What is Model Checking?

- Another static analysis technique
- Easiest to understand when contrasted with dataflow analysis

A Sample Control-Flow Graph

- if (x == 0)
  - lock(l)
  - if (x == 0)
    - unlock(l)
    - lock(l)

Dataflow Analysis

- Consider all possible paths through the graph and the state of the locks on each path.
- if (x == 0)
  - lock(l)
  - if (x == 0)
    - unlock(l)
    - lock(l)

Dataflow Analysis

- Consider all possible paths through the graph and the state of the locks on each path.
- if (x == 0)
  - lock(l)
  - if (x == 0)
    - unlock(l)
    - lock(l)

OK
**Dataflow Analysis**

Consider all possible paths through the graph and the state of the locks on each path.

```plaintext
if (x == 0)
  lock(l)
if (x == 0)
  unlocked
unlock(l)
lock(l)
```

**Notes**

- Dataflow analysis appears exponential
  - Considers all paths through the graph
- In fact, it is linear time
- In locking example,
  - A lock can be in three states:
    - unlocked
    - locked
    - don't know
  - Either locked or unlocked
  - Summarize over all predecessors of a node

**The Example Again**

```plaintext
if (x == 0)
  lock(l)
if (x == 0)
  unlocked
unlock(l)
lock(l)
```

**The Bad Case Again**

The execution path is infeasible; it can never happen.

```plaintext
if (x == 0)
  lock(l)
if (x == 0)
  unlocked
unlock(l)
lock(l)
```

**An Idea**

- To be more accurate, we can track the state of program predicates
  - E.g., x == 0
  - Allows us to rule out infeasible paths
- This is path sensitive analysis
  - Dataflow analysis
  - Plus tracking predicates

**This Lecture**

- Overview of a modern software model checker
  - BLAST
- Elements
  - How we specify properties to check
  - How the checking is done
  - What happens when checking fails
    - Producing counter examples
The Architecture of BLAST

Specifying Properties

- Properties are specified as finite state machines
  - A very common approach
  - Limited, but useful class of properties
- Orthogonal to analysis technique
  - Many different systems use finite state machines
  - Model checking could use other specifications, too

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

Program Representation

- Many static analysis approaches work directly on program syntax
  - Type systems
  - Dataflow analysis
- Model checking represents programs as state transition systems
  - Lower-level representation
Example

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

The Model Checking View...

The Safety Verification Problem

- Error
- Safe
- Initial
  - 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-Saiz 97)
  - (Reps-Sagiv-Wilhelm 99)

Abstract States and Transitions

Abstraction

Existential Lifting
Abstraction

State

lock
old-new

lock
old-new

Analyze Abstraction

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

Problem
Spurious counterexamples

Idea 2: Counterexample-Guided Refinement

Solution
Use spurious counterexamples to refine abstraction

[Kurshan et al 93] [Clarke et al 00]
[Ball-Rajmane 01]

Idea 2: Counterexample-Guided Refinement

Solution
Use spurious counterexamples to refine abstraction

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

[Kurshan et al 93] [Clarke et al 00]
[Ball-Rajmane 01]

Iterative Abstraction-Refinement

Solution
Use spurious counterexamples to refine abstraction
1. Add predicates to distinguish states across cut
2. Build refined abstraction
3. Repeat search
Till real counterexample or system proved safe

[Kurshan et al 93] [Clarke et al 00]
[Ball-Rajmane 01]

The Next Step

- Abstraction, as described, is very expensive
- Need to be more intelligent about what we abstract and how
Problem: Abstraction is Expensive

Observe
Fraction of state space reachable
#Preds ~ 100's, #States ~ 2^100, #Reach ~ 1000's

Solution: Only Abstract Reachable States

Problem
#Abstract states = 2^#Predicates
Exponential Thm. Prover queries

Solution
Build abstraction during search

Solution2: 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

Initial

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

Initial

Unroll Abstraction
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.

SAFE

S1: Only Abstract Reachable States
S2: Don't refine error-free regions
Analyze Counterexample

Example:

1. do
2.  look();
3.  old = new; q = q->next;
4.  if (q != NULL) { return; }
5.  q = q->next;
6.  unlock ();
7.  q pointers
8.  old = new
9.  unlock();
10.  unlock();

Reachability Tree

Predicates: lock

Analyze Counterexample

Example:

1. do
2.  look();
3.  old = new; q = q->next;
4.  if (q != NULL) { return; }
5.  q = q->next;
6.  unlock ();
7.  q pointers
8.  old = new
9.  unlock();
10.  unlock();

Reachability Tree

Predicates: lock

Repeat Build-and-Search

Example:

1. do
2.  look();
3.  old = new; q = q->next;
4.  if (q != NULL) { return; }
5.  q = q->next;
6.  unlock ();
7.  q pointers
8.  old = new
9.  unlock();
10.  unlock();

Reachability Tree

Predicates: lock, new=old

Repeat Build-and-Search

Example:

1. do
2.  look();
3.  old = new; q = q->next;
4.  if (q != NULL) { return; }
5.  q = q->next;
6.  unlock ();
7.  q pointers
8.  old = new
9.  unlock();
10.  unlock();

Reachability Tree

Predicates: lock, new=old

Repeat Build-and-Search

Example:

1. do
2.  look();
3.  old = new; q = q->next;
4.  if (q != NULL) { return; }
5.  q = q->next;
6.  unlock ();
7.  q pointers
8.  old = new
9.  unlock();
10.  unlock();

Reachability Tree

Predicates: lock, new=old

Inconsistent

new == old
Scaling Sequential Verification

C Program → Abstract → Safe
Property → Refine → Trace

Problem: Abstraction is Expensive
Solution: 1. Abstract reachable states, 2. Avoid refining error free regions
Key Idea: Reachability Tree

Repeat Build-and-Search

Example:

```c
while(!done) {
    lock();
    q = q->next;
    unlock();
}
```

Reachability Tree

Predicates: LOCK, new==old

Results

<table>
<thead>
<tr>
<th>Program</th>
<th>Lines*</th>
<th>Time(mins)</th>
<th>Predicates</th>
</tr>
</thead>
<tbody>
<tr>
<td>kbfiltx</td>
<td>12k</td>
<td>1</td>
<td>34</td>
</tr>
<tr>
<td>floppy</td>
<td>17k</td>
<td>7</td>
<td>93</td>
</tr>
<tr>
<td>diskperf</td>
<td>14k</td>
<td>5</td>
<td>78</td>
</tr>
<tr>
<td>osaudio</td>
<td>18k</td>
<td>20</td>
<td>85</td>
</tr>
</tbody>
</table>

Property:
IRP Handler
Win/NT/DOS

#Predicates grows with program size

While(true) {
    if (p) lock();
    if (p) unlock();
}

Tracking lock not enough

Problem:
p1,...,pn needed for verification
Exponential reachable abstract states
Predicates useful locally

```
while(1){
    :LOCK
    1: if (p) lock();
    unlock();
    :LOCK
    2: if (p) lock();
    if (p) unlock();
    :LOCK
    3: m: if (p) lock();
    if (p) unlock();
    ...
}
```

2m Abstract States

Solution: Use predicates only where needed
Using Counterexamples:
Q1. Find predicates
Q2. Find where predicates are needed

Scaling Sequential Verification

Counterexample Traces

```
1: x = ctr;
2: y = ctr;
4: if (x = i-1) { y = i; }
5: if (y = i) { ERROR: }
```
**Building Predicate Maps**

<table>
<thead>
<tr>
<th>Trace</th>
<th>Trace Formula</th>
<th>Predicate Map</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: x = cte</td>
<td></td>
<td>$x_i = ctr_i$</td>
</tr>
<tr>
<td>2: cte = cte + 1</td>
<td></td>
<td>$E x_i = ctr_i + 1$</td>
</tr>
<tr>
<td>3: y = cte</td>
<td></td>
<td>$E y_i = ctr_i$</td>
</tr>
<tr>
<td>4: assume($x$ = $y$ - 1)</td>
<td></td>
<td>$E x_i = y_i -1$</td>
</tr>
<tr>
<td>5: assume($y$ ≠ 1)</td>
<td></td>
<td>$E y_i ≠ y_0$</td>
</tr>
</tbody>
</table>

- Cut + Interpolate at each point
- Pred. Map: $pc_i$, Interpolant from cut $i$

**Local Predicate Use**

Use predicates needed at location

- #Preds grows with program size
- #Preds per location small

**Results**

<table>
<thead>
<tr>
<th>Program</th>
<th>Lines</th>
<th>Previous Time(mins)</th>
<th>Time (mins)</th>
<th>Predicates Total Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>kbdflts</td>
<td>12k</td>
<td>1</td>
<td>3</td>
<td>72</td>
</tr>
<tr>
<td>floppy</td>
<td>17k</td>
<td>7</td>
<td>25</td>
<td>240</td>
</tr>
<tr>
<td>diskpft</td>
<td>14k</td>
<td>5</td>
<td>13</td>
<td>140</td>
</tr>
<tr>
<td>uaudio</td>
<td>18k</td>
<td>20</td>
<td>23</td>
<td>256</td>
</tr>
<tr>
<td>parport</td>
<td>61k</td>
<td>DNFP</td>
<td>74</td>
<td>753</td>
</tr>
<tr>
<td>parclass</td>
<td>138k</td>
<td>DNFP</td>
<td>77</td>
<td>382</td>
</tr>
</tbody>
</table>

- Pre-processed

**Summary**

- Model checking is most expressive form of verification currently in (fairly) wide use
  - And the most expensive
- Explores all paths
  - Exponential for this reason
- Algorithms, optimizations to avoid exponential cost
  - Refine only on demand
  - Refine different parts of the search space differently