Software Model Checking and Counter-example Guided Abstraction Refinement

Claire Le Goues
Motivation: How should we 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: How should we analyze this?

Example() {
    if (*){
        do {
            got_lock = 0;
            if (*){
                lock();
                got_lock++;
            }
        } while (*)
    }
    if (got_lock){
        unlock();
    }
}
Tradeoffs...

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

Symbolic execution shows need to eliminate infeasible paths, see lock/unlock on correlated branches (more complicated logic!).

Dataflow analysis requires fixed abstractions, e.g., zero/non-zero, locked/unlocked.

Explicit-state Model Checking needs programs to be represented as a finite state model...state explosion??
Enter: Abstraction Refinement

- 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.
- ...what will we lose?
  - Answer: Termination guarantees. OH WELL.
CEGAR: Counterexample Guided Abstraction Refinement

Begin with a coarse abstraction

Program, Property Spec -> Abstract Program

Abstract Program -> Model Checker

Model Checker -> Path Feasibility Checker

Path Feasibility Checker -> Report Bug

Check for property violation.

No Error -> Property Holds

Error Found -> Feasible

Feasible -> Report Bug

Infeasible -> New Predicates

New Predicates -> Generate New Predicates

Generate New Predicates -> Refine abstraction to exclude infeasible “error” path

Refine abstraction to exclude infeasible “error” path -> Error Found

Is the error path actually feasible? Hint: weakest preconditions!

Begin with a coarse abstraction

Abstract Program Using Predicates
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

“User applications must not run with root privilege”
When `execv` is called, must have `suid ≠ 0`
Property 3 : IRP Handler

[Diagram of IRP Handler with nodes and edges]
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();
6: return;
}
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;
}

Original program violates spec iff new program reaches ERR

Example ( ) {
1: do{
   if L=1 goto ERR;
   else L=1;
   old = new;
   q = q->next;
2:   if (q != NULL){
3:      q->data = new;
      if L=0 goto ERR;
      else L=0;
      new ++;
   }
4: } while(new != old);
5: if L=0 goto ERR;
else L=0;
return;
ERR: abort();
}
Program As Labeled Transition System

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; }

pc \rightarrow 3
lock \rightarrow 
old \rightarrow 5
new \rightarrow 5
q \rightarrow 0x133a

3: unlock();
new++;

pc \rightarrow 4
lock \rightarrow 
old \rightarrow 5
new \rightarrow 6
q \rightarrow 0x133a
The Safety Verification Problem

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

Problem: Infinite state graph (old=1, old=2, old=...)

Solution: Set of states $\models$ logical formula
Representing [Sets of States] as *Formulas*

<table>
<thead>
<tr>
<th>([F]) states satisfying (F) ({s \mid s \models F})</th>
<th>(F) FO fmla over prog. vars</th>
</tr>
</thead>
<tbody>
<tr>
<td>([F_1] \cap [F_2])</td>
<td>(F_1 \land F_2)</td>
</tr>
<tr>
<td>([F_1] \cup [F_2])</td>
<td>(F_1 \lor F_2)</td>
</tr>
<tr>
<td>(\overline{F})</td>
<td>(\neg F)</td>
</tr>
<tr>
<td>([F_1] \subseteq [F_2])</td>
<td>(F_1 \Rightarrow F_2)</td>
</tr>
</tbody>
</table>

i.e. \(F_1 \land \neg F_2\) unsatisfiable
Idea 1: Predicate Abstraction

- **Predicates** on program state: 
  - `lock` \(\text{(i.e., lock=true)}\)
  - `old = new`

- States satisfying **same** predicates are **equivalent**
  - Merged into one **abstract state**

- **#abstract states is finite**
  - Thus model-checking the abstraction will be feasible!
Abstract States and Transitions

State

Theorem Prover

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

lock \mapsto \text{false}
old \mapsto 5
new \mapsto 5
q \mapsto 0x133a

pc \mapsto 3

\neg lock \mapsto \text{true}
\neg old \mapsto \text{new}

pc \mapsto 4

\neg lock \mapsto \text{false}
old \mapsto 5
new \mapsto 6
q \mapsto 0x133a

\neg old \mapsto \text{new}
Abstraction

State

3: unlock();
new++;

4: }

pc \mapsto 3
lock \mapsto \bullet
old \mapsto 5
new \mapsto 5
q \mapsto 0x133a

pc \mapsto 4
lock \mapsto \circ
old \mapsto 5
new \mapsto 6
q \mapsto 0x133a

Existential Lifting
(i.e., A_1 \to A_2 \text{ iff } \exists c_1 \in A_1. \exists c_2 \in A_2. c_1 \to c_2)

Theorem Prover

\rightarrow lock
\rightarrow old=new

A_1

\rightarrow A_2

\text{lock}
old=new

A_1

\rightarrow A_2

\text{lock}
old=new
Abstraction

State

3: unlock();
4: }

pc → 3
lock → 5
old → 5
new → 5
q → 0x133a

pc → 4
lock → 5
old → 5
new → 6
q → 0x133a

lock
old=new

→ lock
→ old=new
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

Solution
Use spurious *counterexamples* to *refine* abstraction

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

[Kurshan et al 93] [Clarke et al 00] [Ball-Rajamani 01]
Problem: Abstraction is Expensive

Problem
\#abstract states = 2\#predicates
Exponential Thm. Prover queries

Observe
Fraction of state space reachable
\#Preds \sim 100’s, \#States \sim 2^{100}, \#Reach \sim 1000’s
**Solution 1:** Only Abstract Reachable States

Problem

#abstract states = 2 \#predicates

Exponential Thm. Prover queries

Solution

Build abstraction during search
Solution 2: Don’t Refine Error-Free Regions

Problem

#abstract states = 2
#predicates
Exponential Thm. Prover queries

Solution

Don’t refine error-free regions
Build reachability tree.

• Generate Abstract Reachability Tree
  o Contains all reachable nodes
  o Annotates each node with state
    ▪ Initially LOCK = 0 or LOCK = 1
    ▪ Cross product of CFA and data flow abstraction

• Algorithm: depth-first search
  o Generate nodes one by one
  o 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
  o If you come to an error node, stop
Less abstractly: build reachability tree

2: do {
    lock();
    old = new;
3:     if (*){
4:         unlock();
         new++;
7:     }
5: } while (new != old);
6: unlock();
   return;
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.

Error Free
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.

Error Free

SAFE

S1: Only Abstract Reachable States
S2: Don’t refine error-free regions
Less abstractly: build reachability tree

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

Reachability Tree

Predicates: LOCK
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();
}
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();
}

Reachability Tree

Predicates: LOCK
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();
}

Reachability Tree

Predicates: LOCK

Predicates: LOCK

Reachability Tree

Predicates: LOCK
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 ();
}

Reachability Tree

Predicates: LOCK
Example ( ) {
  1: do {
    lock();
    old = new;
    q = q->next;
  2:   if (q != NULL) {
    3:     q->data = new;
    4:       unlock();
    5:       new ++;
  6:   }
  7: while (new != old);
  8: }
  9: unlock();
}

Reachability Tree

Predicates: LOCK
Depth First Search Example
Is the Error Real?

• Use weakest preconditions to find out the weakest precondition that leads to the error
  o If the weakest precondition is false, there is no initial program condition that can lead to the error
  o Therefore the error is spurious

• Blast uses a variant of weakest preconditions
  o creates a new variable for each assignment before using weakest preconditions
  o Instead of substituting on assignment, adds new constraint
  o 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;
- lock1 = 1
- old1 = new1;
- assume True;
- lock2 = 0
- new2 = new1 + 1
- assume new2==old1
- error (lock2==0)

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

Contradictory!
Relevant Sidebar: Craig Interpolation

• Given an unsatisfiable formula $A \land B$, the Craig Interpolant $I$ is a formula such that:
  o $A \rightarrow I$
  o $I \land B$ is unsatisfiable
  o $I$ only refers to variables mentioned in both $A$ and $B$
• It is guaranteed to exist, proof elided.

• $\land True$
• $\land lock1==1$
• $\land old1==new1$
• $\land True$
• $\land lock2==0$
• $\land new2==new1+1$
• $\land new2==old1$
• $lock2==0$
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
  o c1 => I
    ▪ I is true at the node
  o I only contains variables mentioned in both c1 and c2
    ▪ I mentions only variables in scope (not old or future copies)
  o I \( \land \) c2 = false
    ▪ I is enough to show that the rest of the path is infeasible
  o I is guaranteed to exist
    ▪ See Craig Interpolation

• \( \land \) True
• \( \land \) lock1==1
• \( \land \) old1==new1
• \( \land \) True
• \( \land \) lock2==0
• \( \land \) new2==new1+1
• \( \land \) new2==old1
• lock2==0

Interpolant: old == new
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();
}

Reachability Tree

Predicates: LOCK
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

Inconsistent
new == old

old = new

new++

[new==old]
Reanalyzing the Program

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

1. **lock()**
   - old = new
2. **[T]**
3. **[T]**
4. **[new != old]**
5. **[new = old]**
6. **unlock()**
7. **ret**

Already Covered

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
Reanalysis

Example()
{
  if (*){
    do {
      got_lock = 0;
      if (*){
        lock();
        got_lock++;  
      }
      if (got_lock){
        unlock();
      }
    } while (*)
  }
}

Key: L = locked=1  
  Z = got_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)
  - $I \land c2 = false$
    - I is enough to show that the rest of the path is infeasible
  - I is guaranteed to exist
    - See Craig Interpolation

- $\land True$
- $\land got\_lock==0$
- $\land True$
- $\land got\_lock!=0$
- lock==0

Exercise: now find the Craig interpolant
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 ();
}

Reachability Tree

Predicates:  *LOCK, new==old*

...but only at the minimum suffix!
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();
}

Reachability Tree

Predicates:  \( \text{LOCK, new==old} \)
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
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

SAFE
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.

Error Free

S1: Only Abstract Reachable States
S2: Don’t refine error-free regions
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
## 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></td>
<td></td>
<td>Total</td>
<td>Active</td>
<td></td>
<td></td>
</tr>
<tr>
<td>qpmouse.c</td>
<td>23539</td>
<td>2</td>
<td>2</td>
<td>0.50</td>
<td>0.00</td>
</tr>
<tr>
<td>ide.c</td>
<td>18131</td>
<td>5</td>
<td>5</td>
<td>4.59</td>
<td>0.01</td>
</tr>
<tr>
<td>aha152x.c</td>
<td>17736</td>
<td>2</td>
<td>2</td>
<td>20.93</td>
<td>0.00</td>
</tr>
<tr>
<td>tlan.c</td>
<td>16506</td>
<td>5</td>
<td>4</td>
<td>428.63</td>
<td>403.33</td>
</tr>
<tr>
<td>cdaudio.c</td>
<td>17798</td>
<td>85</td>
<td>45</td>
<td>1398.62</td>
<td>540.96</td>
</tr>
<tr>
<td>floppy.c</td>
<td>17386</td>
<td>62</td>
<td>37</td>
<td>2086.35</td>
<td>1565.34</td>
</tr>
<tr>
<td>[fixed]</td>
<td></td>
<td>93</td>
<td>44</td>
<td>395.97</td>
<td>17.46</td>
</tr>
<tr>
<td>kbfiltr.c</td>
<td>12131</td>
<td>54</td>
<td>40</td>
<td>64.16</td>
<td>5.89</td>
</tr>
<tr>
<td>[fixed]</td>
<td></td>
<td>48</td>
<td>35</td>
<td>256.92</td>
<td>165.25</td>
</tr>
<tr>
<td>mouclass.c</td>
<td>17372</td>
<td>57</td>
<td>46</td>
<td>54.46</td>
<td>3.34</td>
</tr>
<tr>
<td>parport.c</td>
<td>61781</td>
<td>193</td>
<td>50</td>
<td>1980.09</td>
<td>519.69</td>
</tr>
</tbody>
</table>
Termination

• Not guaranteed
  o The system could go on generating predicates forever

• Can guarantee termination
  o Restrict the set of possible predicates to a finite subset
    ▪ Finite height lattices in data flow analysis!
  o 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
  o Can’t prove arbitrary properties in this case

• In practice
  o Terminate abstraction refinement after a time bound
Key Points of CEGAR

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