Software Verification and Abstraction

Rupak Majumdar
Spreading fast

The worm slowly spread until July 19, when the number of computers attacking networks skyrocketed. Now, the worm is hibernating, ready to re-infect Aug. 1.

Jul 19, 2001
Code Red:
Buffer overrun
Estimated cost $2.6 billion

Source: Chemical Abstracts Service
Software is Unreliable
The Premise

Reliability is important

- More computers are used in embedded, networked, safety critical applications
- Cost of failure or fixing bug is high

Building large, complex, yet reliable systems is hard

These lectures: What can we do about it?

- Survey research on automatic tools for ensuring a system is “correct”
Systems and Models

Model

Calculate

Abstract
Build Model

Predict
Analyze Model

System

Test

Mathematics

Bridges
Aircraft
Hardware
Software

Mathematics Bridges Aircraft Hardware Software
Mathematical Abstractions

1. Relevant facts*  Mass, Tensile Strength
2. Model          ODEs/PDEs
3. Analysis       Solve Equations

*Bridges Programs

Building Blocks  Mechanics  Logic

* w.r.t. property of interest
System Verification Problem

I ⊨ S

“Implementation” System model

“models” “implements” “refines Satisfaction relation

“Specification” System properties
Example

C code for device driver \iff \text{“Driver does not deadlock”}

\text{“Implementation” System model} \models \text{“Specification” System properties}

\text{“models” “implements” “refines Satisfaction relation}
Example

Microprocessor design $\models$ ISA

"Implementation" System model $\models$ "Specification" System properties

"models" "implements" "refines
Satisfaction relation
Example

Electronic Control Unit \( \models \) “Controller is stable”

“Implementation” System model

“Specification” System properties

“models” “implements” “refines Satisfaction relation
Example

Transaction memory \models \text{Strict serializability}

“Implementation” System model

“models” “implements” “refines
Satisfaction relation

“Specification” System properties
Lecture 1: Model Checking
Basic Concepts

Rupak Majumdar
Model checking, generally interpreted:

Automatic algorithmic techniques for system verification which operate on a system model (semantics)
Somewhat General View

Model checking, narrowly interpreted:
Decision procedures for checking if a given Kripke structure is a model for a given formula of a modal logic (CTL or LTL).

Our view includes
- Dataflow analysis in compilers
- Symbolic execution based methods

Our view excludes
- Language design for ensuring properties
- Proof calculi and interactive theorem proving
There are many different model checking algorithms, depending on

- The system model
- The specification formalism
Discrete Systems Theory

**Trajectory:** dynamic evolution of state sequence of states

**Model:** generates a set of trajectories transition graph

**Property:** assigns boolean values to trajectories temporal logic formula

**Algorithm:** compute values of the trajectories generated by a model

“red and green alternate”
Paradigmatic Example: Mutual Exclusion

loop
  out: x1 := 1; last := 1
  req: await x2 = 0 or last = 2
  in:  x1 := 0
end loop.

||

loop
  out: x2 := 1; last := 2
  req: await x1 = 0 or last = 1
  in:  x2 := 0
end loop.

P1 P2

Property: It is never the case that P1 and P2 are both at `in'
System Modeling

• Various factors influence choice of model
  - State based vs event based
  - Concurrency model

• While the choice of system model is important for ease of modeling in a given situation,

  the only thing that is important for model checking is that the system model can be translated into some form of state-transition graph.

• So: Will not focus much on syntactic constructs
Syntax: Finite State Programs

• Parallel composition of C programs, without function calls

• Each variable has a finite range

• We’ll write such programs as guarded commands
Semantics: State Transition Graph

- $Q$: set of states \{q_1, q_2, q_3\}
- $q_0$: initial state $q_1$
- $\rightarrow \subseteq Q \times Q$: transition relation $q_1 \rightarrow q_2$
- $A$: set of atomic observations \{a, b\}
- $[\cdot]: Q \rightarrow 2^A$: observation function $[q_1] = \{a\}$
Important Restriction

Until notified, restrict attention to finite-state transition systems

Q is finite
Example: Mutual Exclusion

```
loop  ||  loop
  out: x1 := 1; last := 1  ||  out: x2 := 1; last := 2
  req: await x2 = 0 or last = 2  ||  req: await x1 = 0 or last = 1
  in: x1 := 0  ||  in: x2 := 0
end loop.

P1  ||  P2
```
3 · 3 · 2 · 2 · 2 = 72 states
The translation from a system description to a state-transition graph usually involves an exponential blow-up !!!

e.g., $n$ boolean variables $\Rightarrow 2^n$ states
System Verification Problem

I \models S

“Implementation” System model

“models” “implements” “refines” Satisfaction relation

“Specification” System properties
System Properties

Some orthogonal dimensions in choosing specification formalisms

1. operational vs. declarative:
   - automata vs. logic

2. may vs. must:
   - branching vs. linear time

2. prohibiting bad vs. desiring good behavior:
   - safety vs. liveness

The three decisions are orthogonal, and they lead to substantially different model-checking problems
Safety vs Liveness

• **Safety**: Something “bad” will never happen
  - Program does not produce bad result
    “partial correctness”
  - *Example*: Mutual exclusion

• **Liveness**: Something “good” eventually happens
  - The program produces a result “termination”
  - *Example*: A process wanting to go to the critical section eventually gets in
Safety vs Liveness Contd.

• **Safety:** those properties whose violation always has a finite witness
  - “if something bad happens on an infinite run, then it happens already on some finite prefix” --- Can be checked on finite runs

• **Liveness:** those properties whose violation never has a finite witness
  - “no matter what happens along a finite run, something good could still happen later” --- Must be checked on infinite runs
Two Remarks

1. The vast majority of properties to be verified are safety

2. While nobody will ever observe the violation of a true liveness property, liveness is a useful abstraction that turns complicated safety into simple liveness

Accordingly, we focus on safety for most of the lectures
Safety Model Checking

• Requirement: The system should always stay within some safe region

• Input: A state transition graph
• Input: A set of good states “invariants”

• Output: “Safe” if all executions maintain the invariant, “Unsafe” otherwise (and a trace)
From Safety to Reachability

- Input: A state transition graph
- Input: A set of bad states

- Output: “Safe” if there is no run from an initial state to any bad state, “Unsafe” otherwise (and a trace)
Model Checking Algorithm

- **Graph Search**
  - Linear time in the size of the graph
  - Exponential time in the size of the program

![Diagram showing model checking algorithm with bad states highlighted]
Enumerative Model Checking

- Provide access to each state
- For each state, provide access to neighboring states

- Implement classical graph algorithms
  - Depth-first or breadth-first search
  - Starting from initial states and searching forward for bad states
  - Or starting from bad states and searching backward for initial states
State Space Explosion

- Biggest problem is state space explosion
  - N bits $\Rightarrow 2^N$ states
- Many heuristics
  - Search on-the-fly,
  - partial order and symmetry reduction
  - Do not store dead variables

- Many successful implementations
  - Spin, Murphi, Verisoft, ... [Protocol verification]
Symbolic Model Checking

- Idea: Work with sets of states, rather than individual states

Given: Transition graph $G$, target states $\sigma^T$

begin
- $\sigma^R$ = set of Initial states
- repeat forever
  - if $\sigma^R \cap \sigma^T \neq \emptyset$ then return “yes”
  - if $\text{Post}(\sigma^R) \subseteq \sigma^R$ then return “no”
  - $\sigma^R := \sigma^R \cup \text{Post}(\sigma^R)$
end

Here, $\text{Post}(\sigma) = \{s' \mid \exists s \in \sigma. s \rightarrow s'\}$
Encoding Sets through Formulas

- Idea: Represent sets of states symbolically, using constraints
  
  - E.g., $1 \leq x \leq 100$ represents the 100 states $x = 1, x = 2, ..., x = 100$
  
  - Represent both sets of initial states and transition relation implicitly
Representing **States as Formulas**

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

i.e. \(F_1 \land \neg F_2\) unsatisfiable
Symbolic Transition Graph

• A transition graph
  - A Formula $\text{Init}(x)$ representing initial states
  - A Formula $\text{TR}(x, x')$ representing the transition relation

• Example: C program

  $x := e$  \hspace{1cm} $\text{TR}(x, x')$: $\text{loc} = \text{pc} \land \text{loc'} = \text{pc'} \land x' = e \land \{ y' = y \mid y \neq x \}$

  $\text{Assume}(p)$  \hspace{1cm} $\text{TR}(x, x')$: $\text{loc} = \text{pc} \land \text{loc'} = \text{pc'} \land p$
Symbolic Transition Graph

- **Operations:**
  - \( \text{Post}(X) = \{ s' \mid \exists s \in X. \ s \rightarrow s' \} \)
    \[ = \exists s. \ X(s) \land TR(s,s') \]
  - \( \text{Pre}(X) = \{ s \mid \exists s' \in X. \ s \rightarrow s' \} \)
    \[ = \exists s'. \ TR(s,s') \land X(s') \]

- Can implement using formula manipulations
Symbolic Model Checking

Given: Transition graph $G$, target states $\sigma^T$

begin
  - $\sigma^R = \text{Formula representing set of Initial states}$
  - repeat forever
    if $\sigma^R \land \sigma^T$ is satisfiable then return “yes”
    if $\text{Post}(\sigma^R) \Rightarrow \sigma^R$ then return “no”
    $\sigma^R := \sigma^R \lor \text{Post}(\sigma^R)$
  end

Here, $\text{Post}(\sigma)(s’) = \exists s. \sigma(s) \land TR(s,s’)$

Can be implemented using decision procedures for the language of formulas
Finite State Systems

- Symbolic representation in propositional logic
- State described by $n$ bits $X$
- A region is a propositional formula with free variables in $X$

- Can implement symbolic operations using propositional formula manipulations
Example: Mutual Exclusion

Symbolic representation has variables
\[ pc_1, pc_2, x_1, x_2, \text{last} \]

Initial states:
\[ pc_1=\text{out} \land pc_2=\text{out} \land x_1=0 \land x_2=0 \]

Transition relation:
\[ pc_1=\text{out} \land x_1'=1 \land \text{last}'=1 \land pc_2'=pc_2 \land x_2'=x_2 \]
\[ \lor \ldots \]
Additional Desirable Properties

- All operations must be efficient in practice
- Should maintain compactness whenever possible
- Canonical representations
- Representing initial states and transition relation from the program description should be efficient
Binary Decision Diagrams

• Efficient representations of boolean functions [Bryant86]

• Share commonalities

• Ordered BDDs:
  - Fix a linear ordering of the variables in X
  - BDD = DAG, with nodes labeled with boolean variables
  - Each variable occurs 0 or 1 times along a path
  - Paths in the DAG encode assignments to variables

• Extremely successful in hardware verification
More on Safety Properties

• Not all safety properties can be written as invariants on the program state space

• For example, if correctness depends on the order of events
  - Locks can be acquired and released in alternation, it is an error to acquire/release a lock twice in succession without an intermediate release / acquire
Monitors

• Write the ordering of events as an automaton (called the monitor)

• Take the product of the system with the monitor
  - The monitor tracks the sequence of events
  - It goes to a special “bad” state if a bad sequence occurs

• Now we can express the property as an invariant: the monitor state is never bad
Symbolic Search

- Guaranteed to terminate for finite state systems

- And can be applied to infinite state systems as well
  - Although without guarantees of termination in general
  - Application to infinite state requires richer languages for formulas and associated decision procedures
What about Software?

- Can construct an infinite state transition system from a program

- States: The state of the program
  - (stack, heap, pc location)

- Transitions: $q \rightarrow q'$ iff in the operational semantics, there is a transition of the program from $q$ to $q'$

- Initial state: Initial state of the program
Termination

- Each operation can be computed

- But iterating Pre or Post operations may not terminate

- What do we do now?
Observation

- Often, we do not need the exact set of reachable states
  - We need a set of states that separates the reachable states from the bad states
One Possibility

- User gives an estimate (inductive invariant)
  A set of states $\text{Inv}$ such that
  - $\text{Init} \subseteq \text{Inv}$
  - $\text{Inv} \cap \text{bad} = \emptyset$
  - $\text{Post}(\text{Inv}) \subseteq \text{Inv}$

* Can show that this implies system is safe (How?)
* Given $\text{Inv}$, and decision procedures, this procedure is guaranteed to terminate

- This is the idea of classical loop invariants
  - Problem: In general, it can be hard to manually construct $\text{Inv}$
Before we proceed

• What is the sign of the following product:

- 12433454628 * 94329545771?
Idea

• One can “abstract” the behavior of the system, and yet reason about certain aspects of the program

• Abstraction:
  -ve * +ve = -ve
Model Checking Algorithm

- Graph Search
Abstract Interpretation

- The state transition graph is large/infinite
- Suppose we put a finite grid on top
Existential Abstraction

• Every time $s \rightarrow s'$, we put $[s] \rightarrow [s']$
• This allows more behaviors
Abstract Model Checking

• Search the abstract graph until fixpoint
  - Can be much smaller than original graph
  - Can be finite, when original is infinite
Simulation Relations

• A relation \( \preceq \subseteq Q \times Q \) is a simulation relation if \( s \preceq s' \) implies
  - Observation(s) = Observation(s')
  - For all \( t \) such that \( s \rightarrow t \)
    there exists \( t' \) such that \( s' \rightarrow t' \)
    and \( s' \preceq t' \)

Formally captures notion of “more behaviors”
Implies containment of reachable behaviors
Main Theorem

- $s \preceq \llbracket s \rrbracket$ is a simulation relation
- If an error is unreachable in $\text{Abs}(G)$ then it is unreachable in $G$

- Plan:
  1. Find a suitable grid to make the graph finite state
  2. Run the finite-state model checking algorithm on this abstract graph
  3. If abstract graph is safe, say “safe” and stop
What if the Abstract Graph says Unsafe?

• The error may or may not be reachable in the actual system
  - Stop and say “Don’t know”
What if the Abstract Graph says Unsafe?

- Or, put a finer grid on the state space
- And try again
  - The set of abstract reachable states is smaller
  - Where do these grids come from?
Grids: Predicate Abstraction

- Suppose we fix a set of facts about program variables
  - E.g., old = new, lock = 0, lock = 1

- Grid: Two states of the program are equivalent if they agree on the values of all predicates
  - N predicates = $2^N$ abstract states

- How do we compute the grid from the program?
Predicate Abstraction

Region Representation: formulas over predicates

\[ \neg P_1 \lor \neg P_3 \lor P_4 \lor \neg P_4 \]

\[ \neg P_1, \neg P_2 \]

\[ \neg P_1, P_2 \]

\[ P_1, P_2 \]

\[ P_1, \neg P_2 \]

Karnaugh Map

\[ P_1 : x = y \]

\[ P_2 : z = t + y \]

\[ P_3 : x \leq z + 1 \]

\[ P_4 : *u = x \]

Set of states

Abstract Set: \( P_1 P_2 P_4 \lor \neg P_1 P_2 P_3 P_4 \)
Predicate Abstraction

- Box: abstract variable valuation
- BoxCover(S): Set of boxes covering S
- Theorem prover used to compute BoxCover

\begin{align*}
P_1 &: x = y \\
P_2 &: z = t + y \\
P_3 &: x \leq z+1 \\
P_4 &: \ast u = x
\end{align*}
Post#, Pre

- **pre**\((S, \text{op})\) = \(\{ s \mid \exists s' \in S. s \rightarrow^{\text{op}} s'\}\) (Weakest Precondition)
- **post**\((S, \text{op})\) = \(\{ s \mid \exists s' \in S. s' \rightarrow^{\text{op}} s\}\) (Strongest Postcondition)

- Abstract Operators: post#
  
  \(\text{post}(S, \text{op}) \subseteq \text{post}#(S, \text{op})\)
Computing Post#

- For each predicate $p$, check if
  - $S \Rightarrow \text{Pre}(p, \text{op})$ then have a conjunct $p$
  - $S \Rightarrow \text{Pre}(\neg p, \text{op})$ then have a conjunct $\neg p$
  - Else have no conjunct corresponding to $p$

- Use a theorem prover for these queries
Example

- I have predicates
  - lock=0, new=old, lock=1
- My current region is lock = 0 \land new= old
- Consider the assignment new = new + 1

- What is abstract post?
Example

- \( WP(new:=new+1, \ lock=0) \) is \( lock=0 \)
- \( WP(new:=new+1, \ lock=1) \) is \( lock=1 \)
- \( WP(new:=new+1, \ new=old) \) is \( new+1=old \)

- \( lock=0 \land new=old \Rightarrow lock = 0 \quad YES \)
- \( lock=0 \land new=old \Rightarrow lock \neq 0 \quad NO \)
- \( lock=0 \land new=old \Rightarrow lock = 1 \quad NO \)
- \( lock=0 \land new=old \Rightarrow lock \neq 1 \quad YES \)
- \( lock=0 \land new=old \Rightarrow new+1=old \quad NO \)
- \( lock=0 \land new=old \Rightarrow new+1 \neq old \quad YES \)

- So post is \( lock = 0 \land lock \neq 1 \land new \neq old \)
Symbolic Search with Predicates

Symbolic representation:
- Boolean formulas of (fixed set of) predicates
  - Boolean operations: easy
  - Emptiness check: Decision procedures

- Post: The abstract post computation algorithm
- Can now implement symbolic reachability search!
Big Question

- Who gives us these predicates?

- Answer 1: The user
  - Manual abstractions
    - Given a program and property, the user figures out what are the interesting predicates
  - Dataflow analysis
    - For “generic” properties, come up with a family of predicates that are likely to be sufficient for most programs
Abstract Interpretation

• Abstract model checking is formalized through abstract interpretation
  - Formalizes and unifies semantics-based program analysis
More Approximations

• Many program dataflow analyses do not perform exact reachability analysis on the abstract state space

• Instead, use the structure of the control flow graph to further approximate the result
Example: Flow Sensitive Analysis

• For each control flow node, keep track of the set of reachable states (along any program path) to that node
  - Information may be lost at merge points by abstracting $\lor$ by something coarser

• Assumption: All paths of the control flow graph can be executed
  - Ignore conditional statements
Flow Insensitive Analysis

- Even more approximate
- Disregard the order of operations in the program!

- Much faster analysis than abstract model checking
  - But results are much cruder of course!
  - Can still be useful: e.g., primary way to perform alias analysis
When I run a model checker, it goes to compute the result and never comes back. When I run a dataflow analysis, it comes back immediately and says “Don’t know”!

- Patrick Cousot
Questions?
Lecture 2: Software Model Checking and Counterexample-Guided Refinement

Rupak Majumdar
Recap

• Model checking is an algorithmic technique to verify properties of systems

• In conjunction with abstractions, can be effective in proving subtle properties

• Today: Consider the problem of abstract model checking of (sequential) software implementations
Setting: Property Checking

- Programmer gives partial specifications
- Code checked for consistency w/ spec

- Different from program correctness
  - Specifications are not complete
  - Is there a complete spec for Word? Emacs?
Interface Usage Rules

- Rules in documentation
  - Order of operations & data access
  - Resource management
  - Incomplete, unenforced, wordy

- Violated rules ⇒ bad behavior
  - System crash or deadlock
  - Unexpected exceptions
  - Failed runtime checks
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

[Fahndrich]
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!
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;
}
What a program really is...

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

Theorem Prover

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

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

lock
old=new
¬old=new
Abstraction

State

3: unlock();
   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]
Lazy Abstraction

- C Program
- Property

BLAST

- Yes → Safe
- No → Trace
Problem: Abstraction is Expensive

Problem

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

Observe

Fraction of state space reachable
\#Preds \sim 100's, \#States \sim 2^{100},
\#Reach \sim 1000's
**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

**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
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

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;
3:                unlock();
3:                new ++;
 } }while(new != old);
4:}unlock ();
5:}

Predicates: $\text{LOCK}$

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

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
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);
5:  unlock ();
}

Reachability Tree

Predicates: LOCK
Build-and-Search

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

Reachability Tree

Predicates: LOCK
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
Analyze Counterexample

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

Reachability Tree

Predicates: LOCK
Analyze Counterexample

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

Reachability Tree

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 ();
}

Reachability Tree

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

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

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

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

SAFE
**Key Idea: Reachability Tree**

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**

**S1:** Only Abstract Reachable States

**S2:** Don’t refine error-free regions
Lazy Abstraction

Problem: Abstraction is Expensive

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

Key Idea: Reachability Tree
Technical Details

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

Predicates: LOCK, new==old

Reachability Tree
Technical Details

• **Q:** How to compute “successors”?  

• **Q:** How to find predicates?  
  [Interpolation]

• **Q:** How to analyze (recursive) procedures?  
  [Context-free reachability]
Q. How to compute “successors”?
Predicate Abstraction

- From last lecture
Q. How to find predicates ?
# Predicates grows with program size

Problem: $p_1, \ldots, p_n$ needed for verification

Exponential reachable abstract states

Tracking \texttt{lock} not enough
# Predicates grows with program size

```
while(1){
    1: if (p_1) lock();
       if (p_1) unlock();
    ...
    2: if (p_2) lock();
       if (p_2) unlock();
    ...
    n: if (p_n) lock();
       if (p_n) unlock();
}
```

Problem:

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

Exponential reachable abstract states

\[ 2^n \text{ Abstract States} \]
Predicates useful \textit{locally}

\[
\text{while}(1)\
\begin{align*}
\text{1: if } (p_1) \text{ lock() ;} \\
\text{if } (p_1) \text{ unlock() ;} \\
\ldots \\
\text{2: if } (p_2) \text{ lock() ;} \\
\text{if } (p_2) \text{ unlock() ;} \\
\ldots \\
\text{n: if } (p_n) \text{ lock() ;} \\
\text{if } (p_n) \text{ unlock() ;} \\
\end{align*}
\]

2n Abstract States

**Solution:** Use predicates only where needed

Using \textbf{Counterexamples}:

\textbf{Q1.} Find \textit{predicates}

\textbf{Q2.} Find \textit{where} predicates are needed
Lazy Abstraction

**Problem:** #Preds grows w/ Program Size

**Solution:** Localize pred. use, find where preds. needed
Counterexample Traces

\[\begin{align*}
lock() & \\
old = & new \\
q = & q\rightarrow next \\
\text{[q!=NULL]} & \\
q\rightarrow data = & new \\
unlock() & \\
new++ & \\
\text{[new==old]} & \\
unlock() & \\
\end{align*}\]

\[\begin{align*}
lock_1 = & 1 \\
old_1 = & new_0 \\
q_1 = & q_0\rightarrow next \\
assume(q_1 != NULL) & \\
(q_1 \rightarrow data)_1 = & new_0 \\
lock_2 = & 0 \\
new_1 = & new_0 +1 \\
assume(new_1=old_1) & \\
assert(lock_2=1) & \\
\end{align*}\]

\[\begin{align*}
lock_1 = & 1 \land \\
old_1 = & new_0 \land \\
q_1 = & q_0\rightarrow next \land \\
(q_1 \rightarrow data)_1 = & new_0 \land \\
lock_2 = & 0 \land \\
new_1 = & new_0 +1 \land \\
new_1=old_1 & \\
\end{align*}\]

Trace SSA Trace

Trace Feasibility Formula

Thm: Trace is feasible ⇔ TF is satisfiable
Proof of Unsatisfiability

\[\begin{align*}
\text{lock}_1 &= 1 \\
\text{old}_1 &= \text{new}_0 \\
q_1 &= q_0 \rightarrow \text{next} \\
q_1 &\neq \text{NULL} \\
(q_1 \rightarrow \text{data})_1 &= \text{new}_0 \\
\text{lock}_2 &= 0 \\
\text{new}_1 &= \text{new}_0 + 1 \\
\text{new}_1 &= \text{old}_1
\end{align*}\]

Trace Feasibility Formula

Predicates: \(\text{old}=\text{new}, \text{new}=\text{new}+1, \text{new}=\text{old}\)

Add: \(\text{old}=\text{new}\)

[HenzingerJhalaM.Sutre02]
Counterexample Traces: Take 2

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

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 SSA Trace

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

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

Trace Feasibility Formula
Proof of Unsatisfiability

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

Trace Formula

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

\[ \emptyset \]
The Present State...

Trace

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

State...

1. ... after executing trace \textit{past (prefix)}
2. ... knows \textit{present values} of variables
3. ... makes trace \textit{future (suffix)} infeasible

At \textit{pc}_4, \textit{which predicate on present state} shows infeasibility of \textit{future}?
### What 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 \quad \text{ctr}_1 = \text{ctr}_0 + 1 )</td>
</tr>
<tr>
<td>3: ( y = \text{ctr} )</td>
<td>( \land \quad y_1 = \text{ctr}_1 )</td>
</tr>
<tr>
<td>4: ( \text{assume}(x = i-1) )</td>
<td>( \land \quad x_1 = i_0 - 1 )</td>
</tr>
<tr>
<td>5: ( \text{assume}(y \neq i) )</td>
<td>( \land \quad y_1 \neq i_0 )</td>
</tr>
</tbody>
</table>
What Predicate is needed?

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

Trace Formula (TF)
$x_1 = \text{ctr}_0$
$\land \text{ctr}_1 = \text{ctr}_0 + 1$
$\land y_1 = \text{ctr}_1$
$\land x_1 = i_0 - 1$
$\land y_1 \neq i_0$

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: \( \text{assume}(y \neq i) \)

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

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

**Predicate ...**
... implied by TF **prefix**
... on **common** 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)

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

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

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

Trace
1: \( x = \text{ctr} \)
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 \( (y \neq i) \) \( \land \) \( y_1 \neq i_0 \)

Craig Interpolant
[Craig 57]
Computable from Proof of Unsat

Predicate at 4:
\[
\psi^+ y_1 = x_1 + 1
\]

Interpolate \( \Phi \)
Predicate ... ... implied by TF prefix
... on common variables
... & TF suffix is unsatisfiable
Another interpretation ...

Trace Formula

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

Predicate at 4: \( y = x + 1 \)

Interpolant \( \Phi \) =

Overapproximation of states after prefix that cannot execute suffix

Unsat = Empty Intersection = Trace Infeasible
Main Questions

Q. How to find good predicates? Where to track each predicate?

Q: How to compute interpolants? (And do they always exist?)
Another Proof of Unsatisfiability

\[ x_1 = ctr_0 \quad x_1 = i_0 - 1 \]

\[ ctr_0 = i_0 - 1 \quad ctr_1 = ctr_0 + 1 \]

\[ ctr_1 = i_0 \quad y_1 = ctr_1 \]

\[ y_1 = i_0 \quad y_1 \neq i_0 \]

\[ \emptyset \]

Rewritten Proof

\[ x_1 = ctr_0 = 0 \quad x_1 - i_0 + 1 = 0 \]

\[ ctr_0 - i_0 + 1 = 0 \quad ctr_1 - ctr_0 - 1 = 0 \]

\[ ctr_1 - i_0 = 0 \quad y_1 - ctr_1 = 0 \]

\[ y_1 - i_0 = 0 \quad y_1 - i_0 \neq 0 \]

\[ 0 \neq 0 \]
Interpolant from Rewritten Proof?

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

Trace Formula

Rewritten Proof

\[
\begin{align*}
    x_1 &= \text{ctr}_0 \\
    x_1 - \text{ctr}_0 &= 0 \\
    x_1 - i_0 + 1 &= 0 \\
    \text{ctr}_0 - i_0 + 1 &= 0 \\
    \text{ctr}_1 - \text{ctr}_0 - i &= 0 \\
    \text{ctr}_1 - i_0 &= 0 \\
    y_1 - \text{ctr}_1 &= 0 \\
    y_1 - i_0 &\neq 0 \\
    0 &\neq 0
\end{align*}
\]
Interpolant from Rewritten Proof?

\[
\begin{align*}
  x_1 &= ctr_0 \\
  \land \ & ctr_1 = ctr_0 + 1 \\
  \land \ & y_1 = ctr_1 \\
  \land \ & x_1 = i_0 - 1 \\
  \land \ & y_1 \neq i_0
\end{align*}
\]

Trace Formula

\[
\begin{align*}
  x_1 - ctr_0 &= 0 \quad \times (-1) \\
  ctr_1 - ctr_0 - 1 &= 0 \quad \times 1 \\
  y_1 - ctr_1 &= 0 \quad \times 1
\end{align*}
\]

\[
(y_1 \neq x_1) \times 1 = 0
\]

Interpolant!
### 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: \( \text{pc}_i \) \( \square \) Interpolant from cut \( i \)
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
- Pred. Map: \( \text{pc}_i \) \( \square \) Interpolant from cut i
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>$\land \quad \text{ctr}_1 = \text{ctr}_0 + 1$</td>
</tr>
<tr>
<td>3: $y = \text{ctr}$</td>
<td>$\land \quad y_1 = \text{ctr}_1$</td>
</tr>
<tr>
<td>4: assume($x = i-1$)</td>
<td>$\land \quad x_1 = i_0 - 1$</td>
</tr>
<tr>
<td>5: assume($y \neq i$)</td>
<td>$\land \quad y_1 \neq i_0$</td>
</tr>
</tbody>
</table>

- Cut + Interpolate at each point
- Pred. Map: $\text{pc}_i$ $\square$ 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
1: \( x_1 = \text{ctr}_0 \)
2: \( \land \ 	ext{ctr}_1 = \text{ctr}_0 + 1 \)
3: \( \land \ y_1 = \text{ctr}_1 \)
4: \( \land \ x_1 = i_0 - 1 \)
5: \( \land \ y_1 \neq i_0 \)

• Cut + Interpolate at each point
• Pred. Map: \( \text{pc}_i \) Interpolant from cut i

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

Interpolate
• Cut + Interpolate at each point
• Pred. Map: \( \text{pc}_i \) Interpolant from cut i
Local Predicate Use

Use predicates **needed** at **location**

- #Preds. grows with program size
- #Preds 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: \(2n\) states

Global Predicate use

Ex: \(2^n\) states
Question: When Do Interpolants Exist?

- Craig’s Theorem guarantees existence for first order logic

- But we are interpreting formulas over theories (arithmetic, theories of data structures)
The Good News

• Interpolants always exist for recursively enumerable theories
  - The proof is a simple application of compactness

• So: interpolants exist for Presburger arithmetic, sets with cardinality constraints, theory of lists, (quantifier-free) theory of arrays, multisets, ...
The Bad News

• “The proof is a simple application of compactness”
  
  - May be algorithmically inefficient
  
  - Daunting engineering task to construct interpolating decision procedure for each individual theory
An Alternate Path: Reduction

- Want to **compile formulas** in a new theory to formulas in an old theory such that interpolation in the old theory imply interpolation in the new theory

- **T reduces to R**: can compile formulas in theory $T$ to formulas in theory $R$
  - And use decision procedures for $R$ to answer decision questions for $T$

- Technically: Given theories $T$ and $R$, with $R \subseteq T$, a reduction is a computable map $\mu$ from $T$ formulas to $R$ formulas such that for any $T$-formula $\phi$:
  - $\phi$ and $\mu(\phi)$ are $T$-equivalent
  - $\phi$ is $T$-satisfiable iff $\mu(\phi)$ is $R$-satisfiable
Example: Theory of Sets

Theory of sets reduces to theory of equality with uninterpreted functions

\[
\begin{align*}
    x &= y & \forall e. & e \in x \iff e \in y \\
    x &= \emptyset & \forall e. & e \notin x \\
    x &= U & \forall e. & e \in x \\
    x &= \{e\} & e \in x & \forall e'. & e' \in x \implies e = e' \\
    x &= y \cup z & \forall e. & e \in x \iff e \in y \lor e \in z \\
    x &= y \cap z & \forall e. & e \in x \iff e \in y \land e \in z
\end{align*}
\]
Example: Theory of Multisets

Theory of multisets reduces to the combination theory of equality with uninterpreted functions and linear arithmetic

\[
\begin{align*}
    x = y & \quad \forall e. \text{count}(x, e) = \text{count}(y, e) \\
    x = \emptyset & \quad \forall e. \text{count}(x, e) = 0 \\
    x = [(e, n)] & \quad \text{count}(x, e) = \max(0, n) \\
    x = y \uplus z & \quad \forall e. \text{count}(x, e) = \text{count}(y, e) + \text{count}(z, e) \\
    x = y \uplus z & \quad \forall e. \text{count}(x, e) = \max(\text{count}(y, e), \text{count}(z, e)) \\
    x = y \cap z & \quad \forall e. \text{count}(x, e) = \min(\text{count}(y, e), \text{count}(z, e))
\end{align*}
\]
Reduction and Interpolation

\( \Psi^- \) and \( \Psi^+ \) in Theory \( T \)
\( \Phi^- \) and \( \Phi^+ \) in Theory \( R \)
Interpolant \( \alpha \) in Theory \( R \) as well as \( T \)
Quantifier-free interpolant
Reduction Theorem

- Interpolants for the theory of arrays, sets, and multisets can be computed by reduction to the combination theory of linear arithmetic and equality with uninterpreted functions

- We already have interpolating decision procedures for this latter theory
Lazy Abstraction

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

Refine
Ctrex. Trace
Trace Feas Formula
Thm Pvr
Proof of Unsat
Interpolate
Pred. Map PC \( \square \) Preds.
So far ...

Lazy Abstraction

- 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
So how well does all this work?

Quite well, if the program and property are control-dominated.

Not so well when data is involved...
### 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>kbfiltre</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
Refinement Failure: Unrolling Loops

```cpp
x = 0; y = 50;
while ( x<100 ) {
    if ( x>=50 ) y = y+1;
    x = x+1;
}
assert( y==100 );
```

- counterexample:
  x=0; y=50; x>=100; y==100
  refinement: x==0

- counterexample:
  x=0; y=50; x<100; x=x+1; x>=100; y==100
  refinement: x==1

- counterexample:
  x=0; y=50; x<100; x=x+1; x<100; x=x+1; x>=100; y==100
  refinement: x==2
- ...

Refinement Failure: Unfolding Arrays

```c
for (i=0; i<n; i++) {
    a[i]=i;
}
for (j=0; j<n; j++) {
    assert( a[j]==j );
}
```

- **counterexample:**
  
  i=0; i<n; a[i]=i; i++; i>=n;  
  j=0; j<n; a[j]!=j  
  refinement: a[0]==0

- **counterexample:**
  
  i=0; i<n; a[i]=i; i++; i<n; a[i]=i; i++; i>=n;  
  j=0; j<n; a[j]==j; j++; j<n; a[j]!=j  
  refinement: a[1]==1

- ...
What went Wrong?

• Consider all unrolled counterexamples at once
  - Convergence of abstraction discovery

• Inspect families of counterexamples of unbounded length
  - Justification for unbounded universal quantification

• Looking at one counterexample path at a time is too weak  [JhalaMcMillan05, JhalaMcMillan06]
Path Programs

- Treat **counterexamples** as programs
  - “Close” the loops

[BeyerHenzingerM.Rybalchenko07]
Meaning of Path Programs

Path program \( \uparrow \) (Possibly unbounded) sets of counterexamples:

- **Unbounded** counterexamples
  
- **Property-determined** fragment of original program
  - Can be analyzed independently to find good abstractions
Path Invariants

- Invariant for path programs \( \text{path invariant} \)
- Abstraction refinement using path invariants
  - Elimination of all counterexamples \textit{within} path program
  - Justification for \textit{unbounded} quantification
Invariant Generation

- Given a path program, with a designated error location, find an invariant that demonstrates error is not reachable
  - Can scale: Reduced obligation to program fragment
  - Outer model checking loop integrates path invariants into program invariant

- Can use any technique

- We use constraint-based invariant generation
  [SankaranarayananSipmaManna04,BeyerHenzingerM.Rybalchenko07]
Lazy Abstraction

C Program \(\rightarrow\) Abstract \(\rightarrow\) Safe

Property \(\rightarrow\) Refine \(\rightarrow\) Trace

Problem: \#Preds grows w/ Program Size

Solution: Localize pred. use, find where preds. needed

Refine

Ctrex. \(\leftrightarrow\) Trace
Trace \(\leftrightarrow\) Feas
Feas \(\leftrightarrow\) Formula

Thm Pvr \(\leftrightarrow\) Proof of
Proof of \(\leftrightarrow\) Unsat
Unsat \(\leftrightarrow\) Interpolate
Interpolate \(\leftrightarrow\) Pred. Map
Pred. Map \(\leftrightarrow\) Prefs.
So far ...

Lazy Abstraction

- 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**
Questions?
Recap ...

Lazy Abstraction

- 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
Technical Details

Q. How to analyze recursive procedures?
An example

```c
main()
{
    if (flag) {
        y = inc(x, flag);
        if (y <= x) ERROR;
    } else {
        y = inc(z, flag);
        if (y >= z) ERROR;
    }
    return;
}

inc(int a, int sign) {
    if (sign) {
        rv = a + 1;
    } else {
        rv = a - 1;
    }
    return rv;
}
```
main()
{
    if (flag){
        if (y<=x) ERROR;
        y = inc(x,flag);
        } else {
            y = inc(z,flag);
            if (y>=z) ERROR;
        }
    return;
}

inc(int a, int sign){
    if (sign){
        rv = a+1;
    } else {
        rv = a-1;
    }
    return rv;
}
Inline Calls in Reach Tree

Problem

- Repeated analysis for “inc”
- Exploding call contexts

2^n nodes in Reach Tree
Inline Calls in Reach Tree

Problem

- Repeated analysis for “inc”
- Exploding call contexts
- Cyclic call graph (Recursion)
  - Infinite Tree!
Solution: Procedure Summaries

Summaries: Input/Output behavior

- Plug summaries in at each callsite
  ... instead of inlining entire procedure
  [Sharir-Pnueli 81, Reps-Horwitz-Sagiv 95]

- Summary = set of \((F \sqsubseteq F')\)
  - \(F\) : Precondition formula describing input state
  - \(F'\) : Postcondition formula describing output state
Solution: Procedure Summaries

inc(int a, int sign){
    1: if (sign){
        2:    rv = a+1;
    } else {
        3:    rv = a-1;
    }
    4: return rv;
}

• (¬ sign=0 □ rv > a)
• (sign = 0 □ rv < a)

• Summary = set of $(F □ F')$
  - $F$ : Precondition formula describing input state
  - $F'$ : Postcondition formula describing output state

Q. How to compute, use summaries?
Q. How to compute, use summaries?
main(){
  if (flag){
    y = inc(x,flag);
    if (y<=x) ERROR;
  } else {
    y = inc(z,flag);
    if (y>=z) ERROR;
  }
  return; }

inc(int a, int sign){
  if (sign){
    rv = a+1;
  } else {
    rv = a-1;
  }
  return rv;
}

Predicates: flag=0, y>x, y<z
sign=0, rv>a, rv<a
Abstraction with Summaries

main()

1: if (flag){
2:   y = inc(x, flag);
3:   if (y<=x) ERROR;
} else {
4:   y = inc(z, flag);
5:   if (y>=z) ERROR;
}

return;

inc(int a, int sign){
1: if (sign){
2:   rv = a+1;
} else {
3:   rv = a-1;
}

4: return rv;

Predicates: flag=0, y>x, y<z

Summary: (~sign=0 □ rv>a),

sign=0, rv>a, rv<a
Summary: Successor

**main()**

1: if (flag) {
2:     y = inc(x, flag);
3:     if (y <= x) ERROR;
4: } else {
5:     y = inc(z, flag);
6:     if (y >= z) ERROR;
7: }
8: return;

**inc(int a, int sign)**

1: if (sign) {
2:     rv = a + 1;
3: } else {
4:     rv = a - 1;
5: }
6: return rv;

**Predicates:** flag=0, y>x, y<z

**Summary: (¬sign=0 □ rv>a),
sign=0, rv>a, rv<a**
Abstraction with Summaries

main()
{
  1: if (flag) {
  2:     y = inc(x,flag);
  3:     if (y<=x) ERROR;
  4: } else {
  5:     y = inc(z,flag);
  6:     if (y>=z) ERROR;
  7: }
  8: return;
}

inc(int a, int sign){
  1: if (sign) {
  2:     rv = a+1;
  3: } else {
  4:     rv = a-1;
  5: }
  6: return rv;
}

Predicates: flag=0 , y>x , y<z

Summary: (¬ sign=0  □  rv>a),

sign=0 , rv>a , rv<a
Abstraction with Summaries

main()
1: if (flag)
2: y = inc(x,flag);
3: if (y<=x) ERROR;
4: y = inc(z,flag);
5: if (y>=z) ERROR;
return;

inc(int a, int sign)
1: if (sign)
2: rv = a+1;
3: else {
4: rv = a-1;
}
return rv;

Predicates: flag=0 , y>x , y<z
sign=0 , rv>a , rv<a

Summary: (¬ sign=0 □ rv>a), (sign=0 □ rv<a)
Summary Successor

```c
main()
{
    if (flag){
        y = inc(x,flag);
        if (y<=x) ERROR;
    }
    else {
        y = inc(z,flag);
        if (y>=z) ERROR;
    }
    return;
}

inc(int a, int sign){
    if (sign){
        rv = a+1;
    }
    else {
        rv = a-1;
    }
    return rv;
}
```

Predicates:  
- flag=0  
- y>x  
- y<z  
- sign=0  
- rv>a  
- rv<a  

Summary:  
- (∼sign=0  □  rv>a),  
- (sign=0  □  rv<a)
Abstraction with Summaries

```c
main()
{
  if (flag) {
    y = inc(x, flag);
    if (y <= x) ERROR;
  } else {
    y = inc(z, flag);
    if (y >= z) ERROR;
  }
  return;
}

inc(int a, int sign)
{
  if (sign) {
    rv = a + 1;
  } else {
    rv = a - 1;
  }
  return rv;
}
```

**Predicates:**
- flag=0, y>x, y<z
- sign=0, rv>a, rv<a

**Summary:**
- (¬ sign=0 □ rv>a), (sign=0 □ rv<a)
Another Call ...

```
main()
{
  if (flag)
  {
    y = inc(x,flag);
    if (y<=x) ERROR;
  } else {
    y = inc(z,flag);
    if (y>=z) ERROR;
  }
  y1 = inc(z1,1);
  if (y1<=z1) ERROR;
  return;
}

inc(int a, int sign)
{
  if (sign)
  {
    rv = a+1;
  } else {
    rv = a-1;
  }
  return rv;
}
```

Predicates: flag=0, y>x, y<z, y1>z1, sign=0, rv>a, rv<a

Summary: (∼sign=0  □  rv>a), (sign=0  □  rv<a)
Another Call ...

```
main()
{
   if (flag){
      y = inc(x,flag);
      if (y<=x) ERROR;
   } else {
      y = inc(z,flag);
      if (y>=z) ERROR;
   }
   y1 = inc(z1,1);
   if (y1<=z1) ERROR;
   return;
}

inc(int a, int sign){
   if (sign){
      rv = a+1;
   } else {
      rv = a-1;
   }
   return rv;
}
```

Predicates: \( \text{flag=0 , y>x , y<z , y1>z1} \)

Summary: \( (\neg \text{sign=0 } \lor \text{rv>a}) \), \( (\text{sign=0 } \lor \text{rv<a}) \)
Q. How to perform interpolation in the presence of recursive calls?
Traces with Procedure Calls

Trace

\[ \text{pc}_1: x_1 = 3 \]
\[ \text{pc}_2: \text{assume } (x_1 > 0) \]
\[ \text{pc}_3: x_3 = \mathcal{f}_1(x_1) \]
\[ \text{pc}_4: y_2 = y_1 \]
\[ \text{pc}_5: y_3 = \mathcal{f}_2(y_2) = \mathcal{f}_2(y_2) \]
\[ \text{pc}_6: z_2 = z_1 + 1 \]
\[ \text{pc}_7: z_3 = 2z_2 \]
\[ \text{pc}_8: \text{return } z_3 \]
\[ \text{pc}_9: \text{return } y_3 \]
\[ \text{pc}_{10}: x_4 = x_3 + 1 \]
\[ \text{pc}_{11}: x_5 = \mathcal{f}_3(x_4) \]
\[ \text{pc}_{12}: \text{assume } x_4 < 5 \]
\[ \text{pc}_{13}: \text{return } w_1 \]
\[ \text{pc}_{14}: \text{assume } x_4 > 5 \]
\[ \text{pc}_{15}: \text{assume } (x_1 = x_3 + 2) \]

Trace Formula

\[ \mathcal{f}_1(x_1) \]
\[ \mathcal{f}_2(y_2) \]
\[ 2z_2 \]
\[ z_3 \]
\[ y_3 \]
\[ z_3 \]
\[ \mathcal{f}_3(x_4) \]
Interprocedural Analysis

Trace

Trace Formula

Find predicate needed at point i

Require at each point i:

Scoped predicates

YES: Variables visible at i

NO: Caller’s local variables
Problems with Cutting

Caller variables common to $\psi^-$ and $\psi^+$

- Unsuitable interpolant: not well-scoped
Scoped Cuts

Trace

Trace Formula

Call begins
Scoped Cuts

Trace

Predicate at $pc_i = \text{Interpolant from cut } i$

Trace Formula

Call begins

$\psi^+$

$\psi^-$
Common Variables

Trace

Trace Formula

Predicate at $pc_i = \text{Interpolant from } i\text{-cut}$
Lazy Abstraction: Summary

C Program → Abstract → Refine → Path Slice

Property → Refine

Yes → Safe

No → Trace
Lazy Abstraction: Summary

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

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

1. **Abstraction**: Expensive
   Reachability Tree, Procedure summaries

2. **Refinement**: Find predicates, use locations
   Slice irrelevant details
   Proof of unsat of TF + Interpolation
Extension

Merging CEGAR and Abstract Domains
## Comparing dataflow and CEGAR

<table>
<thead>
<tr>
<th></th>
<th>Dataflow</th>
<th>CEGAR with predicate abstraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Precision</td>
<td>- Propagates facts over infeasible branches</td>
<td>+ Analysis considers only feasible branches</td>
</tr>
<tr>
<td></td>
<td>- Dataflow facts lost at join points</td>
<td>+ Path sensitive analysis considers each path independently</td>
</tr>
</tbody>
</table>

- Dataflow is a path-insensitive analysis that propagates facts over infeasible branches. It loses dataflow facts at join points.
- CEGAR, on the other hand, is a path-sensitive analysis that considers each path independently, making it more precise.
## Comparing dataflow and CEGAR

<table>
<thead>
<tr>
<th>Fact discovery</th>
<th><strong>Dataflow</strong></th>
<th><strong>CEGAR with predicate abstraction</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Eager Analysis</td>
<td>Lazy Analysis</td>
</tr>
<tr>
<td></td>
<td>+ Quickly computes relevant facts</td>
<td>- Analysis expensive and predicates not always found</td>
</tr>
<tr>
<td></td>
<td>- Not adaptive, may become overwhelmed with irrelevant facts</td>
<td>+ Discovers relevant facts through counterexamples</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Domain-specific analyses</th>
<th><strong>Dataflow</strong></th>
<th><strong>CEGAR with predicate abstraction</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>+ Easily adapted, flexible representation of facts</td>
<td>- Predicates not convenient for domain-specific analyses</td>
</tr>
</tbody>
</table>

Can we combine the best of both approaches?
Predicated lattices

• For each program location, track a set of first-order predicates over program variables

• Rather than track dataflow facts directly, track a predicated lattice:
  - New dataflow facts:
    map: predicates → original dataflow facts
  - Only join facts associated with the same set of predicates
  - Default predicate: true

• Instance of reduced cardinal powers of lattices
  [Cousot² 79]
Example

1: out = fopen(...);

2: fprintf(out, ...);

3: fclose(out);

true → out→O

OK

true → out→C
Example

4: flag = \{0,1\};

5: if (flag)
6:   out = fopen(...);
7: else
8:   if (flag)
9:     fprintf(out, ...);

Error!

Facts lost due to join

Add predicates:
flag!=0, flag==0

out→C
out→O
out→C
out→T
out→T
Repeat with new predicates

4: flag = \{0,1\};

5: if (flag)
6:   out = fopen(...);
7: else
8: if (flag)
9:   fprintf(out, ...);

Both facts kept at join

true \rightarrow \text{out} \rightarrow C

flag!=0 \rightarrow \text{out} \rightarrow O

flag==0 \rightarrow \text{out} \rightarrow C

flag==0 \rightarrow \text{out} \rightarrow C

flag!=0 \rightarrow \text{out} \rightarrow O

OK
Symbolic execution lattice

- Tracks values of program variables and heap
  - Facts at each program point represented as map from *names* to *values*
  - *Symbolic execution* used to compute new facts after each statement
  - A “static” version of the DART algorithm
Example

```c
int x = 0;
int y = 1;
int *z;

if (*) {
    z = &x;
}
else {
    z = &y;
}

*z = 5;

assert (*z == 5);
```

How to extend this to liveness properties?

Specifically: *Reasoning about termination*
When does a Program Terminate?

• Iff its reachable transition relation is well-founded

• Reachable transition relation = TR(x,x') ∩ Reach(x) × Reach(x') = Restriction of the transition relation to the set of reachable states
Well-Founded Relation

• A binary relation \( > \) is well-founded if there is no infinite descending sequence

• No \( s_0, s_1, s_2, \ldots \) such that
  \[
  s_0 > s_1 > s_2 > \ldots 
  \]

Example: \( > \) on natural numbers
But not \( > \) on integers
Idea: Rank Functions

- Fix a set $X$, and $> a$ wf relation on $X$

- Suppose I can map each reachable state $s$ of the transition graph to a rank $r(s) \in X$ s.t.

  $s \rightarrow s'$ implies $r(s) > r(s')$

Then the system must terminate

The converse is also true
Example

Input x, n
While(x <= n) x++;

Terminates, using (roughly) the rank function n - x

Does it, really?
Disjunctive Rank Functions

• In many cases, finding a single wf relation can be difficult

• Suppose I can find wf relations $T_1, \ldots, T_k$ such that $RTR \subseteq T_1 \cup \ldots \cup T_k$

• Does the program terminate?
  - Not in general (Why?)
Disjunctive Well-foundedness

If $T_1...T_k$ are wf relations and $R^+ \subseteq T_1 \cup ... \cup T_k$
Then: $R$ is well-founded

Such $R$ is called \textit{disjunctively well-founded}
Disjunctive Well-foundedness

\[ P \text{ terminates if } TR \cap \text{Reach} \times \text{Reach is disjunctively well-founded} \]

Useful: Can consider individual portions of the program independent of other parts
Incremental Termination

$T = \text{emptyset}$

While $TR^+ \text{ not included in } T$:
  invariant: $T$ is a finite union of $\text{wf}$ relations
  find abstract counterexample to $\text{wf}$
  if concretely feasible
    does not terminate
  otherwise find $\text{wf}$ relation $T'$

$T = T \cup T'$
Counterexample to Termination

• Lasso = Stem + Cycle
  - Represents infinite execution
    Stem Cycle Cycle ...

Needs rank-finding technique to find a wf relation showing lasso cannot be executed arbitrarily (Heuristics exist)
Reduction to Safety

• How to check if $R^+ \subseteq T$ for the reachable transition relation?

• Can reduce check to safety
• Run program parallel with a monitor for $T$
  - runs in parallel with the program
  - inspects pairs of states wrt. $T$
  - goes to error if observes $(s, s') \not\in T$
  - Use non-determinism to perform check
Reduction to Safety: Idea

selected := ⊥
phase := SELECT

while True {
    switch (phase) {
        SELECT: if ( nondet() ) {
            selected := current
            phase := CHECK
        }
        CHECK: if ( (selected, current) ∉ T ) { ERROR: }
    }
}
Terminator

- Input: program written in C

- Language features supported
  - nested loops, gotos
  - aliasing
  - (mutually) recursive function calls

- Output:
  - termination proof: transition invariant
  - counterexample: lasso = stem + cycle

- Scalability: (on drivers from WinDDK)
Questions?
Lecture 4: Some Techniques for Infinite State

Rupak Majumdar
Topics

1. Safety and liveness verification for asynchronous programs

1. Case splitting and symmetry reduction for parameterized programs
Safety and liveness verification for asynchronous programs
Asynchronous Programs

Useful Model:
- Distributed Systems
- Web Servers
- Embedded Systems

Languages and Libraries:
- LibAsync, LibEvent, ...
- NesC and TinyOS
- Go
- AJAX

Requests queued and executed asynchronously by cooperative scheduler
Asynchronous Programs

```c
global bit b = 0;
main(){
    ...
    async h1();
    ...
}
h1(){
    if(b == 0){
        async h1();
        async h2();
        return;
    }
}
h2(){
    b = 1;
    ...
    return;
}
```

- **main ends in dispatch location**
  - Calls asynchronously posted functions

- **Async calls stored** in task buffer

- Scheduler picks a pending task and runs it to completion
Asynchronous Program Execution

- Execution starts in `main`
- Task buffer empty

```c
global bit b = 0;

main()
{
    ... 
    async h1();
    ...
}

h1()
{
    if(b == 0){
        async h1();
        async h2();
        return;
    }
}

h2()
{
    ... 
    b = 1;
    ... 
    return;
}
```

Pending Calls
- h1

State: b = 0
Asynchronous Program Execution

- Execution enters dispatch loop
- Picks pending call and executes it
- Returns to dispatch loop on return

```
global bit b = 0;

main(){
...
  async h1();
  ...
}

h1(){
  if(b == 0){
    async h1();
    async h2();
    return;
  }
}

h2(){
  ...  
  b = 1;
  ...  
  return;
}

Pending Calls
h1
h2

State: b = 0
```
Asynchronous Program Execution

```c
// Global bit
bool b = 0;

// Main function
void main(){
    // Pick another pending call
    async h1();
}

// Function h1
void h1(){
    if(b == 0){
        async h1();
        async h2();
        return;
    }
}

// Function h2
void h2(){
    // Update state
    b = 1;
    // Pick another pending call
    return;
}
```

Pending Calls
- h1
- h2

State: b = 0
Asynchronous Program Execution

```
global bit b = 0;
main()
  ...
  async h1();
  ...
  }

h1()
  if(b == 0){
    async h1();
    async h2();
    return;
  }

h2()
  ...
  b = 1;
  ...
  return;

PC

• Pick some pending task

Pending Calls
  h2
  h1
  h2

State: b = 1
```
Asynchronous Program Execution

```c
global bit b = 0;
main(){
    ...
    async h1();
    ...
}
h1(){
    if(b == 0){
        async h1();
        async h2();
        return;
    }
}
h2(){
    ...
    b = 1;
    ...
    return;
}
}

• Pick some pending task

Pending Calls
h2
h1

State: b = 1
```
Asynchronous Program Execution

```c
global bit b = 0;
main()
{
    ...
    async h1();
    ...
}

h1()
{
    if(b == 0)
    {
        async h1();
        async h2();
        return;
    }
}

b = 1;
return;
}

h2()
{
    ...
    return;
}
```

- And the program terminates

Pending Calls

h2

State: b = 1
Observations

```javascript
global bit b = 0;
main()
{
    ...
    async h1();
    ...
}

h1()
{
    if(b == 0){
        async h1();
        async h2();
        return;
    }
}

h2()
{
    ...
    b = 1;
    ...
    return;
}
```

- Call stack for a handler can be unbounded
- Shared global state can change between posting and execution
- Task Buffer can be unbounded
  - Consider \( n \) runs of h1 before h2
  - Task buffer contains \( n \) h1, 1 h2
Properties: Safety

Given a **Boolean** asynchronous program and a control location in a handler

Is there an execution which reaches the control location?

We do not care about the task buffer
- Handlers cannot take decisions based on the contents of the task buffer
### Properties: Termination

Given a **Boolean asynchronous program**, does it terminate?

- **main** *does not* terminate on all runs
  - What if `h1` is chosen over `h2` forever?

```c
global bit b = 0;
maint()
  ...  
  async h1(); 
  ... 
}

h1()
  if(b == 0){
    async h1();
    async h2();
    return;
  }
}

h2()
  ... 
  b = 1;
  ...  
  return;
}
An infinite execution is *fair* if
- For every handler $h$ that is pending
- The scheduler eventually picks and runs an instance of $h$

- So: the run choosing h1 over h2 always is not fair

- Will focus on *fair runs*
  - *Captures the intuition that scheduling is fair*
Asynchronous Program Analysis

Main Result:
Safety and Liveness decidable for asynchronous programs
[SenViswanathan06, JhalaM07, GantyMRybalchenko09, GantyM09]

This is hard because: Not finite state or context free:
- potentially unbounded stacks,
- potentially unbounded request queue

This is useful because:
- Asynchronous programs used in many correctness-critical settings
- The style breaks up control flow, making it difficult to reason about code
First Attempt: Does not Work

- Treat asynchronous calls as synchronous and use interprocedural reachability

- Why?
  - Global state can change between posting a task and executing it
First Attempt: Does not Work

- In synchronous computation, the assert holds
- In asynchronous computation, p2 can execute before p1
Second Attempt

```
global bit b = 0;
main(){
  ...
  async h1();
  ...
}
h1(){
  if(b == 0){
    async h1();
    async h2();
    return;
  }
}
h2(){
  ...
  b = 1;
  ...
  return;
}
```

Reduce to sequential analysis

- Add a **counter for each handler** (tracks number of pending instances)
- An async call **increments counter**
- Dispatch loop chooses a non-zero counter, **decrements it and runs corresponding handler**
Second Attempt

global bit b = 0;
    ch1=0, ch2=0;
main(){
    ...
    ch1++;
    while(ch1>0 || ch2>0){
        pick h s.t. ch > 0
        ch--;
        h();
    }
}
h1(){
    if(b == 0){
        ch1++;
        ch2++;
        return;
    }
}
h2(){
    ...
    b = 1;
    ...
    return;
}

Reduce to sequential analysis

- Add a counter for each handler (tracks number of pending instances)
- An async call increments counter
- Dispatch loop chooses a non-zero counter, decrements it and runs corresponding handler

- Sound, but decidability not obvious
  - In general, analyses undecidable for sequential programs with counters
Technique

1. Convert an AP to one without recursion

2. Convert recursion-free AP to Petri nets
Removing the Stack

- Need to summarize the effect of handlers on the task buffer
  - Observation: Just the number of async calls matter, not the order in which they are made

- Parikh’s Lemma: For every context free language $L$, there is a regular language $L'$ such that:
  - For every $w$ in $L$, there is a permutation of $w$ in $L'$ and conversely

- So: can replace a recursive handler with a non-recursive one while maintaining the summary
Example

```javascript
H() {
  if (*) {
    async a();
    H();
    async b();
  }
}
```

This gets rid of unbounded stacks in handlers
From now on, only consider `recursion-free AP`

But there is a second source of unboundedness (the task buffer)
**Petri Nets**

Can convert an asynchronous program into a *Petri net (PN)* s.t. every execution of the async program is equivalent to an execution of the Petri net.

In particular:

- The asynchronous program is *safe* iff the Petri net is *coverable*.
- The asynchronous program *fairly terminates* iff the Petri net *has no fair infinite runs*. 
Petri Nets 101

• Set of places
• Set of transitions
• Places marked with tokens
• Transition function takes tokens from sources of transitions to destinations
• With an initial marking, defines an infinite state system
• But with good decidability properties
Petrification

Tokens:
- Control flow in each handler
- Pending handler calls

Code for h1

```
global bit b = 0;
main()
{
    ...
    async h1();
    ...
}

h1()
{
    if(b == 0){
        async h1();
        async h2();
        return;
    }
}

h2()
{
    ...
    b = 1;
    ...
    return;
}
```
That’s Good, Because...

Can apply algorithmic results on Petri Nets to reason about async programs
- Many strong decidability results
Coverability Problem for PN

• Given Petri net $P$
  - (Places, transitions, initial marking)

• Target marking $m$
  - Places of $P$ $\rightarrow$ tokens

• Is there an execution of $P$ to a marking $m'$ with $m \leq m'$?
  - $m \leq m'$ if $m(p) \leq m'(p)$ for each place
Karp Miller Tree

- Build a tree with markings as nodes
- Transitions as edges
- Infinite tree
- But accelerate
- Terminates by Dickson’s Lemma
Safety Verification

Corollary to Coverability Graph: Safety verification decidable

- Remove recursion using Parikh’s lemma
- Build Petri net, construct coverability tree
- Check if there is a node in the coverability tree with a non-zero entry in the special location

- Similar proof in [SenViswanathan06] using multiset rewrite systems
- Independent, alternate algorithm in [JhalaM07]
Boundedness Verification

An AP is **bounded** if there is some $N$ such that the task buffer always has fewer than $N$ tasks.

Reduce to checking **boundedness of PN**
- Algorithm based on coverability tree construction
Fair Termination

• Given:
  - An asynchronous program with Boolean variables

• Check:
  - There is no fair infinite run

• Two checks:
  (a) Each called handler terminates
  (b) There is no infinite fair execution of handlers

LTL model checking for pushdown systems [Walukiewicz, Steffen]
Hardness: PN Reachability

- Given Petri net $P$
  - (Places, transitions, initial marking)
- Target marking $m$
  - Places of $P$ -> tokens
- Is there an execution of $P$ to marking $m$?

- Known: Decidable, EXPSPACE-hard
- Best known algorithms: non-primitive recursive space
From FT to PN Reachability

• Given PN and place $p$, construct AP such that AP has fair infinite run iff PN reaches a marking where $p$ has no tokens
  - Known to be equivalent to PN reachability

• Simulate a PN using an AP
  - One task for each place
  - One task to simulate transitions

• Post a task to guess when $p$ has no tokens
Idea of Reduction

• Simulate a PN using an AP
  - One task for each place
  - One task to simulate transitions

• Post a task to guess when p has no tokens
  - If task for p is scheduled after guess, then guess was wrong, so shut down program
  - Otherwise execute a simple infinite loop

Fairness ensures the guess is eventually made and eventually falsified (if wrong)
Fair Termination is Decidable

• First part the same:
  - Remove stack using Parikh’s lemma
  - Construct Petri net

• Use a logic over Petri nets [Yen90] to encode the fairness condition
  - Complexity of the logic: polynomial reduction to PN reachability
  - Note: Original paper [Yen90] had a mistake
Main Result

Theorem: [GantyM.09]

Checking safety and fair termination is decidable for asynchronous programs

Safety EXPSPACE-complete for asynchronous automata
  - Using a more careful construction
Observations

• For async programs, fair termination computable
  - Unlike threads [Ramalingam 00]
  - Known that threads and events are dual [NeedhamLauer]

  - So what gives?
    - We restrict global state to be finite!
    - A thread can encode unbounded information in its stack
Topic 2

Parameterized Verification of Software
A Simple Example

- Global int ctr = 0;

- Arbitrary number of threads

- Each thread executes:
  \[ ctr = ctr + 1; \]
  \[ assert (ctr > 0) \]
  \[ ctr = ctr - 1; \]

Show assertion holds
Parameterized Verification

• Intuitively, the assertion holds

• For any fixed number $N$ of threads, can prove the assertion by finite state model checking
  - $\text{Ctr}$ bounded by $N$

• How can we prove the system for any number of threads?
Idea: Temporal Case Splitting

- To show the property for all threads,
  - fix an arbitrary thread, and
  - prove the property for that thread

- Abstraction: Consider thread tid, and environment “not tid” for all threads whose id is not tid

- In general, has Skolem variables and abstract system into Skolems and others
Conclusion
Verification by Theorem Proving

1. Loop Invariants
2. Logical formula
3. Check Validity

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;
}"
Verification by Theorem Proving

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;
}
Verification by Program Analysis

1. Dataflow Facts
2. Constraint System
3. Solve constraints

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;
}
Verification by **Model Checking**

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

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

- Pgm $\rightarrow$ Finite state model
- State explosion
  + State Exploration
  + Counterexamples

**Precise** [SPIN, SMV, Bandera, JPF]
Combining Strengths

**Theorem Proving**
- loop invariants
+ Behaviors encoded in logic
Refine
+ Theorem provers
Computing Successors, Refine

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

**Lazy Abstraction**

**Model Checking**
- Finite-state model, state explosion
+ State Space Exploration
Path Sensitive Analysis
+ Counterexamples
Finding Relevant Facts
Toward More Reliable Systems

• **Tools and theory** are equally important
  - Tools reinforce process change, amplify programmer productivity, ensure quality
  - Theoretical progress drives better tools
  - Applications motivate good theory

• System development is too complex to be reduced to a single problem or solution
  - Model checking is not the only tool
  - Just tools will not make software perfect
  - How can developers and verifiers interact better?
Conclusions

• Software systems need to be more reliable
  - Enormous resources spent in stabilizing software

• Long way to go
  - Inadequate languages, inaccurate tools
  - Make system developers and system verifiers interact

• Tools, by themselves, are not enough
  - Tools and analysis is one view, but social aspects are also important
Thank You!

Acknowledgments:

Tom Ball, Dirk Beyer, Michael Emmi, Jeff Fischer, Pierre Ganty, Tom Henzinger, Ranjit Jhala, Shaz Qadeer, Sriram Rajamani, Andrey Rybalchenko, and many others ...