td>
<td>5</td>
<td>4.59</td>
<td>0.01</td>
<td>253</td>
</tr>
<tr>
<td>aha152x.c</td>
<td>17736</td>
<td>2</td>
<td>20.93</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>tlan.c</td>
<td>16506</td>
<td>5</td>
<td>428.63</td>
<td>403.33</td>
<td>405</td>
</tr>
<tr>
<td>cdaudio.c</td>
<td>17798</td>
<td>85</td>
<td>1398.62</td>
<td>540.96</td>
<td>156787</td>
</tr>
<tr>
<td>floppy.c [fixed]</td>
<td>17386</td>
<td>62</td>
<td>2086.35</td>
<td>1565.34</td>
<td></td>
</tr>
<tr>
<td>kbfiltre.c</td>
<td>12131</td>
<td>93</td>
<td>395.97</td>
<td>17.46</td>
<td>60129</td>
</tr>
<tr>
<td>[fixed]</td>
<td></td>
<td>54</td>
<td>64.16</td>
<td>5.89</td>
<td></td>
</tr>
<tr>
<td>mouclass.c</td>
<td>17372</td>
<td>48</td>
<td>256.92</td>
<td>165.25</td>
<td>7619</td>
</tr>
<tr>
<td>parport.c</td>
<td>61781</td>
<td>193</td>
<td>1980.09</td>
<td>519.69</td>
<td>102967</td>
</tr>
</tbody>
</table>
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 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