Model Checking, Theorem Proving, and Abstract Interpretation: The Convergence of Formal Verification Technologies

Tom Henzinger
EPFL
Three Verification Communities

• Model checking:
  - automatic, but inefficient (does not scale)
  - successful in hardware verification

• Theorem proving:
  - precise, but interactive (requires user intervention)
  - much recent progress in decision procedures

• Static analysis:
  - efficient, but imprecise (many false positives)
  - standard in compilers
Convergence

• Not just interaction of model checkers, theorem provers, and static analyzers via a shared code database

• But deep integration of the three technologies within a single tool

• Is it a model checker exploring states? A theorem prover manipulating formulas? A static analyzer interpreting an abstract program?

All of the above!
Property Checking

• Programmer gives **partial** specifications
  - Is there a complete spec for Word? Emacs?
  - Often implicit: code annotations, memory safety, race freedom

• Code checked for consistency with specification

• Different from program correctness
  - Partial specs of large programs rather than complete specs of small programs
Interface Usage Rules

- Rules in documentation
  - Order of operations and data access
  - Incomplete, unenforced, wordy

- Rule violations lead to bad behavior
  - System crash or deadlock
  - Unexpected exceptions
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: IRP Handler

[Diagram of IRP Handler process with various states and transitions such as MPR3, MPR2, MPR1, NP, SKIP1, SKIP2, IPC, PPC, DC, N/A, CallDriver, and transitions like synch, not pending returned, prop completion, no prop completion, Mark Pending, return Pending, return child status, return not Pending, and Win NT DDK by Fahndrich]
Locking Example

Example () {
  do {
    lock();
    old = new;
    q = q->next;
    if (q != NULL) {
      q->data = new;
      unlock();
      new ++;
    }
  } while (new != old);
  unlock();
  return;
}
The Model Checking View:
Programs are State Transition Systems

[Clarke-Emerson, Queille-Sifakis]
The Safety Verification Problem

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

Theorem proving: guess and verify inductive invariant
The Safety Verification Problem

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

Problem: infinite state graph
Solution: set of states $\sim$ logical formula

Model checking: explore all reachable states
Predicate Abstraction

- **Predicates** on program state:
  - \textit{lock}
  - \textit{old} = \textit{new}

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

- Number of abstract states is **finite**

[Cousot-Cousot, Graf-Saidi]
Abstract States and Transitions

[Cousot-Cousot, Graf-Saidi]

\[
\begin{align*}
    pc & \rightarrow 3 \\
    lock & \rightarrow 0x133a \\
    old & \rightarrow 5 \\
    new & \rightarrow 5 \\
    q & \rightarrow 0x133a
\end{align*}
\]

3: \text{unlock()};
new++;  
4: 

\[
\begin{align*}
    pc & \rightarrow 4 \\
    lock & \rightarrow 0x133a \\
    old & \rightarrow 5 \\
    new & \rightarrow 6 \\
    q & \rightarrow 0x133a
\end{align*}
\]

\begin{align*}
\text{Theorem Prover} \\
\text{lock} & \rightarrow old=\text{new} \\
\neg\text{lock} & \rightarrow \neg old=\text{new}
\end{align*}

[Cousot-Cousot, Graf-Saidi]
Abstraction

Existential Lifting

3: unlock();
new++;

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

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

Theorem Prover

lock
old=new
¬lock
¬old=new
Analyze Abstraction

- Analyze finite graph

- **Overapproximation:**
  - safe $\Rightarrow$ program safe
  - no false negatives

- **Problem:** spurious counterexamples
  - false positives!
Counterexample-guided Refinement

Solution:
Use spurious counterexamples to refine abstraction

1. Add predicates to distinguish states across cut
Imprecision due to merge
Iterative Abstraction Refinement

1. Add predicates to distinguish states across cut
2. Build refined abstraction eliminates counterexample
3. Repeat search until real counterexample found or program proved safe

Solution:
Use spurious counterexamples to refine abstraction

[Kurshan et al., Clarke et al.]
[Ball-Rajamani 01: SLAM]
How to Compute Successors?

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

For each predicate $p$
check if $p$ is true (or false) after $OP$

Q: When is $p$ true after $OP$?
- if $WP( p, OP )$ is true before $OP$
- we know $F$ is true before $OP$
- Thm Prover query: $F \Rightarrow WP( p, OP )$?

Predicates: $LOCK, new=old$
How to Compute Successors?

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();
}

For each predicate $p$
check if $p$ is true (or false) after $OP$

$Q$: When is $p$ true after $OP$ ?
- if $WP(p, OP)$ is true before $OP$
- we know $F$ is true before $OP$
- Thm Prover query: $F \Rightarrow WP(p, OP)$

Predicate: new=old

True ? \hspace{1cm} (LOCK, new=old) $\Rightarrow$ (new + 1 = old) \hspace{1cm} NO
False ? \hspace{1cm} (LOCK, new=old) $\Rightarrow$ (new + 1 $\neq$ old) \hspace{1cm} YES
Abstraction is Expensive

Problem:
- \#abstract states = 2^\#predicates
- exponential Thm Prover queries

Observe:
- often only fraction of state space reachable
Abstract Only Reachable States

Problem:
- \#abstract states = 2^\#predicates
- exponential Thm Prover queries

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

Problem:
- #abstract states = 2^#predicates
- exponential Thm Prover queries

Solution:
Refine only spurious counterexamples
Putting It Together: Lazy Abstraction

Unroll abstraction
1. Pick tree node (= abs. state)
2. Add children (= abs. successors)
3. On revisiting abs. state, cut off

Find min. spurious suffix
- learn new predicates
- rebuild subtree with new predicates

Abstract Reachability Tree
[Henzinger-Jhala-Majumdar-Sutre 03: BLAST]
Putting It Together: Lazy Abstraction

Unroll abstraction
1. Pick tree node (= abs. state)
2. Add children (= abs. successors)
3. On revisiting abs. state, cut off

Find min. spurious suffix
- learn new predicates
- rebuild subtree with new predicates

Error Free
Putting It Together: Lazy Abstraction

Unroll abstraction
1. Pick tree node (= abs. state)
2. Add children (= abs. successors)
3. On revisiting abs. state, cut off

Find min. spurious suffix
- learn new predicates
- rebuild subtree with new predicates

Error Free

SAFE

Only abstract reachable states
Don’t refine error-free regions
Abstract Interpretation

Example

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

Predicates: \( \neg \text{LOCK} \)
Abstract Interpretation

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

Abstract Reachability Tree
Abstract Interpretation

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

Abstract Reachability Tree
Abstract Interpretation

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

Abstract Reachability Tree
Abstract Interpretation

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: \( \text{LOCK} \)

Abstract Reachability Tree
Abstract Interpretation

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

Predicates: LOCK

Abstract 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

Abstract 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}

Abstract Reachability Tree

Spurious \[\text{new} = \text{old}\]

\[\text{old} = \text{new}\]
Refined Abstract Interpretation

Example ( ) {
1: do {
    lock();
    old = new;
    q = q->next;
2:    if (q != NULL) {
3:        q->data = new;
un3:    lock();
    new ++;
        }
4:    }while(new != old);
5: unlock();
}

Predicates:  $\neg LOCK$, new=old
Refined Abstract Interpretation

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

Example ( ) {
    do{
        lock();
        old = new;
        \( q = q->\text{next}; \)
    } while( new != old);
    unlock();
}
Refined Abstract Interpretation

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

Abstract Reachability Tree
Refined Abstract Interpretation

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: \(\text{LOCK}, \text{new}=\text{old}\)
Refined Abstract Interpretation

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

Predicates:  LOCK, new=old

Abstract Reachability Tree
Refined Abstract Interpretation

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

Abstract Reachability Tree

SAFE
# Predicates Grows with Program Size

Problem:

\( p_1, \ldots, p_n \) needed for verification

# Reachable abstract states exponential

Tracking \textit{lock} not enough
# Predicates Grows with Program Size

Problem: \( p_1, \ldots, p_n \) needed for verification

# Reachable abstract states exponential
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();
}
```

2n abstract states

**Solution:** Use predicates *only* where needed

Using *counterexamples*:

**Q1.** Find *predicates*

**Q2.** Find *where* predicates are needed
Counterexample Trace

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)
Build Trace Formula

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

1: \( x_1 = \text{ctr}_0 \)
2: \( \text{ctr}_1 = \text{ctr}_0 + 1 \)
3: \( y_1 = \text{ctr}_1 \)
4: assume(\( x_1 = i_0 - 1 \))
5: assume(\( y_1 \neq i_0 \))

Trace
SSA Trace
Trace Formula

Trace is feasible \( \iff \) TF is satisfiable
Which Predicate is Needed?

<table>
<thead>
<tr>
<th>Trace</th>
<th>Trace Formula (TF)</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>
Which 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)
\[
\begin{align*}
\text{x}_1 &= \text{ctr}_0 \\
\land \quad \text{ctr}_1 &= \text{ctr}_0 + 1 \\
\land \quad \text{y}_1 &= \text{ctr}_1 \\
\land \quad \text{x}_1 &= i_0 - 1 \\
\land \quad \text{y}_1 &\neq i_0
\end{align*}
\]

Predicate ...
... implied by TF prefix

Relevant Information
1. ... after executing trace prefix
Which Predicate is Needed?

Trace

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

Trace Formula (TF)

\[ x_1 = \text{ctr}_0 \]
\[ \land \quad \text{ctr}_1 = \text{ctr}_0 + 1 \]
\[ \land \quad y_1 = \text{ctr}_1 \]
\[ \land \quad x_1 = i_0 - 1 \]
\[ \land \quad y_1 \neq i_0 \]

Relevant Information

1. ... after executing trace prefix
2. ... has present values of variables

Predicate ...

... implied by TF prefix
... on common variables
Which 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)
\[
\begin{align*}
  x_1 &= c_{r0} \\
  \land & c_{r1} = c_{r0} + 1 \\
\end{align*}
\]
\[
\begin{align*}
  y_1 &= c_{r1} \\
  \land & x_1 = i_0 - 1 \\
\end{align*}
\]
\[
\begin{align*}
  \land & y_1 \neq i_0 \\
\end{align*}
\]

Predicate ...
... implied by TF prefix
... on common variables
... and TF suffix is unsatisfiable

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

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

\[ \begin{align*}
\psi^- & : x_1 = \text{ctr}_0 \\
\psi^+ & : y_1 = x_1 + 1
\end{align*} \]

Interpolate \( \Phi \)

Craig Interpolant

Predicate at 4:

\[ y = x + 1 \]

Computable from proof of unsatisfiability

[Krajicek. Pudlak, McMillan]

... implied by TF prefix

... on common variables

... and TF suffix is unsatisfiable
Building Predicate Maps

Trace | Trace Formula

1: \( x = \text{ctr} \) | \( x_1 = \text{ctr}_0 \)
2: \( \text{ctr} = \text{ctr} + 1 \) | \( \land \text{ctr}_1 = \text{ctr}_0 + 1 \)
3: \( y = \text{ctr} \) | \( \land y_1 = \text{ctr}_1 \)
4: assume\( (x = i-1) \) | \( \land x_1 = i_0 - 1 \)
5: assume\( (y \neq i) \) | \( \land y_1 \neq i_0 \)

- Cut + interpolate at each point
- Predicate Map: \( \text{pc}_i \rightarrow \) interpolant from cut i
Building Predicate Maps

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

\( x_1 = \text{ctr}_0 \)
\( \land \quad \text{ctr}_1 = \text{ctr}_0 + 1 \)
\( \land \quad y_1 = \text{ctr}_1 \)
\( \land \quad x_1 = i_0 - 1 \)
\( \land \quad y_1 \neq i_0 \)

Predicate Map

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

\( \psi^- \)
Interpolate
\( x_1 = \text{ctr}_1 - 1 \)

\( \psi^+ \)

- Cut + interpolate at each point
- Predicate Map: \( \text{pc}_i \rightarrow \) interpolant from cut i
## Building Predicate Maps

<table>
<thead>
<tr>
<th>Trace</th>
<th>Trace Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: $x = ctr$</td>
<td>$x_1 = ctr_0$</td>
</tr>
<tr>
<td>2: $ctr = ctr + 1$</td>
<td>$\land ; ctr_1 = ctr_0 + 1$</td>
</tr>
<tr>
<td>3: $y = ctr$</td>
<td>$\land ; y_1 = ctr_1$</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
- Predicate Map: $pc_i \rightarrow$ interpolant from cut $i$

---

**Predicate Map**
- 2: $x = ctr$
- 3: $x = ctr - 1$
- 4: $y = x + 1$
Building Predicate Maps

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

\[
\begin{align*}
1: & \quad x_1 = \text{ctr}_0 \\
2: & \quad \text{ctr}_1 = \text{ctr}_0 + 1 \\
3: & \quad y_1 = \text{ctr}_1 \\
4: & \quad x_1 = i_0 - 1 \\
5: & \quad y_1 \neq i_0
\end{align*}
\]

Predicate Map

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

• Cut + interpolate at each point
• Predicate Map: \( \text{pc}_i \rightarrow \text{interpolant from cut } i \)
Local Predicate Use

Use predicates **needed at location**

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

**Predicate Map**
2: $x = \text{ctr}$
3: $x = \text{ctr} - 1$
4: $y = x + 1$
5: $y = i$

Verification scales!

Local Predicate use
Ex: $O(n)$ states

Global Predicate use
Ex: $2^n$ states
<table>
<thead>
<tr>
<th>Program</th>
<th>Lines*</th>
<th>Non-lazy Time (mins)</th>
<th>Lazy 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

Property 3: IRP Handler Win NT DDK
Verification by Theorem Proving

1. Loop invariants
2. Logical formula
3. Check validity

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

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

1. Loop invariants
2. Logical formula
3. Check validity
- Loop invariants
- Multithreaded programs
  + Behaviors encoded in logic
  + Decision procedures

Precise [e.g. ESC]
Verification by Program Analysis

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

1. Dataflow facts
2. Constraint system
3. Solve constraints

- Imprecision due to fixed facts
+ Abstraction
+ Type/flow analyses

Scalable [e.g. CQUAL, ESP, MC]
Verification by Model Checking

1. (Finite State) Program
2. State Transition Graph
3. Reachability

- Finite state model
- State explosion
  + State exploration
  + Counterexamples

Automatic [e.g. SPIN, Bandera, JPF]
Combining Strengths

**Theorem Proving**
- Loop invariants
  + Behaviors encoded in logic
  + Theorem provers
Computing successors; refine

**Program Analysis**
- Imprecise
  + Abstraction
Shrink state space

**Model Checking**
- Finite-state model, state explosion
  + State space exploration
Path-sensitive analysis
  + Counterexamples
Finding relevant facts

**Lazy Abstraction**
Conclusions

- take the best of each technology: automatic, precise, scalable
- verify programs, not models: verifying compiler
- current research: concurrency, heap

http://mtc.epfl.ch/software-tools/blast
joint with R. Jhala (UCSD) and R. Majumdar (UCLA)