The Software Model Checker
BLAST

http://mtc.epfl.ch/software-tools/blast/

BLAST 2.0 Team:
Dirk Beyer, Tom Henzinger,
Ranjit Jhala, and Rupak Majumdar

Guest Lecture in Viktor Kuncak’s Verification Class, 2008-05-08
Motivation

Software stands for
- Functionality
- Flexibility
- Affordability in today’s products and infrastructures.

Practice:
- Vulnerability
- Obstacle to redesign
- Cost overruns
- Buggy, brittle, insecure, and not interoperable.
French Guyana, June 4, 1996
$600 million software failure
Mars, December 3, 1999
Crashed due to uninitialized variable
Mars, July 4, 1997
Lost contact due to priority inversion bug
Something reliable

Uptime: 68 years
An exception 06 has occurred at 0028:C11B3ADC in VxD DiskTSD(03) + 00001660. This was called from 0028:C11B40C8 in VxD voltrack(04) + 00000000. It may be possible to continue normally.

* Press any key to attempt to continue.
* Press CTRL+ALT+RESET to restart your computer. You will lose any unsaved information in all applications.

Press any key to continue
Our Application Areas

• Verification of systems code
  - Locking disciplines
  - Interface specifications

• Temporal properties
  - Require path-sensitive analysis
  - Swamped by false positives

• Really hard to check
Specifying and Checking Properties of Programs

• Goals
  - Defect detection
  - Partial validation

• Properties
  - Memory safety
  - Temporal safety
  - Security
  - ...

■ Many (mature) techniques
  - Automated deduction
  - Program analysis
  - Type checking
  - Model checking

■ Many projects
  Bandera, Blast, ESC-Java, FeaVer, JPF, LClint, OSQ, PolyScope, PREfix, SLAM, TVLA, Verisoft, xgcc, …
Property Checking

- Programmer gives partial specifications
- Code checked for consistency with spec
- Different from program correctness
  - Specifications are not complete
  - Are there actually complete specs?
  - Look for problems that occur often
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
Does a given usage rule hold?

- Undecidable!
  - Equivalent to the halting problem

- Restricted computable versions are prohibitively expensive (PSPACE)

- Why bother?
  - Just because a problem is undecidable, it doesn’t go away!
Running Example

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

Example () {
  do {
    lock();
    old = new;
    q = q->next;
  } while (q != NULL);
  if (q != NULL) {
    q->data = new;
    unlock();
    new ++;
  }
  while (new != old);
  unlock();
  return;
}
What a program \textit{really} is...

\begin{example}[h]
\begin{verbatim}
1: do{
    lock();
    old = new;
    q = q->next;
2:    if (q != NULL){
3:        q->data = new;
        unlock();
        new ++;
    }
4: } while(new != old);
5: unlock (old);
6: return;
\end{verbatim}
\end{example}
The Safety Verification Problem

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

Problem: Infinite state graph

Solution: Set of states $\models$ logical formula
## Representing States as Formulas

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

i.e. \(F_1 \land \neg F_2\) unsatisfiable
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**
Abstract States and Transitions

State

3: unlock();
new++;

4: }

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

State

3: unlock();
new++;

4: }

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

Theorem Prover

lock
old=new
¬lock
¬old=new
Abstraction

State

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

pc = 3
lock = false
old = 5
new = 5
q = 0x133a

pc = 4
lock = false
old = 5
new = 6
q = 0x133a

Existential Lifting

Theorem Prover

lock
¬
old=new

¬lock
¬
old=new
Abstraction

State

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

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

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

lock
¬ lock
old=new
¬ 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
   Till real counterexample or system proved safe

[Kurshan et al 93] [Clarke et al 00] [Ball-Rajamani 01]
Software Model Checking

BLAST

C Program → Yes → Safe

Property → No → Trace
Lazy Abstraction

C Program → spec.opt → Instrumented C file With ERROR label → BLAST

Property

Yes → Safe

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
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: 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.

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

Error Free
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
Example () {
  do{
    lock();
    old = new;
    q = q->next;
  } while (new != old);
  unlock();
}

Predicates: LOCK

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

Reachability Tree
Build-and-Search

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

Predicates:  LOCK

Reachability Tree
Build-and-Search

Example () {
  do{
    lock();
    old = new;
    q = q->next;
  } while (new != old);
  unlock();
}

Predicates: LOCK

Reachability Tree
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: \textit{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();
}

Reachability Tree
Repeat Build-and-Search

Example ( ) {
1:   do{
2:       lock();
3:       old = new;
4:       q = q->next;
5:       if (q != NULL){
6:           q->data = new;
7:           unlock();
8:           new ++;
9:       }
10:   }while(new != old);
11: }
12: unlock ();

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

Reachability Tree
Repeat 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:   }
  7: while(new != old);
  8: }
  9: unlock ();
}

Predicates: \( \text{LOCK, new==old} \)
Repeat Build-and-Search

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

Predicates:  \begin{align*}
& \text{LOCK, new==old} \\
& \neg \text{LOCK, \neg new = old} \\
& \neg \text{LOCK, new==old} \\
& [\text{new==old}] 
\end{align*}

Reachability Tree
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:  \textit{LOCK}, \textit{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

SAFE

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

Key Idea: Reachability Tree

Problem: Abstraction is Expensive

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

Q. How to find predicates?
# Predicates grows with program size

Problem:

\[ p_1, \ldots, p_n \text{ needed for verification} \]

Exponential reachable abstract states

While the code snippet above shows a program with multiple conditions, it highlights the issue of tracking \textit{lock} not being enough. The program checks multiple conditions \( p_1, \ldots, p_n \) within loops, and tracking each \textit{lock} and \textit{unlock} operation individually is insufficient for verifying the program's correctness due to the exponential growth in reachable abstract states.
# Predicates grows with program size


code

\begin{verbatim}
while(1) {
    1: if (p_1) lock();
        if (\neg p_1) unlock();
        ...
    2: if (p_2) lock();
        if (\neg p_2) unlock();
        ...
    n: if (p_n) lock();
        if (\neg p_n) unlock();
}
\end{verbatim}

\[ 2^n \text{ Abstract States} \]

\textbf{Problem:}

\[ p_1, \ldots, p_n \text{ needed for verification} \]

Exponential reachable abstract states
Predicates useful *locally*

```plaintext
while(1) {
    1: if (p1) lock();
        if (p1) unlock();
    ...
    2: if (p2) lock();
        if (p2) unlock();
    ...
    n: if (pn) lock();
        if (pn) unlock();
}
```

**Solution:** Use predicates only where needed

Using **Counterexamples:**

Q1. Find predicates

Q2. Find where predicates are needed
Lazy Abstraction

Problem: #Preds grows w/ Program Size
Solution: Localize pred. use, find where preds. needed
Counterexample Traces

1: x = ctr;
2: ctr = ctr + 1;
3: y = ctr;
4: if (x = i-1) {
   5:   if (y != i) {
        ERROR: }
   }

1: x = ctr
2: ctr = ctr + 1
3: y = ctr
4: assume(x = i-1)
5: assume(y ≠ i)
**Trace Formulas**

<table>
<thead>
<tr>
<th>Trace</th>
<th>SSA Trace</th>
<th>Trace Feasibility Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: ( x = \text{ctr} )</td>
<td>1: ( x_1 = \text{ctr}_0 )</td>
<td>( x_1 = \text{ctr}_0 )</td>
</tr>
<tr>
<td>2: ( \text{ctr} = \text{ctr} + 1 )</td>
<td>2: ( \text{ctr}_1 = \text{ctr}_0 + 1 )</td>
<td>( \text{ctr}_1 = \text{ctr}_0 + 1 )</td>
</tr>
<tr>
<td>3: ( y = \text{ctr} )</td>
<td>3: ( y_1 = \text{ctr}_1 )</td>
<td>( y_1 = \text{ctr}_1 )</td>
</tr>
<tr>
<td>4: assume(( x = i - 1 ))</td>
<td>4: assume(( x_1 = i_0 - 1 ))</td>
<td>( x_1 = i_0 - 1 )</td>
</tr>
<tr>
<td>5: assume(( y \neq i ))</td>
<td>5: assume(( y_1 \neq i_0 ))</td>
<td>( y_1 \neq i_0 )</td>
</tr>
</tbody>
</table>

**Thm:** Trace is **feasible** \( \iff \) TF is **satisfiable**
The Present State...

Trace

1: \( x = \text{ctr} \)

2: \( \text{ctr} = \text{ctr} + 1 \)

3: \( y = \text{ctr} \)

4: assume(\( x = i-1 \))

5: assume(\( y \neq i \))

... is all the information the executing program has here

State...

1. ... after executing trace past (prefix)

2. ... knows present values of variables

3. ... makes trace future (suffix) infeasible

At \( pc_4 \), which predicate on present state shows infeasibility of future?
What Predicate is needed?

Trace

1: \( x = \text{ctr} \)
2: \( \text{ctr} = \text{ctr} + 1 \)
3: \( y = \text{ctr} \)
4: \( \text{assume}(x = i-1) \)
5: \( \text{assume}(y \neq i) \)

Trace Formula (TF)

\[
\begin{align*}
\text{x}_1 &= \text{ctr}_0 \\
\land \quad \text{ctr}_1 &= \text{ctr}_0 + 1 \\
\land \quad \text{y}_1 &= \text{ctr}_1 \\
\land \quad \text{x}_1 &= i_0 - 1 \\
\land \quad \text{y}_1 &\neq i_0
\end{align*}
\]
What Predicate is needed?

Trace

1: \( x = \text{ctr} \)
2: \( \text{ctr} = \text{ctr} + 1 \)
3: \( y = \text{ctr} \)
4: \( \text{assume}(x = i-1) \)
5: \( \text{assume}(y \neq i) \)

Trace Formula (TF)

\[
\begin{align*}
    x_1 &= \text{ctr}_0 \\
    \land &\quad \text{ctr}_1 = \text{ctr}_0 + 1 \\
    \land &\quad y_1 = \text{ctr}_1 \\
    \land &\quad x_1 = i_0 - 1 \\
    \land &\quad y_1 \neq i_0
\end{align*}
\]

Relevant Information

1. ... after executing trace prefix

Predicate ...

... implied by TF prefix
What Predicate is needed?

Trace
1: \( x = \text{ctr} \)
2: \( \text{ctr} = \text{ctr} + 1 \)
3: \( y = \text{ctr} \)
4: \( \text{assume}(x = i-1) \)
5: \( \text{assume}(y \neq i) \)

Trace Formula (TF)
\[
\begin{align*}
x_1 &= \text{ctr}_0 \\
\land \quad \text{ctr}_1 &= \text{ctr}_0 + 1 \\
\land \quad y_1 &= \text{ctr}_1 \\
\land \quad x_{x_1} &= i_0 - 1 \\
\land \quad y_1 &\neq i_0
\end{align*}
\]

Predicate ...
... implied by TF prefix
... on \text{common} variables

Relevant Information
1. ... after executing trace prefix
2. ... has \text{present values} of variables
What Predicate is needed?

Trace
1: \( x = \text{ctr} \)
2: \( \text{ctr} = \text{ctr} + 1 \)
3: \( y = \text{ctr} \)
4: \( \text{assume}(x = i-1) \)
5: \( \text{assume}(y \neq i) \)

Trace Formula (TF)
\[
\begin{align*}
x_1 &= \text{ctr}_0 \\
\land \quad \text{ctr}_1 &= \text{ctr}_0 + 1 \\
\land \quad y_1 &= \text{ctr}_1 \\
\land \quad x_1 &= i_0 - 1 \\
\land \quad y_1 &\neq i_0
\end{align*}
\]

Relevant Information
1. ... after executing trace prefix
2. ... has present values of variables
3. ... makes trace suffix infeasible

Predicate ...
... implied by TF prefix
... on common variables
... & TF suffix is unsatisfiable
What Predicate is needed?

Trace
1: \( x = ctr \)
2: \( ctr = ctr + 1 \)
3: \( y = ctr \)
4: assume\((x = i-1)\)
5: assume\((y \neq i)\)

Trace Formula (TF)
\[
x_1 = ctr_0 \\
\land ctr_1 = ctr_0 + 1 \\
\land y_1 = ctr_1 \\
\land x_1 = i_0 - 1 \\
\land y_1 \neq i_0
\]

Relevant Information
1. ... after executing trace prefix
2. ... has present values of variables
3. ... makes trace suffix infeasible

Predicate ...
... implied by TF prefix
... on common variables
... & TF suffix is unsatisfiable
Interpolant = Predicate !

Trace
1: x = ctr
2: ctr = ctr + 1
3: y = ctr
4: assume(x = i-1)
5: assume(y ≠ i)

Trace Formula
\[ x_1 = ctr_0 \]
\[ \land \quad ctr_1 = ctr_0 + 1 \]
\[ \land \quad y_1 = ctr_1 \]
\[ \land \quad x_1 = i_0 - 1 \]
\[ \land \quad y_1 ≠ i_0 \]

Craig Interpolant
[Craig 57]

Predicate at 4:
\[ y = x + 1 \]

Interpolate \( \Phi \)

\( \Psi^- \)
\[ y_1 = x_1 + 1 \]

\( \Phi \)

\( \Psi^+ \)

Computable from Proof of Unsat
[Krajicek 97] [Pudlak 97]

Predicate ...
... implied by TF prefix
... on common variables
... & TF suffix is unsatisfiable
Interpolant = Predicate!

<table>
<thead>
<tr>
<th>Trace</th>
<th>Trace Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: ( x = \text{ctr} )</td>
<td>( x_1 = \text{ctr}_0 )</td>
</tr>
<tr>
<td>2: ( \text{ctr} = \text{ctr} + 1 )</td>
<td>( \text{ctr}_1 = \text{ctr}_0 + 1 )</td>
</tr>
<tr>
<td>3: ( y = \text{ctr} )</td>
<td></td>
</tr>
<tr>
<td>4: ( \text{assume}(x = i-1) )</td>
<td>( \text{assume}(x_i = i_0 - 1) )</td>
</tr>
<tr>
<td>5: ( \text{assume}(y \neq i) )</td>
<td>( y_i \neq i_0 )</td>
</tr>
</tbody>
</table>

**Q. How to compute interpolants?**

Craig Interpolant
- [Craig 57]
- Computable from Proof of Unsat
- [Krajíček 97] [Pudlák 97]

Predicate...
- ... implied by TF prefix
- ... on common variables
- ... & TF suffix is unsatisfiable
### Building Predicate Maps

<table>
<thead>
<tr>
<th>Trace</th>
<th>Trace Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: ( x = \text{ctr} )</td>
<td>( x_1 = \text{ctr}_0 )</td>
</tr>
<tr>
<td>2: ( \text{ctr} = \text{ctr} + 1 )</td>
<td>( \land \text{ctr}_1 = \text{ctr}_0 + 1 )</td>
</tr>
<tr>
<td>3: ( y = \text{ctr} )</td>
<td>( \land y_1 = \text{ctr}_1 )</td>
</tr>
<tr>
<td>4: assume(( x = i-1 ))</td>
<td>( \land x_1 = i_0 - 1 )</td>
</tr>
<tr>
<td>5: assume(( y \neq i ))</td>
<td>( \land y_1 \neq i_0 )</td>
</tr>
</tbody>
</table>

- Cut + Interpolate at each point
- Pred. Map: \( \text{pc}_i \mapsto \text{Interpolant from cut } i \)
Building Predicate Maps

Trace

1: x = ctr
2: ctr = ctr + 1
3: y = ctr
4: assume(x = i-1)
5: assume(y ≠ i)

Trace Formula

x_1 = ctr_0
\land ctr_1 = ctr_0 + 1
\land y_1 = ctr_1
\land x_1 = i_0 - 1
\land y_1 ≠ i_0

• Cut + Interpolate at each point
• Pred. Map: \( pc_i \mapsto \text{Interpolant from cut } i \)
# Building Predicate Maps

## Trace

<table>
<thead>
<tr>
<th>Trace</th>
<th>Trace Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: ( x = \text{ctr} )</td>
<td>( x_1 = \text{ctr}_0 )</td>
</tr>
<tr>
<td>2: ( \text{ctr} = \text{ctr} + 1 )</td>
<td>( \land \text{ctr}_1 = \text{ctr}_0 + 1 )</td>
</tr>
<tr>
<td>3: ( y = \text{ctr} )</td>
<td>( \land y_1 = \text{ctr}_1 )</td>
</tr>
<tr>
<td>4: ( \text{assume}(x = i-1) )</td>
<td>( \land x_1 = i_0 - 1 )</td>
</tr>
<tr>
<td>5: ( \text{assume}(y \neq i) )</td>
<td>( \land y_1 \neq i_0 )</td>
</tr>
</tbody>
</table>

- Cut + Interpolate at each point
- Pred. Map: \( \text{pc}_i \mapsto \text{Interpolant from cut } i \)
Building Predicate Maps

Trace | Trace Formula
---|---
1: x = ctr | x₁ = ctr₀
2: ctr = ctr + 1 | \( x₁ = ctr₀ + 1 \)
3: y = ctr | \( y₁ = ctr₁ \)
4: assume(x = i-1) | \( x₁ = i₀ - 1 \)
5: assume(y ≠ i) | \( y₁ ≠ i₀ \)

- Cut + Interpolate at each point
- Pred. Map: \( pcᵢ \) \( \mapsto \) Interpolant from cut \( i \)

Predicate Map
2: \( x = ctr \)
3: \( x = ctr - 1 \)
4: \( y = x + 1 \)
5: \( y = i \)
Local Predicate Use

Use predicates **needed** at location

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

---

**Predicate Map**
2: \( x = ctr \)
3: \( x = ctr - 1 \)
4: \( y = x + 1 \)
5: \( y = i \)

---

**Local Predicate use**
Ex: \( 2n \) states

**Global Predicate use**
Ex: \( 2^n \) states
### Localizing

Property3:
IRP Handler
Win NT DDK

<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
Lazy Abstraction

Problem: \#Preds grows w/ Program Size
Solution: Localize pred. use, find where preds. needed
Lazy Abstraction: Summary

- **Predicates:**
  - Abstract infinite program states

- **Counterexample-guided Refinement:**
  - Find predicates tailored to prog, property

1. **Abstraction**: Expensive
   Reachability Tree

2. **Refinement**: Find predicates, use locations
   Proof of unsat of TF + Interpolation
The BLAST Query Language

1. (Possibly Infinite-State) **Monitor Automata** for Reachability Queries over Program Locations

2. First-Order Imperative **Scripting Language** for Combining Relations over Program Locations
GLOBAL int locked;

EVENT {
    PATTERN { init() }
    ACTION { locked = 0; }
}

EVENT {
    PATTERN { lock() }
    ASSERT { locked == 0 }
    ACTION { locked = 1; }
}

EVENT {
    PATTERN { unlock() }
    ASSERT { locked == 1 }
    ACTION { locked = 0; }
}
Two-State Locking Monitor

GLOBAL int locked;

EVENT {
    PATTERN { init() }
    ACTION { locked = 0; }
}

EVENT {
    PATTERN { lock() }
    ASSERT { locked == 0 }
    ACTION { locked = 1; }
}

EVENT {
    PATTERN { unlock() }
    ASSERT { locked == 1 }
    ACTION { locked = 0; }
}

U

L

E

E

else REJECT

else REJECT
source(11) := LOC_LABEL(11,"START");

target(12) := TRUE(12);

error-traces(11,12) := REJECT(source,target,monitor);

error-locs(12) := EXISTS(11,error-traces(11,12));

PRINT "The following locations are reachable and cause a locking error:"

PRINT error-locs(12);

REJECT(11,12,monitor) is the set of all location pairs (11,12) such that there is a feasible program trace from 11 to 12 which is rejected by the automaton monitor.
Type-State Locking Monitor

SHADOW lock_t { int locked; }

EVENT {
    PATTERN { init($1) }
    ACTION { $1->locked = 0; }
}

EVENT {
    PATTERN { lock($1) }
    ASSERT { $1->locked == 0 }
    ACTION { $1->locked = 1; }
}

EVENT {
    PATTERN { unlock($1) }
    ASSERT { $1->locked == 1 }
    ACTION { $1->locked = 0; }
}
Dead-Code Analysis

source\((11)\) := LOC_LABEL\((11, "START")\);

target\((12)\) := TRUE\((12)\);

feasible-traces\((11, 12)\) := ACCEPT\((source, target, EMPTY)\);

reached-locs\((12)\) := feasible-traces\((_, 12)\);

PRINT "The following locations are not reachable:"

PRINT !reached-locs\((12)\);
GLOBAL int defined;

INITIAL { defined = 0; }

EVENT {
   PATTERN { j = $1; }
   ACTION { defined ++ ; }
}

FINAL { defined == 1 }

affected(l1,l2) :=
ACCEPT(LOC_LHS(l1,"j"),LOC_RHS(l2,"j"),monitor);
PRINT affected(l1,l2);

else REJECT
Benefits of Two-Level Specifications

1. Separates properties from programs, while keeping a familiar syntax for writing properties

2. Treats a program as a database of facts that can be queried, and supports macros for traditional temporal-logic specifications

3. Supports the formulation of decomposition strategies for verification tasks

4. Supports the incremental maintenance of properties during program development
The BLAST Two-Level Query Language

1. (Possibly Infinite-State) **Monitor Automata** for Reachability Queries over Program Locations:
   checked by the BLAST model checking engine

2. First-Order Imperative **Scripting Language** for Combining Relations over Program Locations:
   checked by the CrocoPat relational query engine
   [Beyer, Noack, Lewerentz]