tball@microsoft.com

• Falcons, Apple [,1981
• B.A. Cornell, 1987
• Ph.D. Univ. Wisc., 1993
• AT&T Bell Labs, 1993-96
• Lucent Technologies Bell Labs, 1996-99
• Microsoft Research, 1999-present
• Research interests
  – software reliability
  – programming languages, program analysis, model checking, automated theorem proving
Software Development

Software SLAM → Productivity Static

Zap Testing → theorem Verifier prover

Bartok & Phoenix backends
Testing, Verification and Measurement

• Tom Ball
• Madan Musuvathi (Stanford)
• Shuvendu Lahiri (CMU)
• Nachi Nagappan (NCSU)

• Visitors
  – Orna Kupferman (Hebrew Univ.), Mooly Sagiv (Tel-Aviv Univ.), Andrei Voronkov (Univ. Manchester), Andreas Zeller (Univ. Saarland)
  – Domagoj Babic, Sumit Gulwani, Krishna Mehra, Roman Manevich, Carlos Pacheco, Greta Yorsh
Microsoft Research: University Relations

- Hiring Ph.D.s
- Fellowships
- Summer internships
- New faculty awards
- Research grants in selected areas
- Sabattical
- Faculty Summit
Automatic

Abstract
Automating Verification of Software

• Remains a “grand challenge” of computer science

• Behavioral abstraction is central to this effort
  – abstractions simplify our view of program behavior
  – proofs over the abstractions carry over to proofs over the program
Reachability

States

unsafe

unreachable

reachable

init
Safe Invariants

- Q is a safe invariant if
  - init \subseteq Q
  - T(Q) \subseteq Q
  - Q \subseteq \text{safe}
Abstraction = Overapproximation of Behavior
More Concretely

do {
    KeAcquireSpinLock();
    nPacketsOld = nPackets;
    if(request){
        request = request->Next;
        KeReleaseSpinLock();
        nPackets++;
    }
} while (nPackets != nPacketsOld);

KeReleaseSpinLock();
do {
    KeAcquireSpinLock();
    nPacketsOld = nPackets;
    if(request){
        request = request->Next;
        KeReleaseSpinLock();
        nPackets++;
    }
} while(nPackets!=nPacketsOld);

KeReleaseSpinLock();
s:=U;
do {
    assert(s=U); s:=L;
    if(*){
        assert(s=L); s:=U;
    }
} while (*);
assert(s=L); s:=U;
Overapproximation Too Large!
do {
  KeAcquireSpinLock();
  nPacketsOld = nPackets;
  if(request){
    request = request->Next;
    KeReleaseSpinLock();
    nPackets++;
  }
} while(nPackets!=nPacketsOld);

KeReleaseSpinLock();
Refined Boolean Abstraction

\[
\begin{align*}
\text{s} &:= \text{U}; \\
\text{do } & \{ \\
\text{assert } (s = \text{U}) \quad & \quad \text{s} := \text{L}; \\
\text{b} & := \text{true}; \\
\text{if } (*) & \{ \\
\text{assert } (s = \text{L}) \quad & \quad \text{s} := \text{U}; \\
\text{b} & := \text{b} ? \text{false} : *; \\
\} & \text{ while } ( \text{!b} ); \\
\text{assert } (s = \text{L}) \quad & \quad \text{s} := \text{U};
\end{align*}
\]
Invariant

"The lock is held of the loop if
Software Verification:
A Search for Abstractions

• A complex search space with a fitness function (false errors)
  – search for right abstraction
  – search within state space of abstraction

• Can a machine beat a human at search for the right abstractions?
Overview

• Part I: Abstract Interpretation
  – [Cousot & Cousot, POPL’77]
  – *Manual abstraction and refinement*
  – ASTRÉE Analyzer

• Part II: Predicate Abstraction
  – [Graf & Saïdi, CAV ’97]
  – *Automated abstraction and refinement*
  – SLAM and Static Driver Verifier

• Part III: Comparing Approaches
Concrete System

\[ \text{Prog} = (c, I, T, c : \text{infinite}) \]
Safe Invariants

- $I \subseteq Q$
- $T(Q) \subseteq Q$
- $Q \subseteq F$
$\alpha$ and $\gamma$ function

$\alpha$ maps a set of abstract element

$\alpha : 2^c \rightarrow A$
Abstraction

Sets of states ordered
Abstraction

Sets of states ordered
Abstraction

Sets of states ordered
A

embedded

with

lattice

in

Abs

Ordering
Ordering in Abs

A embedded with in lattice A = S.
Ordering in Abs

A embedded with in lattice $A = S$. 
Galois Connection
Example:

\[ D = 2 \text{int} \]
Example: \[ D = 2 \int^1 \]
Abstract Transition Relation
Signs Transition

\[ x := c \]
Assume \((x > 0)\);
Signs                     Transition

assert (x > 0);
\textbf{Abstract}

\begin{equation*}
X := \alpha(I);
\end{equation*}

\textbf{Fixpoint}

\begin{equation*}
\text{while } X \subseteq \alpha(F) \quad X \leftarrow X
\end{equation*}
Example

\[ x := 0 \]

while \( x < \)
Example

```
let
    x := 0;

while x < 10
    T
```
Signs  

Transition  

assert \( x \leq 10,000 \)
Refinement

- signs

- intervals
Effective computable approximations of an [in]finite set of points; Signs

\[
\begin{align*}
\{ & x \geq 0 \\
& y \geq 0
\end{align*}
\]

---

Effective computable approximations of an [in]finite set of points; Intervals

\[
\begin{align*}
\{ \ &x \in [19, 77] \\
\ &y \in [20, 03] 
\end{align*}
\]

---

Effective computable approximations of an\n[in]finite set of points; Octagons$^5$

$$\begin{cases} 1 \leq x \leq 9 \\
x + y \leq 77 \\
1 \leq y \leq 9 \\
x - y \leq 99 \end{cases}$$

---

Effective computable approximations of an [in]finite set of points; Polyhedra

\[
\begin{align*}
19x + 77y &\leq 2004 \\
20x + 03y &\geq 0
\end{align*}
\]

---


Slide courtesy of Patrick Cousot
Overview

• Part I: Abstract Interpretation
  – [Cousot & Cousot, POPL’77]
  – Manual abstraction and refinement
  – ASTRÉE Analyzer

• Part II: Predicate Abstraction
  – [Graf & Saïdi, CAV ’97]
  – Automated abstraction and refinement
  – SLAM and Static Driver Verifier

• Part III: Comparing Approaches
Abstract Interpretation, So Far

- Create abstract domain and supporting algorithms
- Relate domains via $\alpha$ and $\gamma$ functions
- Prove Galois connection
- Create abstract transformer $T#$
- Show that $T#$ approximates $\alpha \circ T \circ \gamma$
- Refinement to reduce false errors
- Widening to achieve termination
Example

\begin{algorithm}
\State $x := 0$
\While {$x < 100$} \\
\end{algorithm}
Diagram from Cousot, Cousot, POPL 1977
Interval

Transition

\[ X_0 = C \]

\[ X := X + 1 \]
\[ x_1 := 0 \]

while \[ x < 10.0 \]

\[ x_2 \]

\[ x_3 \]

\[ x_4 \]

\[ x_1 = [0, 0] \]

\[ x := x + 1 \]
\[
\text{while} \quad x \geq 0 \\
\quad x \\
\end{array}
\]
Interval

\[ \text{old} \]

Widening

\[ \text{new} \]

\[
\left[ l_0, u_0 \right] \lor \left[ l_1, u_1 \right] = \]

\[
\left[ \begin{array}{l}
\text{if } l_1 < l_0 \text{ then } -\infty
\end{array} \right]
\]
Abstract

\[ X := \alpha(I); \]

while \( X \subseteq \alpha(F) \)

\[ X^1 := X \]
\(x_1 \quad x := 0\); 

while \(x < 10,000\) 

\(x_1 = [0,0]\) 

\(x := x + 1\)
ASTRÉE analyzes structured C programs, without dynamic memory allocation and recursion.

In Nov. 2003, ASTRÉE automatically proved the absence of any run-time error in the primary flight control software of the Airbus A340 fly-by-wire system

a program of 132,000 lines of C analyzed in 1\textsuperscript{h}20 on a 2.8 GHz 32-bit PC using 300 Mb of memory
Abstraction Refinement: PLDI’03 Case Study of Blanchet et al.

• “… the initial design phase is an iterative manual refinement of the analyzer.”

• “Each refinement step starts with a static analysis of the program, which yields false alarms. Then a manual backward inspection of the program starting from sample false alarms leads to the understanding of the origin of the imprecision of the analysis.”

• “There can be two different reasons for the lack of precision:
  – some local invariants are expressible in the current version of the abstract domain but were missed
  – some local invariants are necessary in the correctness proof but are not expressible in the current version of the abstract domain.”
Part I: Summary

- Create abstract domains and supporting algorithms
- Relate domains via $\alpha$ and $\gamma$ functions
- Prove Galois connection
- Create abstract transformer $T#$
- Show that $T#$ approximates $\alpha \circ T \circ \gamma$
- Refinement to reduce false errors
- Widening to achieve termination
Overview

• Part I: Abstract Interpretation
  – [Cousot & Cousot, POPL’77]
  – Manual abstraction and refinement
  – ASTRÉE Analyzer

• Part II: Predicate Abstraction
  – [Graf & Saïdi, CAV ’97]
  – Automated abstraction and refinement
  – SLAM and Static Driver Verifier

• Part III: Comparing Approaches
Boolean Abstraction

\[ b : (n\text{PacketsOld} == n\text{Packets}) \]

\[
\text{do} \{ \\
\hspace{1cm} \text{KeAcquireSpinLock}(); \\
\hspace{1cm} n\text{PacketsOld} = n\text{Packets}; \\
\hspace{1cm} \text{if}(\text{request}){ \\
\hspace{2cm} \text{request} = \text{request}\rightarrow\text{Next}; \\
\hspace{2cm} \text{KeReleaseSpinLock}(); \\
\hspace{2cm} n\text{Packets}++; \\
\hspace{1cm} } \text{while}(n\text{Packets}! = n\text{PacketsOld}); \\
\}
\]

\[
\text{KeReleaseSpinLock}(); \\
\text{do} \{ \\
\hspace{1cm} \text{assert}(s=U); s:=L; \\
\hspace{1cm} b := \text{true}; \\
\hspace{1cm} \text{if}(*){ \\
\hspace{2cm} \text{assert}(s=L); s:=U; \\
\hspace{2cm} b := b ? \text{false} : *; \\
\hspace{1cm} } \text{while}(n\text{Packets}!=n\text{PacketsOld}); \\
\}
\]

\[
\text{assert}(s=L); s:=U; \\
\]
Counterexample-driven Abstraction Refinement

C Prog

Predicate abstraction

boolean program

Symbolic reachability

Path feasibility & predicate discovery

Refinement predicates

SLIC Rule

[Clarke et al. ’00]
[Ball, Rajamani ’00]
Part II: Overview

• Predicate Abstraction

• Symbolic Reachability with BDDs

• Predicate Refinement
Predicate Abstraction

– Graf & Saïdi, CAV ’97

• Idea
  – Given set of predicates \( P = \{ P_1, \ldots, P_k \} \)
    • Formulas describing properties of system state

• Abstract State Space
  – Set of Boolean variables \( B = \{ b_1, \ldots, b_k \} \)
    • \( b_i = \text{true} \iff \) Set of states where \( P_i \) holds
Approximating concrete states

Fundamental Operation
- Approximating a set of concrete states by a set of predicates
- Requires exponential number of theorem prover calls in worst case

Compute Symbolically
- Main Operation
\[ \exists X. \left[ \psi \land \left( \land_i b_i \equiv P_i \right) \right] \]

Partitioning defined by the predicates

Similar to existential abstraction of finite state machines [Clarke, Grumberg, Long]
Abstraction $\alpha$ and Concretization $\gamma$

Functions

$\alpha : 2^c \rightarrow A$
Abstraction $\alpha$ and Concretization $\gamma$

Functions

$\alpha : 2^c \rightarrow A$

$2^c \subseteq y$
Abstraction $\alpha$ and Concretization $\gamma$

Functions

$\alpha : 2^c \rightarrow A$

$2^c \rightarrow \mathcal{F}$
Example

\[ \psi = (x = 1) \]
Example

\exists x. \ (x = 1 \lor \neg x)
Example

\[ \exists x, \quad (x = 1 \lor \neg x) \]
Example

\[ \exists x, \quad (x = 1 \lor \neg) \]
Example

\[ \exists x, \quad (x = 1 \lor \)
Alternatively, check if $\psi$ against

$(x = 1, \ldots, x = 6)$
Abstracting Assigns via WP

- $WP(x:=e, Q) = Q[x \rightarrow e]

- $WP(y:=y+1, \ y<5) = (y<5) \ [y \rightarrow y+1] = (y+1<5) = (y<4)$
WP Problem

- \( WP(s, p_i) \) not always expressible via \( P \)

- Example
  - \( P = \{ x=0, x=1, x<5 \} \)
  - \( WP( x:=x+1, x<5 ) = x<4 \)
$\text{Implies}_F(e)$ and $\text{ImpliedBy}_F(e)$
Abstracting Assignments

- if $\text{Implies}_P(WP(s, p_i))$ is true before $s$ then
  - $p_i$ is true after $s$

- if $\text{Implies}_P(WP(s, \neg p_i))$ is true before $s$ then
  - $p_i$ is false after $s$

$b_i := \text{Implies}_P(WP(s, p_i)) \ ? \ true : \ \text{Implies}_F(WP(s, \neg p_i)) \ ? \ false : \ *$
Assignment Example

Statement: y := y+1;
Predicates in P: {x=y}

Weakest Precondition:
WP(y:=y+1, x=y) = x=y+1

\[ \text{Implies}_F( x=y+1 ) = ? \]
\[ \text{Implies}_F( x\neq y+1 ) = ? \]
Assignment Example

Statement:  
y := y+1;  

Predicates in P:  
{x=y}

Weakest Precondition: 
WP(y:=y+1, x=y) = x=y+1

\[ \text{Implies}_F( x=y+1 ) = \]
\[ \text{Implies}_F( x\neq y+1 ) = \]

Abstraction of assignment in B: 
b = b ? false : *;
Abstracting Assumes

• assume(e) is abstracted to:
  \[
  \text{assume}( \text{ImpliedBy}_P(e) )
  \]

• Example:
  \[
  P = \{ x=2, x<5 \}
  \]
  assume(x < 2) is abstracted to:
  \[
  \text{assume}( \{x<5\} \&\& \neg \{x==2\} )
  \]
Assume, explained

if "assume"

evaluates

then if must eval.
Refined Boolean Abstraction

\[
s := U; \\
\text{do } \{ \\
\quad \text{KeAcquireSpinLock()}; \\
\quad \text{nPacketsOld} = \text{nPackets}; \\
\quad \text{if} (\text{request}) \{ \\
\qquad \text{request} = \text{request} \rightarrow \text{Next}; \\
\qquad \text{KeReleaseSpinLock()}; \\
\qquad \text{nPackets}++; \\
\quad \} \text{ while (nPackets}! = \text{nPacketsOld}); \\
\} \text{ while (} !b \text{);} \\
\text{KeReleaseSpinLock()};
\]

\[
\begin{aligned}
\textbf{b} &: (\text{nPacketsOld} == \text{nPackets}) \\
s &:= U; \\
\text{do } \{ \\
\text{assert}(s=U); s := L; \\
\text{b} &:= \text{true}; \\
\text{if}(*) \{ \\
\text{assert}(s=L); s := U; \\
\text{b} &:= \text{b} \ ? \text{false} : \,*; \\
\} \\
\} \text{ while (} !b \text{);} \\
\text{assert}(s=L); s := U;
\end{aligned}
\]
Aside

Predicate

-procedures

abstraction
Part II: Overview

• Predicate Abstraction

• Symbolic Reachability with BDDs

• Predicate Refinement
Reachability in Boolean Programs

```plaintext
bool id (bool x, bool z)

decl y j

l1 : y := !x j
```
Reachability in Boolean Programs

```
bool id (bool x, bool z)

decl y j

ll : y := ! x j
```
Reachability in Boolean Programs

bool id (bool x, bool z)

decl y j

L1: y := ! x j
Reachability in Boolean Programs

```
bool id ( bool x, bool z )

decl y ;

l1 : y := ! x ;

x
```
Reachability in Boolean Programs

```plaintext
bool id (bool x, bool z)

decl y j

Li : y := ! x j
```
Binary Decision Diagrams

• Acyclic graph data structure for representing a boolean function (equivalently, a set of bit vectors)
• \( F(x,y,z) = (x=y) \)
Binary Decision Diagrams

• Acyclic graph data structure for representing a boolean function (equivalently, a set of bit vectors)
• \( F(x,y,z) = (x=y) \)
Binary Decision Diagrams

- Acyclic graph data structure for representing a boolean function (equivalently, a set of bit vectors)
- $F(x, y, z) = (x=y)$
Hash Consing + Variable Elimination
Aside

How to deal with procedure calls in

Program
Part II: Overview

• Predicate Abstraction

• Symbolic Reachability with BDDs

• Predicate Refinement
Refinement
Refinement
Algorithm

for R

path = So ... 

e = error

for i = k downto
Abstraction (via Boolean program)

```c
do {
    KeAcquireSpinLock();

    nPacketsOld = nPackets;

    if(request){
        request = request->Next;
        KeReleaseSpinLock();
        nPackets++;
    }
} while(nPackets!=nPacketsOld);

KeReleaseSpinLock();
```
Abstraction (via Boolean program)

\[
do \{ \\
    \text{KeAcquireSpinLock}(); \\
    \text{nPacketsOld} = \text{nPackets}; \\
    \text{if}(request)\{ \\
        \text{request} = \text{request}->\text{Next}; \\
        \text{KeReleaseSpinLock}(); \\
        \text{nPackets}++; \\
    \}\} \text{while}(\text{nPackets}! = \text{nPacketsOld}); \\
\text{KeReleaseSpinLock}();
\]
do {
    KeAcquireSpinLock();

    nPacketsOld = nPackets;

    if(request){
        request = request->Next;
        KeReleaseSpinLock();
        nPackets++;
    }
} while(nPackets!=nPacketsOld);

KeReleaseSpinLock();
Abstraction (via Boolean program)

```plaintext
do {
    KeAcquireSpinLock();
    nPacketsOld = nPackets;
    if (request) {
        request = request->Next;
        KeReleaseSpinLock();
        nPackets++;
    }
} while (nPackets != nPacketsOld);
KeReleaseSpinLock();
```
Precise API Usage Rules (SLIC)
Static Driver Verifier - Finding Driver Bugs at Compile-Time

Static Driver Verifier (SDV) is a compile-time tool that explores code paths in a device driver by symbolically executing the source code. SDV is a unit-testing tool for Microsoft® Windows® device drivers based on Windows Driver Model (WDM) and Windows Driver Foundation (WDF).

SDV places a driver in a hostile environment and systematically tests all code paths by looking for violations of WDM usage rules. The symbolic execution makes very few assumptions about the state of the operating system or the initial state of the driver, so it can exercise situations that are difficult to exercise by traditional testing.

The set of rules packaged with SDV define how device drivers should use the WDM API. The categories of rules tested include the following.

<table>
<thead>
<tr>
<th>Category</th>
<th>Rules tested for ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRP</td>
<td>Functions that use of I/O request packets</td>
</tr>
<tr>
<td>IRQL</td>
<td>Functions that use interrupt request levels</td>
</tr>
<tr>
<td>PnP</td>
<td>Plug and Play functions</td>
</tr>
<tr>
<td>PM</td>
<td>Power management</td>
</tr>
<tr>
<td>WMI</td>
<td>Functions using Windows Management Instrumentation</td>
</tr>
<tr>
<td>Sync</td>
<td>Synchronization related to spin locks, semaphores, timers, mutexes, and other methods of access control</td>
</tr>
<tr>
<td>Other</td>
<td>Functions that are not fully described by any of the other categories</td>
</tr>
</tbody>
</table>

Note: SDV is distributed as part of the WDF Beta program. To sign up for the WDF Beta program, visit the following URL: [http://www.microsoft.com/whdc/devtools/tools/SDV.mspx](http://www.microsoft.com/whdc/devtools/tools/SDV.mspx)
Part III: Comparison

- Informal
- Formal
Informal

Abstract

- domain-specific
- large manual

Comparison

Interpretation

efficient
Informal

Abstract

- domain-specific
- large manual

Comparison

Interpretation

Efficient
Informal

Predicate

- domain-specific

- automatic

Comparison

Abstraction
Formaly Comparing the Two Approaches

- **WAIL**  
  - widening + abstract interpretation over infinite lattice

- **FAIR**  
  - finite abstraction + iterative refinement
Abstraction/Refinement

- [Cousot-Cousot, PLILP’92]  
  - widening + abstract interpretation with infinite lattices (WAIL) is more powerful than a (single) finite abstraction

- [Namjoshi/Kurshan, CAV’00]  
  - if there is a finite (bi-)simulation quotient then WAIL with no widening will terminate [and therefore so will FAIR]

- [Ball-Podelski-Rajamani, TACAS’02]  
  - finite abstractions plus iterative refinement (FAIR) is more powerful than WAIL
Guarded Command Language

- Variables $X = \{x_1, \ldots, x_n\}$

- Guarded command $c$
  - $g \land x_1 = e_1 \land \ldots \land x_n = e_n$

- Program is a set of guarded commands
  - each command is deterministic
  - set of commands may be non-deterministic
Symbolic Representation of States

\[ \varphi \equiv \bigvee_{i \in I} \bigwedge_{j \in J(i)} \varphi_{ij} \]

\( \varphi_{ij} : \) atomic formula such as \((x < 5)\)

\[ \varphi' \leq \varphi \equiv \varphi' \implies \varphi \]
pre of

\[ c \equiv g \land x_1' = e_1 \land \ldots \land x_n' = e_n \]

• \( \text{pre}_c(\varphi) \equiv g \land \varphi[e_1,\ldots,e_n/x_1,\ldots,x_n] \)

• \( \text{pre}(\varphi) \equiv \bigvee_{c \in C} \text{pre}_c(\varphi) \)
Safe Backward Invariants

∀ ψ is a safe backward invariant if

- unsafe ⇒ ψ
- pre(ψ) ⇒ ψ
- ψ ⇒ noninit
Predicate Abstraction

– A set $P$ of predicates over a program’s state space defines an abstraction of the program
  • $P = \{ (a=1), (b=1), (a>0) \}$
  • Uninterpreted atoms $[a=1][b=1][a>0]$

– If $P$ has $n$ predicates, the abstract domain contains exactly $2^n$ elements
  • an abstract state = conjunction ($\wedge$) of atoms
  • a set of abstract states = disjunction ($\vee$) of abstract states
Free Lattice of DNF over \{a, b\}

\[
\begin{align*}
&\text{Logical Implication} \\
&\text{true} \\
&a \lor b \\
&a \lor (a \land b) & b \lor (a \land b) \\
&a \lor (a \land b) & b \lor (a \land b) \\
&a \lor (a \land b) \\
(a \land b) \\
\lor \quad \text{false}
\end{align*}
\]
$$\text{pre}^#_P \equiv \alpha_P \text{ pre } \gamma$$

$$\forall \gamma \quad \equiv \text{the identity function}$$

$$\forall \alpha_P(\varphi) \equiv \text{the least } \varphi' \text{ such that } \varphi \leq \gamma \varphi'$$

- **Example:**

  - $P$ = \{ (x<2), (x<3), (x=0) \}  
  - $\alpha_P( x=1 )$ = (x<2) \land (x<3)
\[ n := 0; \varphi := \text{unsafe} \]

\textbf{loop}

\[ P_n := \text{atoms}(\varphi) \]

\textit{construct pre}[^n], as defined by \( P_n \)

\[ \psi := \text{lfp}(\text{pre}[^n], \text{unsafe}) \]

\textbf{if} (\( \psi \leq \text{noninit} \)) \textbf{then}

\textbf{return} “success”

\[ \varphi := \varphi \lor \text{pre}(\varphi); \]

\[ n := n + 1; \]

\textbf{forever}
Widening

- $\text{widen}(\varphi) = \varphi'$ such that $\varphi \leq \varphi'$

- We consider widening that simply drops terms from some conjuncts

$$\text{widen} \left( \bigvee_{i \in I} \bigwedge_{j \in J(i)} \varphi_{ij} \right) = \bigvee_{i \in I} \bigwedge_{j \in J'(i)} \varphi_{ij} \quad \text{where } J'(i) \subseteq J(i)$$

- Results can be extended to other classes of widenings
Interval Widening, Revisited

\[ [l_0, u_0] \uparrow [\infty, \infty] \]

\[ l_0 \leq x \land x \leq u_0 \]
n := 0; φ := unsafe; old := false;
loop
  if (φ ≤ old) then
    if (φ ≤ noninit) then
      return “success”
    else
      return “Don’t know”
  else
    old := φ
    i := guess provided by oracle
    φ := widen(i, φ ∨ pre(φ) )
    n := n+1
  forever
FAIR

n := 0; \( \varphi \) := unsafe
loop
\( P_n \) := atoms(\( \varphi \))
construct \( \text{pre}_n \), as defined by \( P_n \)
\( \psi \) := lfp(\( \text{pre}_n \), unsafe)
if \( \psi \leq \text{noninit} \) then
    return "success"
else
    \( \varphi \) := \( \varphi \lor \text{pre}(\varphi) \);
    \( n := n + 1 \);
forever

WAIL

n:= 0; \( \varphi \) := unsafe; old := false;
loop
    if \( \varphi \leq \text{old} \) then
        if \( \varphi \leq \text{noninit} \) then
            return "success"
        else
            return "Don't know"
    else
        old := \( \varphi \);
        i := guess provided by oracle
        \( \varphi \) := widen(i, \( \varphi \lor \text{pre}(\varphi) \) )
        \( n := n+1 \);
forever
Theorem. For any program $P$, if WAIL terminates with success for some sequence of widening choices, then FAIR will terminate with success as well.

- **Lemma 1**: If a safe invariant $\psi$ can be expressed in terms of predicates in $P$ then $\text{lfp}(\text{pre}_P^#, \text{unsafe})$ is a safe invariant.

- **Lemma 2**: For any guarded command $c$,
  \[
  \text{pre}_c(\varphi \lor \varphi') = \text{pre}_c(\varphi) \lor \text{pre}_c(\varphi') \\
  \text{pre}_c(\varphi \land \varphi') = \text{pre}_c(\varphi) \land \text{pre}_c(\varphi')
  \]

- **Corollary**: For any guarded command $c$,
  \[
  \text{atoms}(\text{pre}_c(\varphi \lor \varphi')) = \text{atoms}(\text{pre}_c(\varphi)) \cup \text{atoms}(\text{pre}_c(\varphi')) \\
  \text{atoms}(\text{pre}_c(\varphi \land \varphi')) = \text{atoms}(\text{pre}_c(\varphi)) \cup \text{atoms}(\text{pre}_c(\varphi'))
  \]
Proof of Theorem

\[ \varphi_0 = \text{unsafe} \quad \quad \varphi'_0 = \text{unsafe} \]

\[ \varphi_{n+1} = \varphi_n \lor \text{pre}(\varphi_n) \quad \varphi'_{n+1} = \text{widen}(\varphi'_n \lor \text{pre}(\varphi'_n)) \]

for all \( i \), \( \text{atoms}(\varphi_i) \supseteq \text{atoms}(\varphi'_i) \)

by induction on \( i \) and Lemma 2

if \( \varphi'_i \) is a safe inv. then

by Lemma 1 and above result

\[ \text{lfp}(F_{\text{atoms}(\varphi_i)}, \text{start}) \] is a safe inv.
Summary

• Predicate abstraction + refinement and widening can be formally related to each other

• Predicate abstraction + refinement = widening with “optimal” guidance
What We Did

• Part I: Abstract Interpretation
  – [Cousot & Cousot, POPL’77]
  – Manual abstraction and refinement
  – ASTRÉE Analyzer

• Part II: Predicate Abstraction
  – [Graf & Saïdi, CAV ’97]
  – Automated abstraction and refinement
  – SLAM and Static Driver Verifier

• Part III: Comparing Approaches
Searching for Solutions

• Once upon a time, only a human could play a great game of chess…
  – … but then smart brute force won the day

• Once upon a time, only a human could design a great abstraction…