Scalable Program Verification by Lazy Abstraction

Ranjit Jhala
U.C. Berkeley

Mars, July 4, 1997
Lost contact due to real-time priority inversion bug

Mars, December 3, 1999
Crashed due to uninitialized variable
French Guyana, June 4, 1996
$600 million software failure

Something Reliable

Uptime: 67 years
Why don’t Bridges Crash?

Building Blocks
1. Relevant facts* Mass, Tensile Strength
2. Model Free Body Diagram
3. Analysis Solve Equations

Bridges
Mechanics

Programs
Logic

* w.r.t. property of interest

Contributions

C Program → BLAST

Yes → Safe

No → Trace
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

A data race on $x$ is a state where:

1. Two threads can access $x$
2. One of the accesses is a write

There should be no races on shared variables
Contributions

Sequential Programs
Counterex.-Guided Abstraction-Refinement
For large programs, complex properties
New Algorithms: Abstraction [POPL 02], Refinement [POPL 04]

Property 1: Double Locking (Linux/Windows Drivers)
Property 2: Drop Root Privilege (Linux Daemons ~59kloc)
  - Precise: No false Errors
Property 3: IRP Handler (NT Drivers ~130Kloc)
  - Large Programs

Contributions

Multithreaded Programs
New models for thread interactions
New algorithms to compute models and
Verify multithreaded programs [CAV 03] [PLDI 04]

Property 4: Data Races
- Linux/Windows Drivers
- Sensor Network Apps. (TinyOS/NesC) ~10kloc
- Arbitrarily many threads
- Any synchronization mechanisms
- Real counterexamples, Safety Proofs
Plan
1. C.G. Abstraction-Refinement

2. Lazy Abstraction
   • Sequential Programs
   • Multithreaded Programs

3. Future Work

Example

Example ( ) {
  1: do{
      lock();
      old = new;
      q = q->next;
      2:   if (q != NULL){
      3:     q->data = new;
      unlock();
      new ++;
      }
  } while(new != old);
  4: return;
}
What a program *really* is...

The Safety Verification Problem

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

Problem: Infinite state graph

Solution: Set of states' logical formula
Idea 1: Predicate Abstraction

- **Predicates** on program state:
  - `lock`
  - `old = new`

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

- `#abstract states is finite`

[Graf-Saidi 97]

Abstract States and Transitions
Analyze Abstraction

Analyze finite graph
Over Approximate:
Safe System Safe
No false negatives

Problem
Spurious counterexamples

Idea 2: Counterex.-Guided Refinement

Solution
Use spurious counterexamples to refine abstraction!

[Kurshan et al. 93] [Clarke et al. 00]
[Ball-Rajamani 01]
Idea 2: Counterex. -Guided Refinement

Solution
Use spurious counterexamples to refine abstraction

1. Add predicates to distinguish states across cut
2. Build refined abstraction
- eliminates counterexample

Idea 2: Counterex.-Guided 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
   - Till real counterexample or system proved safe

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
   - Till real counterexample or system proved safe
Plan

1. C.G. Abstraction-Refinement
2. Lazy Abstraction
   - Sequential Programs [POPL 02] [POPL04]
   - Multithreaded Programs
3. Future Work

Scaling Sequential Verification

<Diagram of BLAST process>

- C Program → Yes → Safe
- Property → No → Trace
**Problem: Abstraction is Expensive**

- Problem
  - \#abstract states = \(2^{\#predicates}\)
  - Exponential Thm. Prover queries

- Observe
  - Fraction of state space reachable
  - \#Preds ~ 100’s, \#States ~ \(2^{100}\),
  - \#Reach ~ 1000’s

**Solution1: Only Abstract Reachable States**

- Problem
  - \#abstract states = \(2^{\#predicates}\)
  - Exponential Thm. Prover queries

- Solution
  - Build abstraction **during** search

- Safe

Reachable
**Solution 2: Don’t Refine Error-Free Regions**

**Problem**

- #abstract states = $2^\#predicates$
- Exponential Thm. Prover queries

**Solution**

Don’t refine error-free regions

---

**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
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

Reachability Tree

Predicates: LOCK

lock()
old = new
q=q->next

: LOCK

: LOCK

1

1—→2

2
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
Build-and-Search

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

Reachability Tree

Predicates: LOCK
Analyze Counterexample

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: unlock ();
}

Predicates: LOCK

Reachability Tree

Inconsistent
new == old

Analyze Counterexample

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: unlock ();
}

Predicates: LOCK

Reachability Tree

Inconsistent
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 ();
}

Reachability Tree

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 ();
}

Reachability Tree

Predicates: \textit{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 ();
}

Reachability Tree

Predicates: \textit{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 ();
}

Reachability Tree

Predicates: LOCK, new==old

SAFE

Reachability Tree

Predicates: LOCK, new==old
Scaling Sequential Verification

Abstract

Refine

Problem: Abstraction is Expensive

Solution: 1. Abstract reachable states,
2. Avoid refining error-free regions

Key Idea: Reachability Tree

Results

<table>
<thead>
<tr>
<th>Program</th>
<th>Lines*</th>
<th>Previous Time (mins)</th>
<th>Time (mins)</th>
<th>Predicates Total</th>
<th>Predicates Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>kbfiltr</td>
<td>12k</td>
<td>1</td>
<td>3</td>
<td>72</td>
<td>6.5</td>
</tr>
<tr>
<td>floppy</td>
<td>17k</td>
<td>7</td>
<td>25</td>
<td>240</td>
<td>7.7</td>
</tr>
<tr>
<td>diskprf</td>
<td>14k</td>
<td>5</td>
<td>13</td>
<td>140</td>
<td>10</td>
</tr>
<tr>
<td>cdaudio</td>
<td>18k</td>
<td>20</td>
<td>23</td>
<td>256</td>
<td>7.8</td>
</tr>
<tr>
<td>parport</td>
<td>61k</td>
<td>DNF</td>
<td>74</td>
<td>753</td>
<td>8.1</td>
</tr>
<tr>
<td>parclss</td>
<td>138k</td>
<td>DNF</td>
<td>77</td>
<td>382</td>
<td>7.2</td>
</tr>
</tbody>
</table>

* Pre-processed

Property3: IRP Handler
Win NT DDK
Analyzing Programs

1. Relevant facts
2. Model
3. Analysis

* w.r.t. property of interest

Plan

1. C.G. Abstraction-Refinement
2. Lazy Abstraction
   • Sequential Programs [POPL 02, POPL 04]
   • Multithreaded Programs
3. Future Work
Multithreaded Programs

Curse of Interleaving
- Non-deterministic scheduling
- Exponentially many behaviors
- Errors are hard to detect, reproduce, eliminate

Testing exercises a tiny fraction of possible behaviours

Data Races

A data race on x is a state where:
- Two threads can access x
- One of the accesses is a write

Unpredictable, undesirable program
Brute Force Approach

Model Checking: Explore (abstract) State Space
- The curse of Interleavings
- \#Control Combinations = m.n
  - 250,000 if 500 lines/thread, ignoring predicates
  - 3,4,5,...,k threads ? Unbounded threads ?

A Thread-Modular Approach

Key Idea: Summarize each thread
- Interactions with others w.r.t. property

Automaton on predicates on global variables
A Thread-Modular Approach

Problem: Find Summary which:
1. [Scalability] is small
2. [Verification] has all behaviors of thread

Verify (Thread || Other’s Summary)

- Control Combinations: Thread £ Summary
  - Small (if summary is small)
Check that Summaries are Valid

Thread-Modular Verification

Assume-Guarantee
[Owicki-Gries 73]
[Jones 83] [Stark 85]
[Abadi-Lamport 93]
[Alur-Henzinger 96]
[McMillan 97]
[Flanagan-Qadeer 01]

Q: Finding Summaries?
Data Races in NesC Programs [PLDI 04]

- PL for Networked Embedded Systems [Gay et al. 03]
  - TinyOS Sensor Networks Applications

- Interrupts fire events, which fire other events or post tasks which run asynchronously

- Race-freedom important
  - Non-trivial synchronization idioms
  - Flow-based analysis

- Compiled to C

Case Study: sense.nc [PLDI 04]

```ncl
atomic{
  old:= state;
  if(state==0){
    state:=1;
    if (old == 0){
      about to write x
    }
  }
  ...;
  if(old==0){
    x++;
    ...
  }
}
```

Interrupt 1 fires
- old := state
- if (state == 0){
  state := 1
  if (old == 0){
    about to write x
  }
  ...
  if(old==0){
    x++;
    ...
  }
}

Interrupt 1 handler
- disables interrupt 2

BLAST finds information
- proves no races

Interrupt 2 fires
- state := 0
- Interrupt 1 fires
  - old := state
  - if (state == 0){
    state := 1
    if (old == 0){
      about to write x
    }
    ...
  }
  ...
Analyzing Programs

Building Blocks
1. Relevant facts*
2. Model
3. Analysis

Programs
Logic
Predicates
Reach Tree
Search

Multithreaded
Predicates
Summary
Thread-Modular

* w.r.t. property of interest