Having a BLAST with SLAM

Meeting 4, CSCI 5535, Fall 2013

Announcements

- Homework 0 due Sat
  - Questions?

- Move Tue office hours to 4-5pm
Software Model Checking via Counterexample Guided Abstraction Refinement

There are easily dozens of papers.

We will skim.

SLAM Overview/Review

- **Input:**
  - Specification - properly to check: “no deadlocks”
  - “program uses lock correctly”
  - “I am webserver”
  - program (programs, device drivers)

- **Output:**
  - verified = program has no deadlocks
  - counterexample = path that may result in error that violates the spec
SLAM Overview

- **Input:** Program and Specification
  - Standard C Program (pointers, procedures)
  - Specification = Partial Correctness
    - Given as a finite state machine (typestate)
    - “I use locks correctly”, not “I am a webserver”

- **Output:** Verified or Counterexample
  - Verified = program does not violate spec
    - Can come with proof!
  - Counterexample = concrete bug instance
    - A path through the program that violates the spec

Take-Home Message

- SLAM is a software model checker. It abstracts C programs to boolean programs and model-checks the boolean programs.
- No errors in the boolean program implies no errors in the original.
- An error in the boolean program may be a real bug. Or SLAM may refine the abstraction and start again.
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

Example SLAM Input

```c
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;
}
```
SLAM in a Nutshell

\[
\text{SLAM}(\text{Program } p, \text{Spec } s) = \\
\text{Program } q = \text{incorporate_spec}(p, s); \quad \text{// slic} \\
\text{PredicateSet } \text{abs} = \{ \}; \\
\text{while } \text{true} \text{ do} \\
\quad \text{BooleanProgram } b = \text{abstract}(q, \text{abs}); \quad \text{// c2bp} \\
\quad \text{match } \text{model_check}(b) \text{ with } \text{// bebop} \\
\qquad | \text{No_Error} \leftrightarrow \text{print("no bug"); exit(0)} \\
\qquad | \text{Counterexample}(c) \leftrightarrow \\
\qquad \quad \text{if } \text{is_valid_path}(c, p) \text{ then } \text{// newton} \\
\qquad \quad \text{print("real bug"); exit(1)} \\
\qquad \quad \text{else } \\
\qquad \quad \text{abs } \not\subseteq \text{abs} \cup \text{new_preds}(c) \quad \text{// newton} \\
\text{done}
\]

Incorporating Specs

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;
}

Ideas?
Incorporating Specs

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

Original program violates spec iff new program reaches ERR

Program As Labeled Transition System

State

Transition

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

Representing [Sets of States] as Formulas

<table>
<thead>
<tr>
<th>([F]) states satisfying (F) ({s \mid s \in F})</th>
<th>(F) FO formula over program vars</th>
</tr>
</thead>
<tbody>
<tr>
<td>([F_1] \cap [F_2])</td>
<td>(F_1 \wedge F_2)</td>
</tr>
<tr>
<td>([F_1] \cup [F_2])</td>
<td>(F_1 \vee F_2)</td>
</tr>
<tr>
<td>(\overline{[F]})</td>
<td>(\neg F)</td>
</tr>
<tr>
<td>([F_1] \subsetneq [F_2])</td>
<td>i.e. (F_1 \wedge \neg F_2) unsatisfiable</td>
</tr>
</tbody>
</table>
Representing [Sets of States] as Formulas

- \([F]\) states satisfying \(F\) \([s \mid s \models F]\)
- \([F_1] \cap [F_2]\) \(F_1 \land F_2\)
- \([F_1] \cup [F_2]\) \(F_1 \lor F_2\)
- \(\overline{F}\) \(\neg F\)
- \([F_1] \subseteq [F_2]\) \(F_1 \Rightarrow F_2\)

i.e. \(F_1 \land \neg F_2\) unsatisfiable

Idea 1: Predicate Abstraction

- **Predicates** on program state:
  - \(\text{lock}\) \((i.e., \text{lock}=\text{true})\)
  - \(\text{old} = \text{new}\)
- States satisfying **same** predicates are **equivalent**
  - Merged into one abstract state
- Num of abstract states is **finite**
  - Thus model-checking the abstraction will be feasible!
Abstract States and Transitions

State Transition

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

3: unlock();
new++;

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

Theorem Prover

lock
old=new

\neg lock
\neg old=new

Existential Lifting
(i.e., \exists A_1 \exists A_2. A_1 \rightarrow A_2 \iff \exists \mathbf{e}_1 \in A_1. \exists \mathbf{e}_2 \in A_2. \mathbf{e}_1 \rightarrow \mathbf{e}_2)

Abstraction

State Transition

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

3: unlock();
new++;

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

A_1 \rightarrow A_2

Theorem Prover

lock
old=new

\neg lock
\neg old=new
Abstraction

Theorem Prover

State Transition

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

3: unlock();
new++;

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

Analyse Abstraction

Analyze finite graph

Over Approximate
Safe \Rightarrow System Safe
No false negatives

Problem
Spurious counterexamples

false positives
Idea 2: Counterexample-Guided Refinement

Solution
Use spurious counterexamples to refine abstraction!

Imprecision due to merge

1. Add predicates to distinguish states across cut
2. Build refined abstraction
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 until real counterexample or system proved safe 25

Problem: Abstraction is Expensive

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

Observe
Fraction of state space reachable
#Preds ~ 100's, #States ~ 2100, #Reach ~ 1000's

Why?

Reachable
**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
Key Idea for Solutions?
**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.
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
SAFE

S1: Only Abstract Reachable States
S2: Don’t refine error-free regions

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
Build-and-Search

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

Reachability Tree

Predicates: LOCK

Build-and-Search

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

Reachability Tree

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

Reachability Tree
Predicates: LOCK

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

Predicates: LOCK

Reachability Tree

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
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

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

Reachability Tree
Repeat Build-and-Search

Example ( )
1: do
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();
}
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

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.

S1: Only Abstract Reachable States
S2: Don’t refine error-free regions

Repeat Build-and-Search

Reachability Tree

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.

S1: Only Abstract Reachable States
S2: Don’t refine error-free regions
Two Handwaves

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

Q. How to compute “successors”?

Q. How to find predicates?

Refinement

Predicates: LOCK, new==old

Reachability Tree

Two Handwaves

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

Q. How to compute “successors”?

Predicates: LOCK, new==old

Reachability Tree
Weakest Preconditions

\[ WP(P, OP) \]

Weakest formula \( P' \) s.t.

if \( P' \) is true before \( OP \)

then \( P \) is true after \( OP \)

More on this later in the semester!
How to compute successor?

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();
}

For each p
- Check if p is true (or false) after OP
Q: When is p false after OP?
- If WP(¬p, OP) is true before OP!
- We know F is true before OP
- Thm. Pvr. Query: F ⇒ WP(¬p, OP)

Predicates: LOCK, new == old
How to compute successor?

Example

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

Predicate: `new == old`

True?

\[ (LOCK, \text{new}=\text{old}) \implies (\text{new} + 1 = \text{old}) \]

False?

\[ (LOCK, \text{new}=\text{old}) \implies (\text{new} + 1 \neq \text{old}) \]

For each \( p \)
- Check if \( p \) is true (or false) after \( OP \)

\[ \text{Q: When is } p \text{ false after } OP? \]
- If \( WP(\neg p, OP) \) is true before \( OP \)
- We know \( F \) is true before \( OP \)
- Thm. Pvr. Query: \( F \Rightarrow WP(\neg p, OP) \)

Advanced SLAM/BLAST

Too Many Predicates
- Use Predicates Locally

Counter-Examples
- Craig Interpolants

Procedures
- Summaries

Concurrency
- Thread-Context Reasoning
SLAM Summary

1) Instrument Program With Safety Policy
2) Predicates = {} 
3) Abstract Program With Predicates
   - Use Weakest Preconditions and Theorem Prover Calls
4) Model-Check Resulting Boolean Program
   - Use Symbolic Model Checking
5) Error State Not Reachable?
   - Original Program Has No Errors: Done!
6) Check Counterexample Feasibility
   - Use Symbolic Execution
7) Counterexample Is Feasible?
   - Real Bug: Done!
8) Counterexample Is Not Feasible?
   1) Find New Predicates (Refine Abstraction)
   2) Goto Line 3

Bonus: SLAM/BLAST Weakness

```
1: F() {
  2:   int x = 0;
  3:   lock();
  4:   x++;
  5:   while (x ≠ 88);
  6:     if (x < 77)
  7:       lock();
  8: }
```

- Preds = {}, Path = 234567
- [x=0, x+1≠88, x+1<77]
- Preds = {x=0}, Path = 234567
- [x=0, x+1≠88, x+1<77]
- Preds = {x=0, x+1=88}
- Path = 23454567
- [x=0, x+2≠88, x+2<77]
- Preds = {x=0,x+1=88,x+2=88}
- Path = 2345454567
- ...
- Result: the predicates “count” the loop iterations
For Next Time

• Post about today’s class and reading