On the Design + Implementation of Static Analysis Tools

Thomas Ball

Microsoft Research

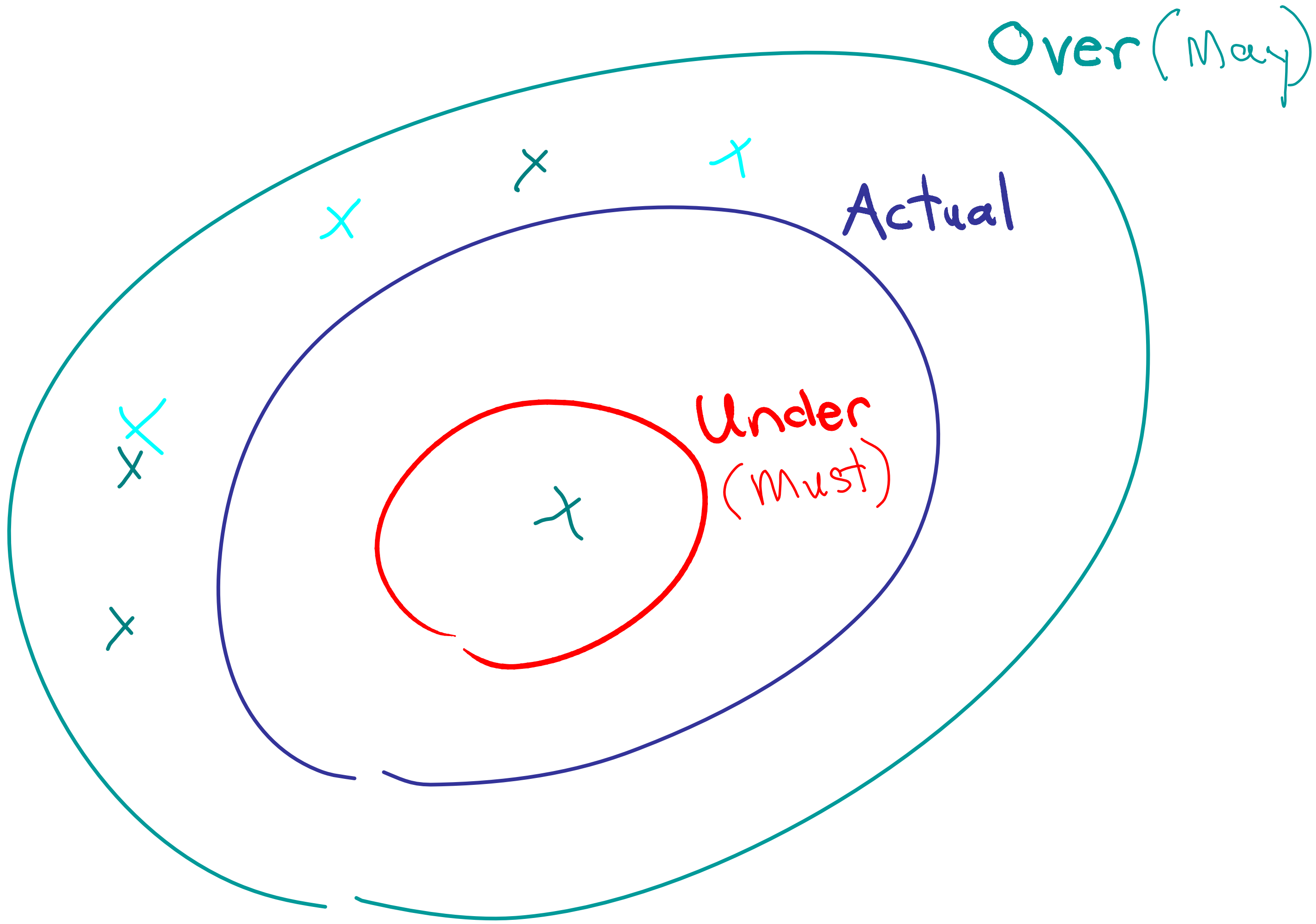
Static Analysis

the algorithmic discovery
of properties of a program
by inspection of its source text

Manna, Pnueli "algorithmic verification"

Properties

Defect Detection / Verification



Tool

webster.com

2a.

something used in performing an operation or necessary in the practice of a vocation or profession

A New Generation of Software Tools

- Windows device drivers
  - Static Driver Verifier http://www.microsoft.com/whdc/devtools/tools/sdv.mspx
  - part of Windows Vista DDK

A New Generation of Software Tools

- Buffer overflow checking for C/C++
  - SAL + PREfast http://en.wikipedia.org/wiki/Microsoft_Platform_SDK
  - part of Windows Vista SDK + VS
- Windows device drivers
  - Static Driver Verifier http://www.microsoft.com/whdc/devtools/tools/sdv.mspx
  - part of Windows Vista DDK

A New Generation of Software Tools

- .NET (managed) code
  - FxCop http://www.gotdotnet.com/team/fxcop/ (Visual Studio + SDK)
  - Spec# and Boogie static checker
    http://research.microsoft.com/specsharp/
- Buffer overflow checking for C/C++
  - SAL + PREfast http://en.wikipedia.org/wiki/Microsoft_Platform_SDK
  - part of Windows Vista SDK + VS
- Windows device drivers
  - Static Driver Verifier http://www.microsoft.com/whdc/devtools/tools/sdv.mspx
  - part of Windows Vista DDK

Types + Separate Compilation

```
void foo(int *arr, int  len);
```

Modular Reasoning

```
void foo(__ecount(len) int *arr, int  len);
```

arr points to a buffer with
at least len ints

Modular Reasoning

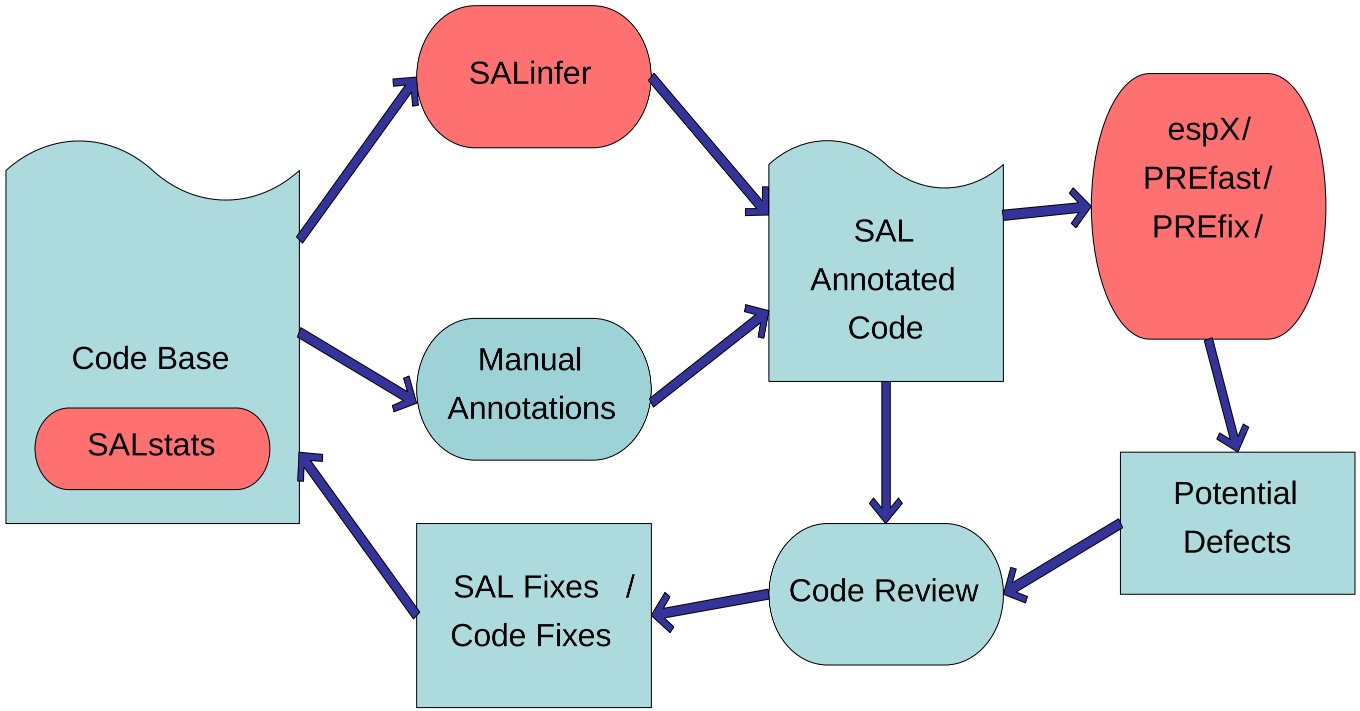
```
int a[20];
foo(a,21);
```

```
void foo(__ecount(len) int *arr, int  len);
```

Modular Reasoning

```
void foo(__ecount(len) int *arr, int  len);


for(int i=0;i<=len;i++) {
      arr[i] = 0;
 }
```

SAL Ecosystem



Windows Vista

- mandate: Annotate 100,000 mutable buffers
- developers annotated 500,000+ parameters
- developers fixed 20,000+ bugs

Office 2007

- developers fixed 6,500+ bugs

FM in the last decade at MSR

- Mathematical specification:
  - http://research.microsoft.com/users/lamport/tla/
- Project **X**
  - http://research.microsoft.com/projects/**X**/
- Model-based specification and testing: **X→specexplorer**
- Languages: **X**∈{**clrgen, comega**, **fsharp**}
- Modular verification of programs with contracts:
  - **SAL** and buffer overflows (Visual Studio)
  - **X**∈{**specsharp**, **havoc**}
- Model checking (sequential programs): **X→slam**
- Model checking (concurrent programs): **X**∈{**zing**, **chess**}
- Security for web services: **X→samoa**
- Test generation: **X→pex**
- Automated theorem proving: **X→z3**

Microsoft Research

A New Generation of Software Tools

- outside MS -

- ASTREE (C, verify avionics code)
- FindBugs (Java, bug finder)
- Saturn (C, null deref bug finder)
- Calysto "
- ... many examples of defect detection

A New Generation of Software Tools

- Static analysis
- Sequential
- Safety
- Contracts

- Widely deployed + effective
- Based on fundamental techniques

Why Static Analysis Tools?

Worse is Better

http://en.wikipedia.org/wiki/Worse_is_Better

Worse is better, also called the **New Jersey style**, is the name of a computer software design approach (or design philosophy) in which simplicity of both interface and implementation is more important than any other system attribute (including correctness, consistency, and completeness).

Why C won out over Lisp

Why dynamic languages won the Web

Defects = $$$

http://en.wikipedia.org/wiki/Robert_Tappan_Morris

defect

exploit

Control computer

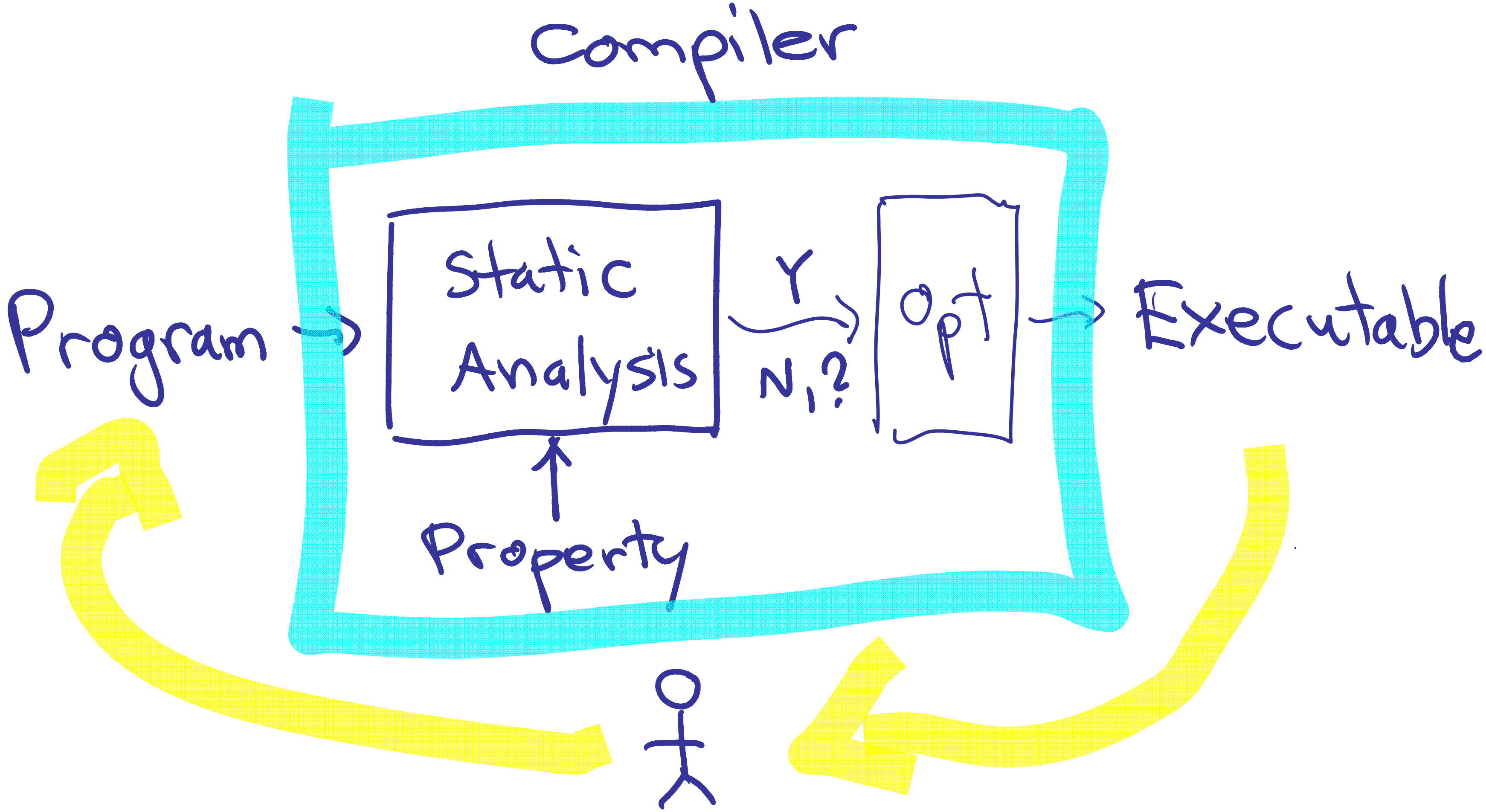
botNet

private info

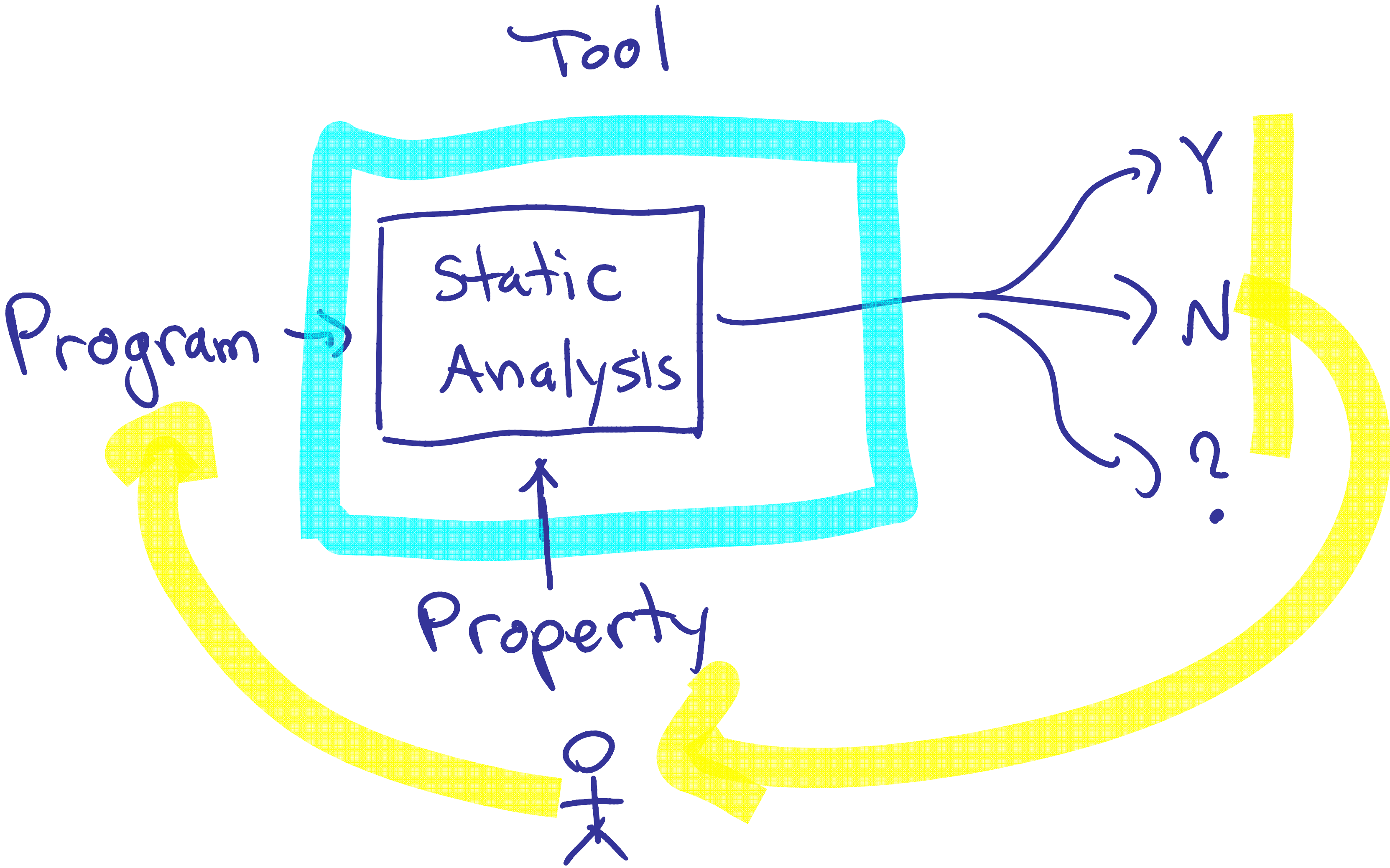
...

How do I design and implement a static analysis tool chain to help people effectively address a software reliability problem?

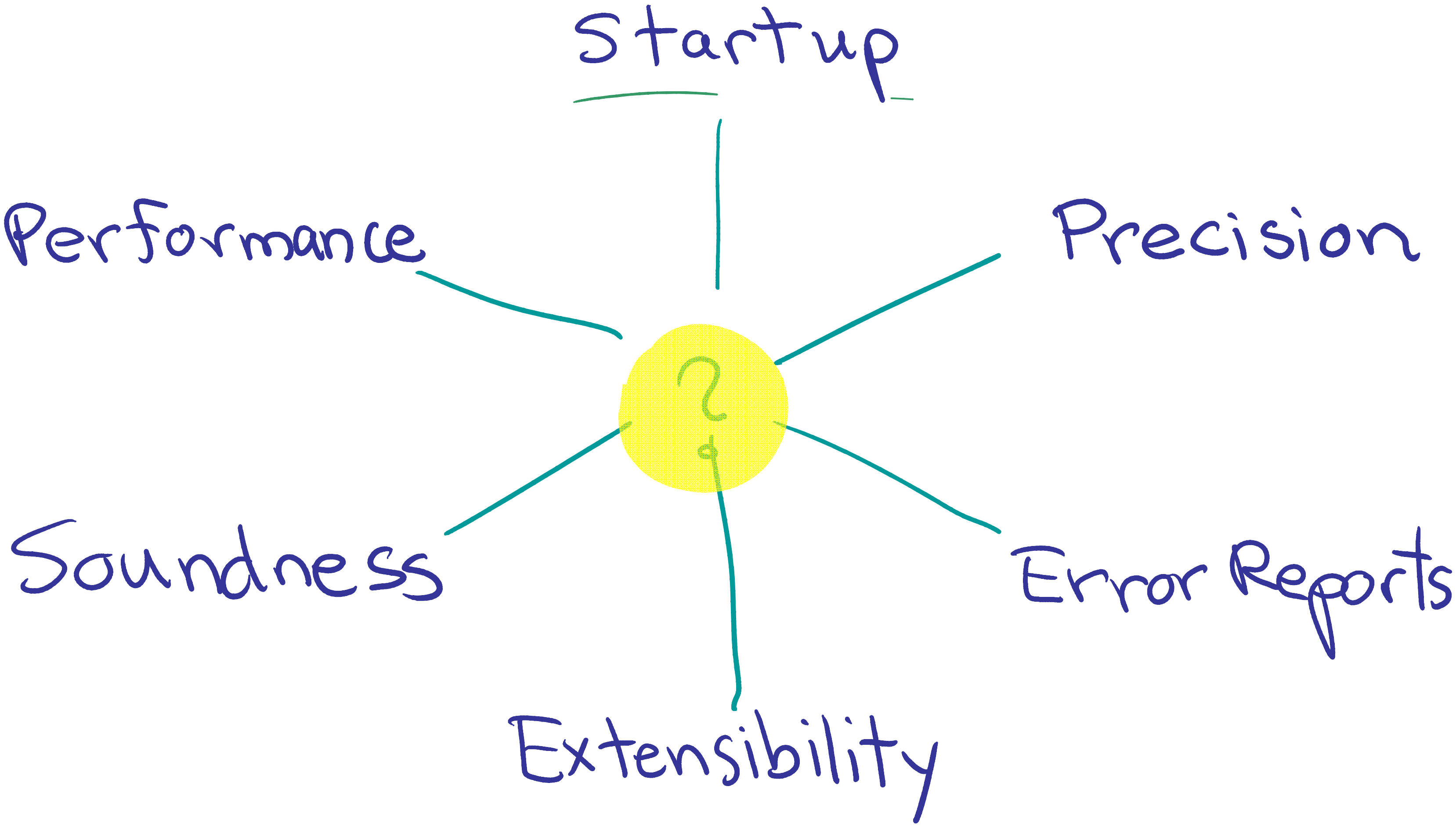
The Invisibility of Static Analysis



The Visibility of Static Analysis

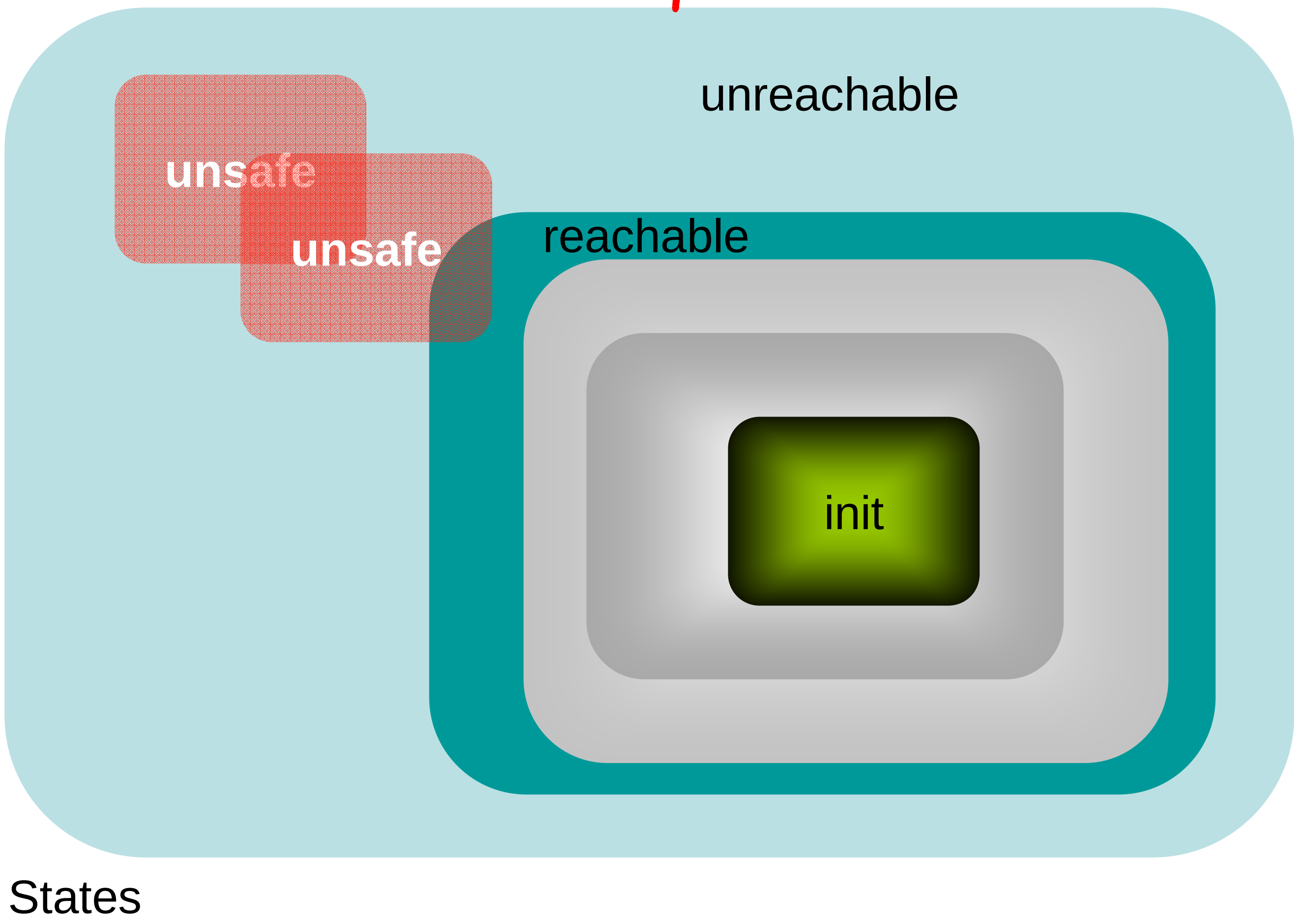


Tradeoffs



predictability

Reachability (Safety)



Program Analysis Problems

- unboundedness
- huge search space
- invariant inference
- frame problem
  - pointers + procedures
- efficiency vs. precision

Fundamentals

- Finite state machines + pushdown systems
- Modular verification via types
- Abstract interpretation
- Abstraction refinement
- Hoare logic and axiomatic reasoning
- Scalable interprocedural analysis
- Symbolic analysis engines

Overview

1. Boolean programs
   complexity
2. Symbolic reachability
   BDDs, SAT, interpolants
3. Interprocedural reachability
   pushdown systems
4. Mystery lecture

Static

analysis of

Sequential

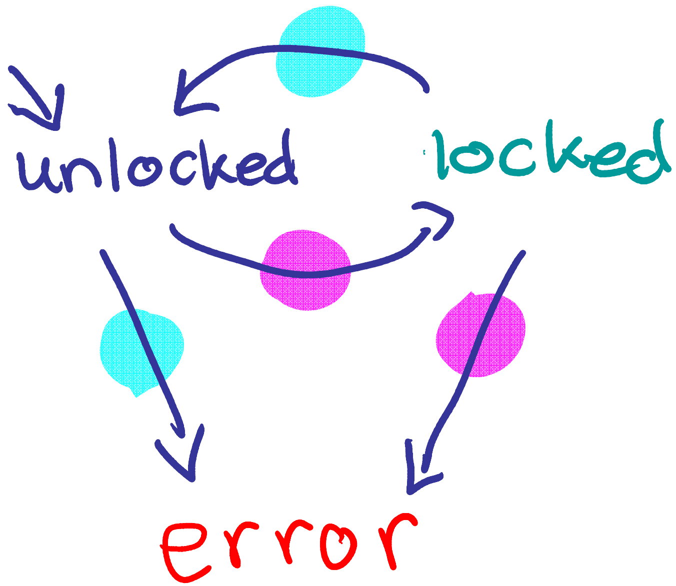
programs on

Safety

properties

Boolean Programs + Complexity

State Machine for Locking



Locking Rule in SLIC

```
state {
  enum {unlocked,locked}
    s := unlocked;
}

KeAcquireSpinLock.entry {
  if (s=locked) error;
  else s := locked;
}

KeReleaseSpinLock.entry {
  if (s=unlocked) error;
  else s := unlocked;
}
```

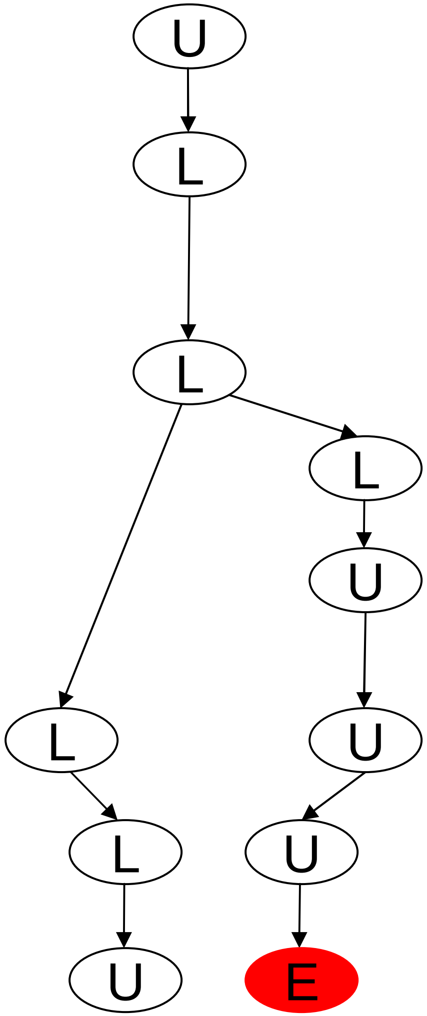
Example

Does this code obey the locking rule?

```
do {
   KeAcquireSpinLock();

   nPacketsOld := nPackets;

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

KeReleaseSpinLock();
```

Example

Model checking
boolean program

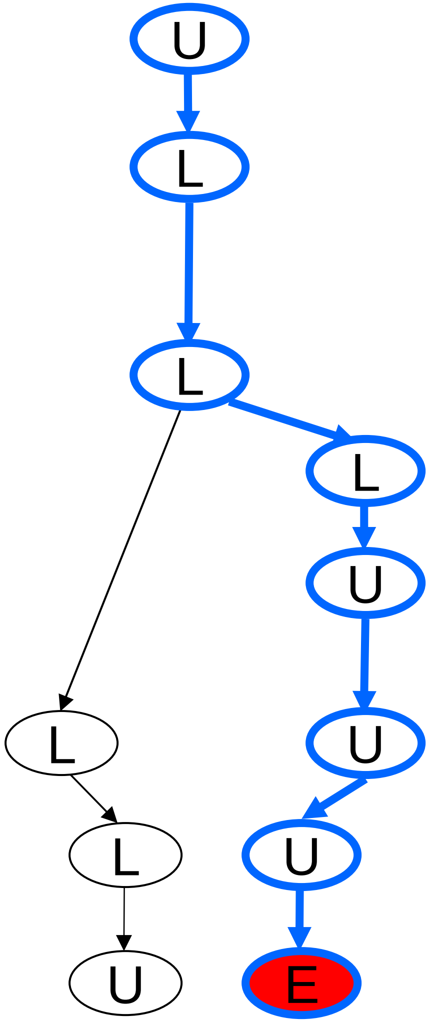



```
do {
   KeAcquireSpinLock();

   if(*){

      KeReleaseSpinLock();

   }
} while (*);

KeReleaseSpinLock();
```

Example

b : (nPacketsOld = nPackets)

Is error path feasible in C program?

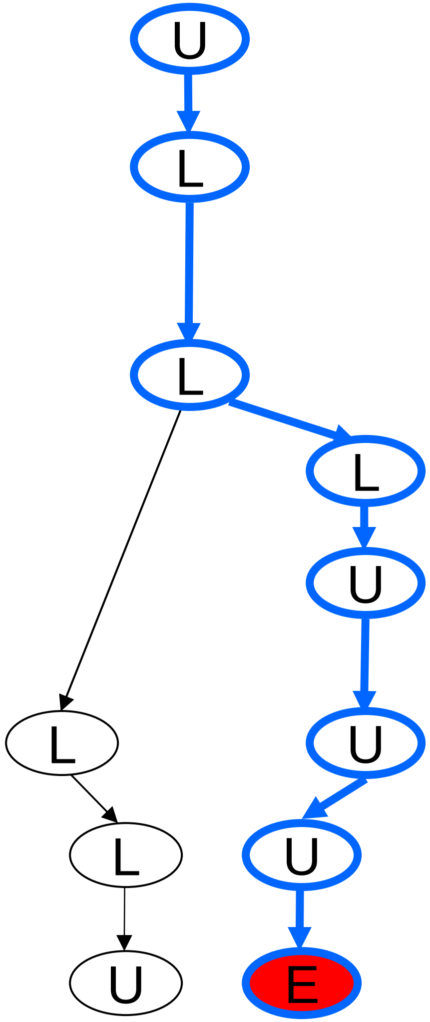



```
do {
   KeAcquireSpinLock();

   nPacketsOld := nPackets;

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

KeReleaseSpinLock();
```

Example

b : (nPacketsOld = nPackets)

Add new predicate to boolean program

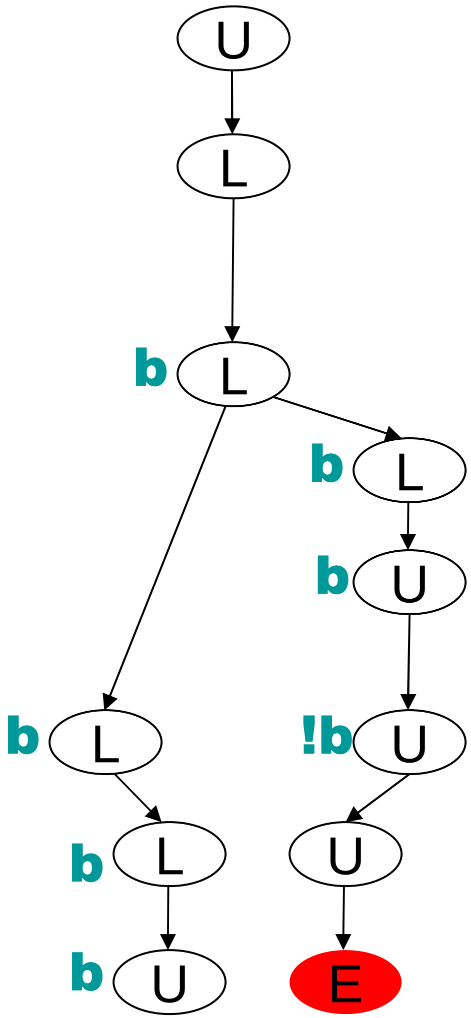



```
do {
   KeAcquireSpinLock();

   nPacketsOld := nPackets; b := true;

   if(request){
      request := request->Next;
      KeReleaseSpinLock();
      nPackets++; b := b ? false : *;
   }
} while (nPackets != nPacketsOld);  !b

KeReleaseSpinLock();
```

Example

b : (nPacketsOld = nPackets)

Model checking
refined
boolean program

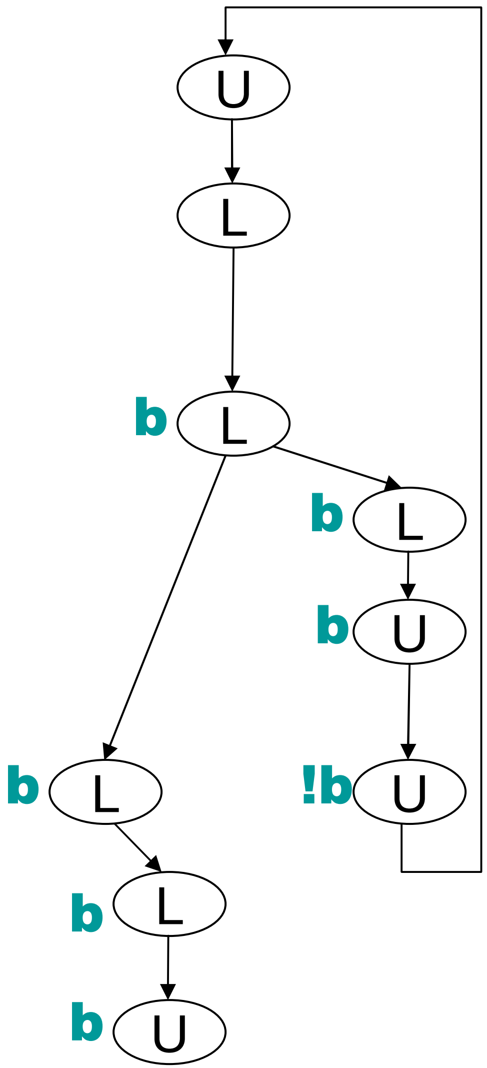



```
do {
   KeAcquireSpinLock();

   b := true;

   if(*){

      KeReleaseSpinLock();
      b := b ? false : *;
   }
} while ( !b );

KeReleaseSpinLock();
```

Example

b : (nPacketsOld = nPackets)

Model checking
refined
boolean program

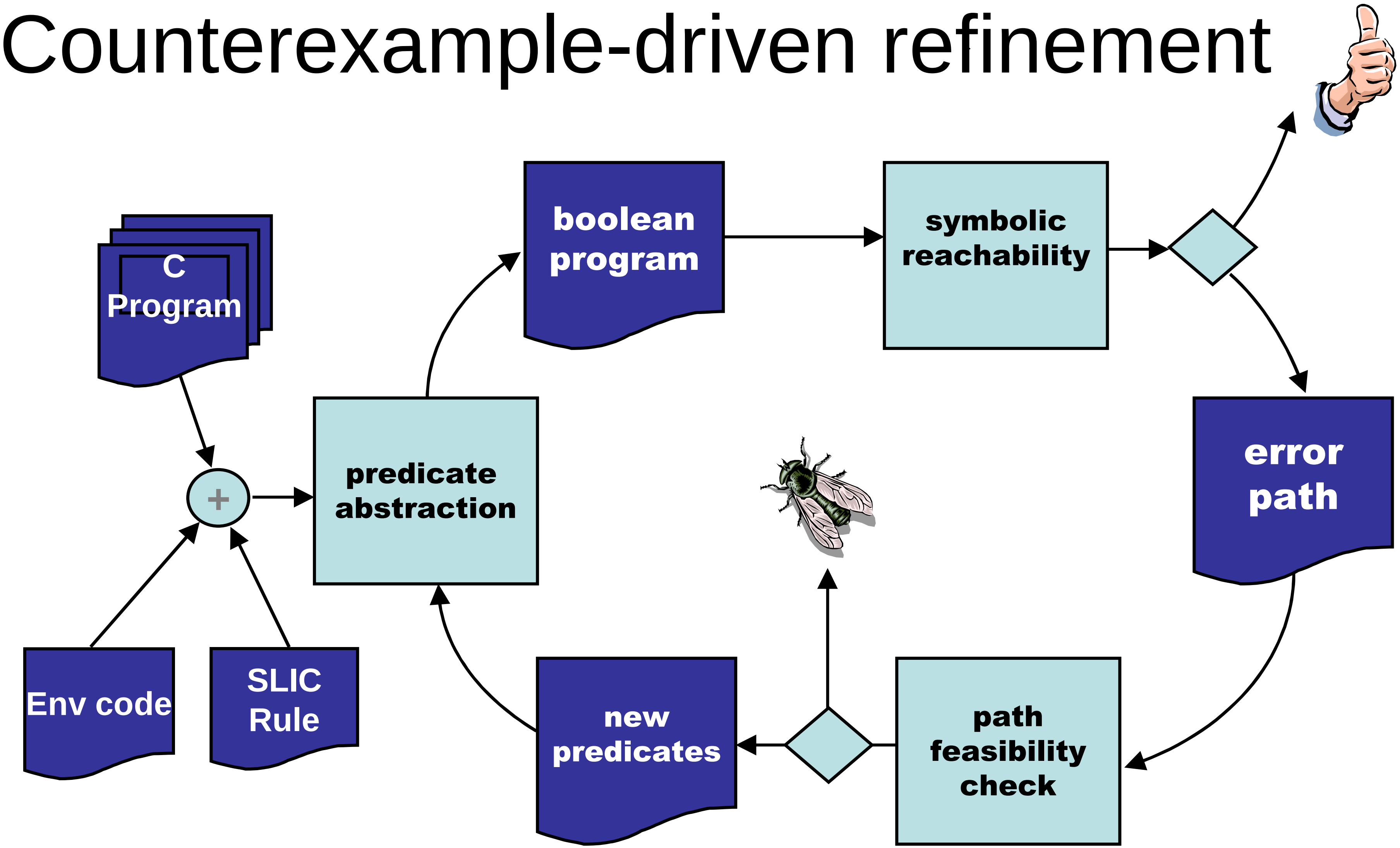



```
do {
   KeAcquireSpinLock();

   b := true;

   if(*){

      KeReleaseSpinLock();
      b := b ? false : *;
   }
} while ( !b );

KeReleaseSpinLock();
```

Inferred Invariant

"The lock is held at the end of the loop if and only if nPackets = nPacketsOld"

Counterexample-driven refinement



Kurshan '93, Ball/Rajamani '00, Clarke etal '00

Quote

- Ed Clarke
  - *“The SLAM model checker developed at Microsoft Research for finding errors in Windows device drivers is probably the most successful software model checker. However,…”*
  - July’08 CACM, page 111

Propositional Logic

$$\bar{x}_1 \wedge \bar{x}_2 \wedge \bar{x}_3 \vee x_2 \wedge x_1 \vee x_2 \wedge \bar{x}_3$$

Boolean function

$f(x_1, x_2, x_3)$

Relation

$R = \{\langle 000 \rangle, \ldots\}$

State predicate

$P(s)$

Binary Relation

$R = \{\langle 0, 00 \rangle, \ldots\}$

$$
\begin{aligned}
E \to\ & 0 \mid 1 \\
\mid\ & x \\
\mid\ & \overline{E} \\
\mid\ & E \lor E \\
\mid\ & E \land E
\end{aligned}
$$

Boolean Programs

$$Prog \to \text{var } id^* \; P^*$$

$$P \to f(id^*) \{ \; \text{var } id^* \; S \; \}$$

$$
\begin{aligned}
S \to\ & \text{error} \\
|\ & \text{assume}(E) \\
|\ & x := E \\
|\ & S ; S \\
|\ & S \,\square\, S \\
|\ & \text{while}(E) \{ S \} \\
|\ & f(E_1, \ldots E_k)
\end{aligned}
$$

Exercise

Define assert(E)

and havoc x

in terms of given language

Complexity

$$P \subseteq NP \subseteq PSPACE$$

Acyclic Single-Procedure BP

$$L \to x \mid \bar{x}$$

$$S \to \text{error} \mid \text{assume}(L) \mid S \,;\, S \mid S \,\square\, S$$

3SAT Reduction

$$\exists x_1 \exists x_2 \exists x_3 \exists x_4 : (x_1 \vee \overline{x_3} \vee x_4) \wedge (\overline{x_2} \vee x_3 \vee \overline{x_4})$$

Strongest Postcondition

$$
\begin{aligned}
S \to\ & \mathrm{error} \\
\mid\ & \mathrm{assume}(E) \\
\mid\ & S \,;\, S \\
\mid\ & S \,\square\, S
\end{aligned}
$$

Exercise

Show that single-proc. loop-free BP has equivalent "passive" form

Active

$$
\begin{aligned}
S \to\ & x := E \\
\mid\ & \text{assume}(E) \\
\mid\ & S ; S \\
\mid\ & S \,\square\, S
\end{aligned}
$$

Passive

$$
\begin{aligned}
S \to\ & \text{assume}(E) \\
\mid\ & S ; S \\
\mid\ & S \,\square\, S
\end{aligned}
$$

While Loops

$$S \rightarrow \text{error} \mid \text{assume}(E) \mid x := E$$
$$\mid S ; S \quad \mid S \,\square\, S$$
$$\mid \text{while}(E) \{ S \}$$

Quantified Boolean Formula

$$\exists x_1 \forall x_2 \exists x_3 \forall x_4 : (x_1 \vee \overline{x_3} \vee x_4) \wedge (\overline{x_2} \vee x_3 \vee \overline{x_4})$$

PSPACE-complete

QBF → BP Reduction

$$\exists x_1 \forall y_1 \ldots \exists x_k \forall y_k : \varphi(x_1, y_1, \ldots, x_k, y_k)$$

Exercise

Show reduction using **1** while loop and polynomially bounded # of variables

Procedures + Procedure Calls

$$\exists x_1 \forall y_1 \ldots \exists x_k \forall y_k : \varphi(x_1, y_1, \ldots, x_k, y_k)$$

Exercise

Show reachability in BP is in PSPACE

Loop Invariants (Checking)

(A)

```
while (E) { S }
   I, T
```

(A) correct

$\Longleftrightarrow$

(B) correct

(B)

```
assert I;
havoc T;
assume I;
if (E) {
    S;
    assert I;
    assume false;
}
```

Loop Invariants (Inferring)

Single proc. BP correct
(no error stmt. reachable)

$\Longleftrightarrow$

$\exists$ loop invariants $I_1, \ldots, I_k$
s.t. loop-free$(P, I_1, \ldots, I_k)$ correct

Circuit Complexity

What is the worst-case size of loop invariants and pre/post- conditions?

Exercise:

- demonstrate a program with a large loop inv.

Pre- and Post- Conditions

```
f ( ) ;
  pre, T, post
```

```
assert(pre);
havoc T;
assume(post);
```

Recursion + Reachability

Decidable!

Lecture 4 ...

Concurrency (in programs)

Undecidable

for two recursive Boolean programs running concurrently with single shared bit.

Ramalingam

,

Other Language Features

What features can be added to Boolean Programs and retain decidability?

Boolean Program Summary

assume
S ; S
S $\Box$ S

NP

while
foo()

PSPACE

inv. checking - NP

inv. inference - PSPACE

recursion - decidable (PSPACE)

concurrency - undecidable

Concurrency (in analysis)

NC = "Nick's Class"

Monotone Circuits

$out(p_a)$

$p_b = p_c \wedge p_d$

$p_e = p_f \vee p_g$

$p_i \to p_h$

$true \to p_j$

$false \to p_k$

Reachability

- Explicit
  - bit-state hashing — SPIN, CMC, ...
  - stateless — Verisoft, CHESS, ...
- Symbolic
  - BDDs — SMV, bebop, ...
  - SAT solvers — Calysto, Saturn, ...
  - SMT solvers — ?

Reachability and Invariants

Symbolic transition system

$$(\mathit{init}: 2^S, \quad T: 2^{S \times S}, \quad \mathit{safe}: 2^S)$$

Reachability and Invariants

Symbolic transition system

$$(\mathit{init}: 2^S, \quad T\ 2^{S \times S}, \quad \mathit{safe}: 2^S)$$

$R_0 = \mathit{init}$

$R_1 = R_0 \cup T(R_0)$

$\vdots$

$R_{k+1} = R_k \cup T(R_k)$

stop when $R_{k+1} = R_k$

+ check $R_k \subseteq \mathit{safe}$

Reachability and Invariants

Symbolic transition system

$$(\text{init}: 2^S, \quad T: 2^{S\times S}, \quad \text{safe}: 2^S)$$

$R_0 = \text{init}$

$R_1 = R_0 \cup T(R_0)$

$\vdots$

$R_{k+1} = R_k \cup T(R_k)$

stop when $R_{k+1} = R_k$

+ check $R_k \subseteq \text{safe}$

$Q : 2^S$ is a safe inductive invariant if

(1) $\text{init} \subseteq Q$

(2) $Q \subseteq \text{safe}$

(3) $T(Q) \subseteq Q$

Symbolic Reachability

(1) with Binary Decision Diagrams

(2) with SAT solvers

$$f(x_1, x_2, x_3) = \overline{x}_1 \wedge \overline{x}_2 \wedge \overline{x}_3 \vee x_2 \wedge x_1 \vee x_2 \wedge \overline{x}_3$$

x_1	x_2	x_3	f
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Reduced, Ordered Binary Decision Diagrams

- canonical form
  - SAT, UNSAT, VALID, =
- quantifier elimination

  $\exists x\ \varphi$
- efficient Boolean operations

  $\vee, \wedge, \neg, \Rightarrow$
- substitution: $\varphi[t/x]$

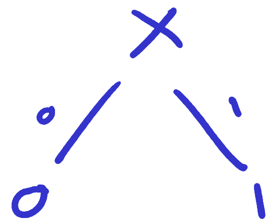
Variable

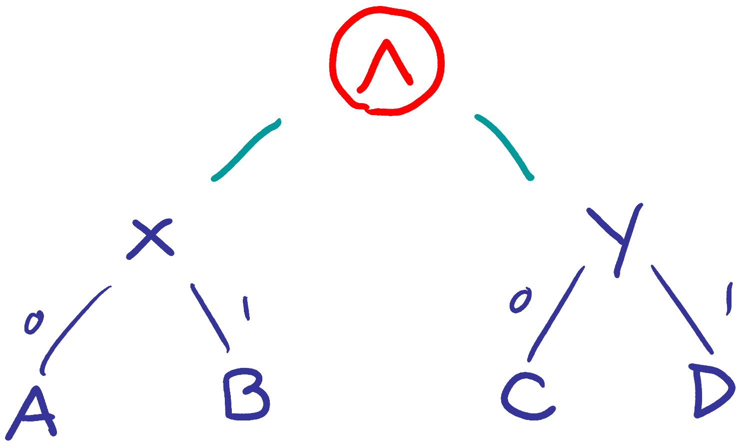
Order

Matters!!

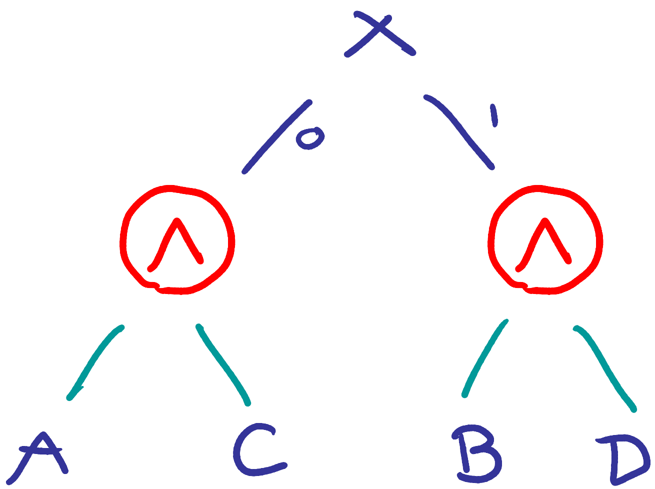
Hash Consing

BDDofVar(x) =

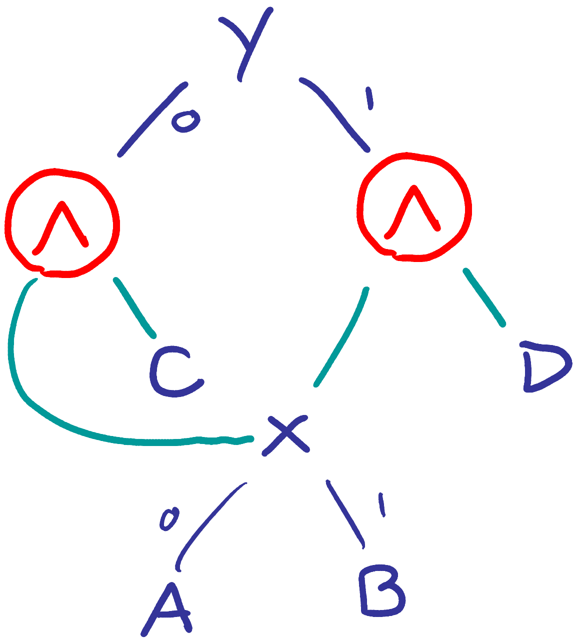




$x = y$



$y < x$ (WLOG)



Transition Relations with BDDs

$$T(\underbrace{old(x), old(y)}_{OLD_V}, \underbrace{x, y}_{V}): \ x = old(y) \wedge y = old(x)$$

$$Image(P, T): \exists OLD_V . P[OLD_V / V] \wedge T$$

$$P: \quad x = 1 \wedge y = 0$$

$$Image(P, T): \quad \exists old(x), old(y): \; old(x) = 1 \wedge old(y) = 0 \; \wedge \; x = old(y) \wedge y = old(x)$$

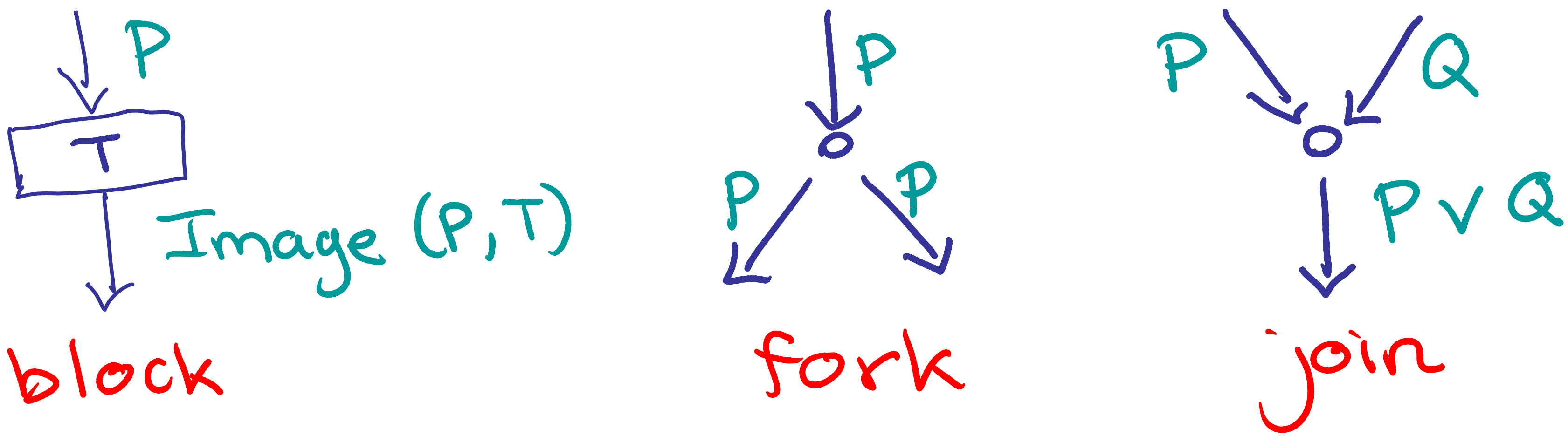
$$= \quad x = 0 \wedge y = 1$$

Strongest Postconditions w/ BDDs

$$SP(P, \text{assume}(E)) = P \wedge E$$

$$SP(P, x := E) = \exists t : P[t/x] \wedge (x = E[t/x])$$

Control Flow



Fixpoint Algorithm

```
forall edges e,  R(e) := 0;
R(entry) := 1;
WS := {entry};
while WS ≠ ∅  {
    remove e from WS;
    case tgt(e) of
        block (T, f):
            next := Image(R(e), T);
            if VALID(next => R(f)) {
                R(f) := next;
                add f to WS;
            }
    ...
}
```

Other Applications of BDDs

- pointer analysis
- context-sensitive analysis
- Datalog (bddbddb)
- ...

Modern SAT Solvers

DPLL   Unit Propagation

Backjump   Learning

Bounded model checking
+ reachability

Example

Stack

$\bar{1} \vee 2 \wedge \bar{3} \vee 4 \wedge \bar{5} \vee \bar{6} \wedge 6 \vee \bar{5} \vee \bar{2}$

$\emptyset$

$1 \quad 2 \quad 3 \quad 4 \quad 5 \quad \bar{6}$

Bounded Model Checking (BMC)

Transition system $(init, T, safe)$

$BMC(init, T, \overline{safe}, k)$

can we reach an unsafe state in at most k steps?

BMC(init, T, unsafe, k)

SAT formula

$init \wedge T_{0,1} \wedge \ldots \wedge T_{k-1,k} \wedge$

$(unsafe_0 \vee unsafe_1 \vee \ldots \vee unsafe_k)$

SAT => error

UNSAT => no error in k steps

$T^k(init) \Rightarrow safe$

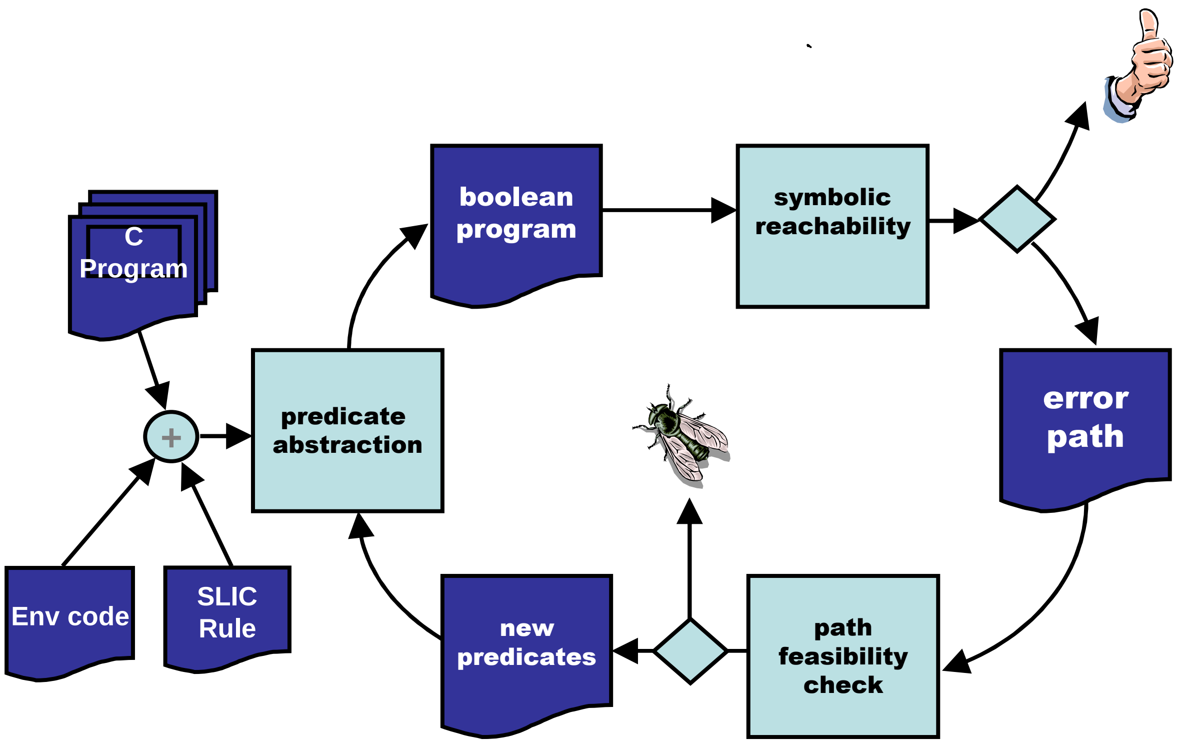
How to Prove Correctness

Diameter = smallest # of steps
from which all reachable states
are reachable

for $i = 0$ to Diameter(init, T) {
  if BMC(init, T, $\overline{\text{safe}}$, i) = SAT then
    return Error;
}
return Correct

Does This Seem Familiar?

Does This Seem Familiar?



Generalize!

```
main(init, T, safe) {
  if SAT(init ∧ ¬safe) then return Error;
  for k := 1 to Diameter(init, T) {
      // 0 ≤ i < k, T^i(init) => safe
      case Chk(init, T, safe, k) of
        Correct:   return Correct
        Error:     return Error
        kcorrect:  skip
      end
  }
  return Correct
}
```

Thanks, Rustan!

```
Chk(init, T, safe, k) {
  if BMC(init, T, ¬safe, k) = SAT then
    return Error;
  else
    return Abstract(init, T, safe, k)
}
```

safe to return kCorrect

```
Abstract(init, T, safe, k) {        k>0
    Q := init;
    while (true) {
      // init => Q  ∧ ∀i: 0≤i≤k, T^i(Q) => safe
       Q' := NextAbs(Q, T, safe, k);
      //  Q => Q'  ∧ ∃j: 1≤j≤k, T^j(Q) => Q'
      if Q = Q' then
         return Correct
      else if BMC(Q', T, ¬safe, k) = SAT then
         return kCorrect
      Q := Q'
    }
}
```

McMillan '03

Correctness

$\exists k > 0 \; \exists j, 1 \le j \le k:$

(1) $init \Rightarrow Q \quad \wedge$

(2) $\forall i, 0 \le i \le k, T^i(Q) \Rightarrow safe \quad \wedge$

(3) $T^j(Q) \Rightarrow Q$

---

(1) $init \subseteq Q$

(2) $Q \subseteq safe$

(3) $T(Q) \subseteq Q$

$\forall n, n \geq 0, T^n(init) \Rightarrow safe$

Goal: Find Suitable NextAbs

// $k > 0 \;\wedge$
// $init \Rightarrow Q \;\;\wedge\; \forall i: 0 \le i \le k,\; T^i(Q) \Rightarrow safe$

$Q' := NextAbs(Q, T, safe, k);$

// $Q \Rightarrow Q' \;\wedge\; \exists j: 1 \le j \le k,\; T^j(Q) \Rightarrow Q'$

Widening (Abstract Interpretation)

NextAbs(Q , T, safe, k) =

return $Q \vee T(Q)$

// $Q \Rightarrow Q' \wedge \exists j : 1 \leq j \leq k, T^j(Q) \Rightarrow Q'$

Interpolants

Craig '57

$\forall$ A, B s.t $A \Rightarrow B$

$\exists$ interpolant P s.t

(1) $A \Rightarrow P \Rightarrow B$

(2) $vars(P) \subseteq vars(A) \cap vars(B)$

Example:

$A = p \wedge q$

$B = q \vee \bar{r}$

$P = q$

NextAbs(Q , T, safe, k)

$$Q_0 \wedge T_{0,1} \wedge \ldots \wedge T_{k-1,k} \wedge$$
$$(unsafe_0 \vee unsafe_1 \vee \ldots \vee unsafe_k)$$

UNSAT iff $A \Rightarrow B$, where

$$A = Q_0 \wedge T_{0,1}$$

$$B = (T_{1,2} \ldots \wedge T_{k-1,k}) \Rightarrow (safe_1 \wedge \ldots \wedge safe_k)$$

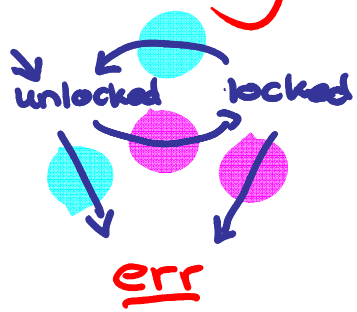
return $Q \vee \text{Interpolant}(A, B)[X/X_1]$

Summary

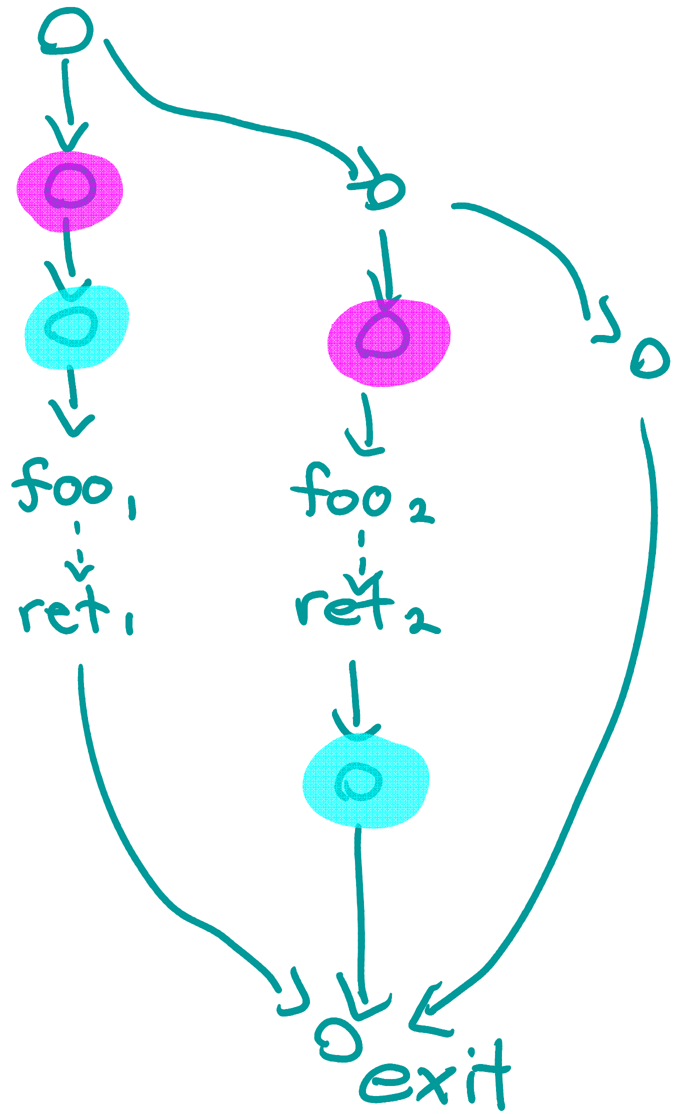
- Symbolic Reachability
- Precise solution with BDDs and SAT solvers
- Goal-directed abstraction with interpolants

Reachability with Procedures

Matching Calls / Returns

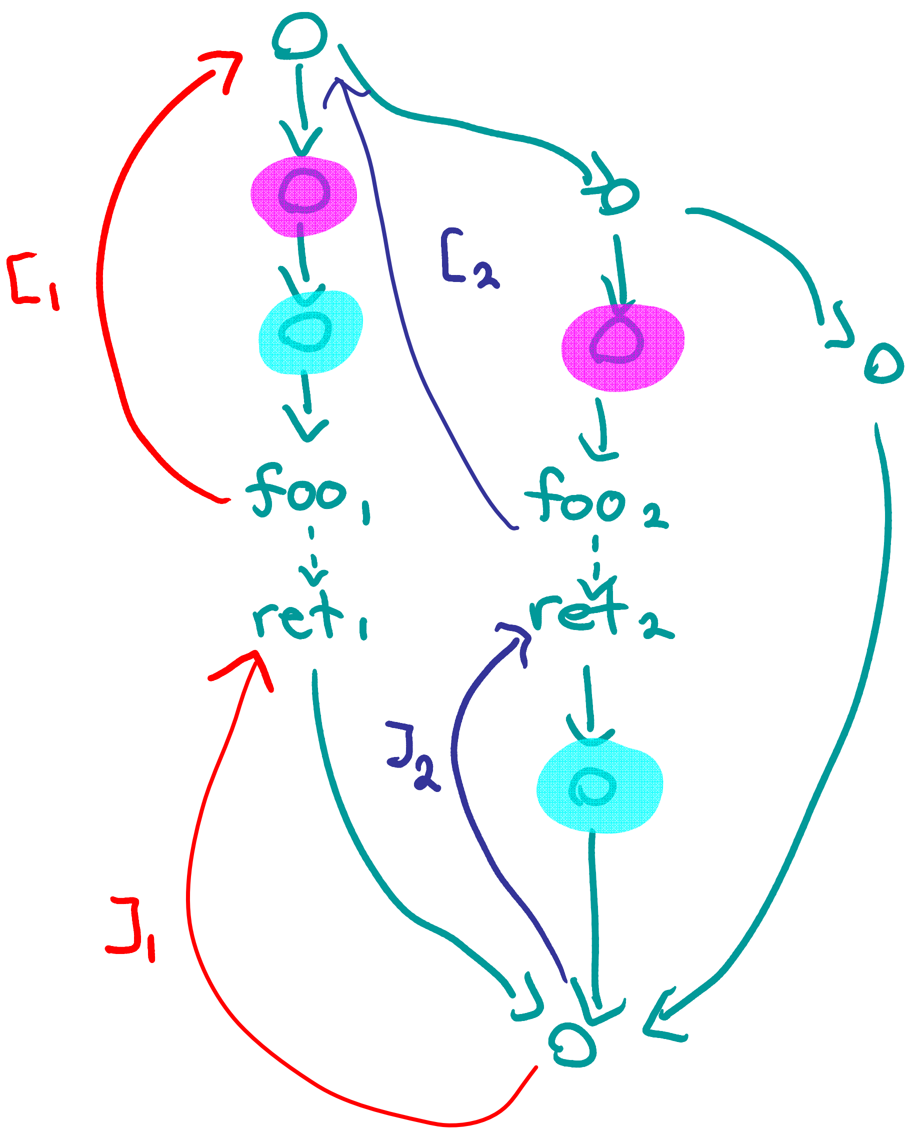



```
foo() {
    if (...) {
        Acq; Rel;
        foo();
    } else if (...) {
        Acq;
        foo();
        Rel;
    } else {
        ...
    }
}
```



Matching Calls / Returns

```
foo() {
    if(...) {
        Acq; Rel;
        foo();
    } else if (...) {
        Acq;
        foo();
        Rel;
    } else {
        ...
    }
}
```



Context-free Language Reachability

Directed rooted graph G with edges labelled from $\{\varepsilon, [_1,]_1, [_2,]_2\}$

Grammar S

$S \to S\,S$

$S \to [_1\, S\,]_1$

$S \to [_2\, S\,]_2$

$S \to \varepsilon$

Is there a path p from root to vertex v s.t labels of p form a string in S?

Pushdown System

$$PDS = (G, L, (g_0, l_0), \rightarrow)$$

G : finite set of global states

L : finite set of local/stack states

$(g_0, l_0) \in G \times L^*$, initial configuration

$\rightarrow \subseteq (G \times L) \times (G \times L^*)$, transitions

Encoding Boolean Programs

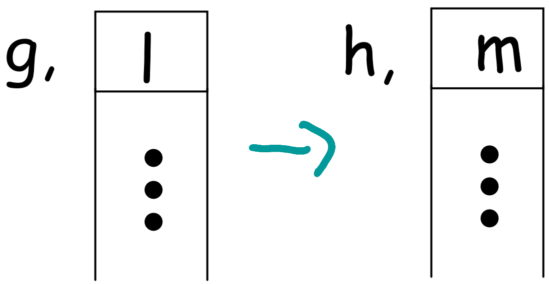
- $g \in G$ $\approx$ valuation of global variables

- $\ell \in L$ $\approx$ stack frame
  - valuation to local variables
  - PC of next instruction to execute

Three Transitions (I)

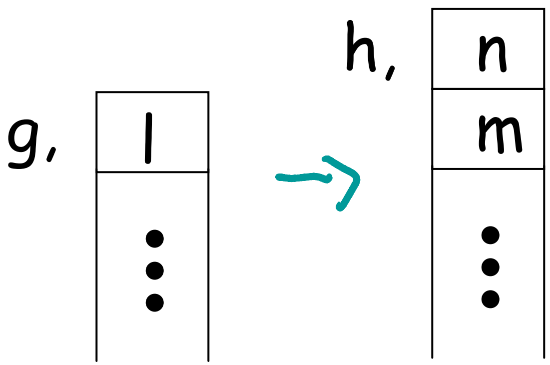
block

$(g, l) \to (h, m)$



call

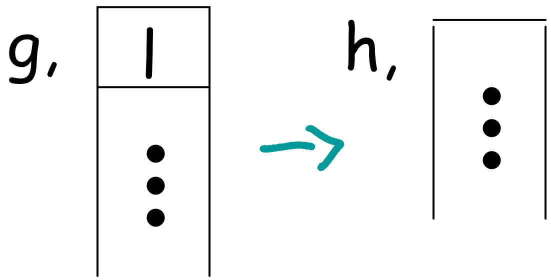
$(g, l) \to (h, nm)$



$g, h \in G \quad l, m, n \in L$

Three Transitions (II)

return $(g, \ell) \to (h, \varepsilon)$



Transitions

```
var g := T;
main() {
   L0 : flip();
   L1 : flip();
   L2 : assert(g);
}

flip() {
   L3: g := !g;
   L4: ;
}
```

$(T, L0) \rightarrow (T, L3.L1)$

$(F, L0) \rightarrow (F, L3.L1)$

$(T, L1) \rightarrow (T, L3.L2)$

$(F, L1) \rightarrow (F, L3.L2)$

$(T, L3) \rightarrow (F, L4)$

$(F, L3) \rightarrow (T, L4)$

$(F, L4) \rightarrow (F, \varepsilon)$

$(T, L4) \rightarrow (T, \varepsilon)$

Reachable Configurations

```
var g := T;
main() {
L0: flip();
L1: flip();
L2: assert(g);
}

flip() {
L3: g := !g;
L4: ;
}
```

(G, L*)

(T, L0)

(T, L3.L1)

(F, L4.L1)

(F, L1)

(F, L3 L2)

(T, L4.L2)

(T, L2)

(T, L0) -> (T, L3.L1)

(T, L3) -> (F, L4)

(F, L4) -> (F, ε)

(F, L1) -> (F, L3.L2)

(F, L3) -> (T, L4)

(T, L4) -> (T, ε)

Reachability in a PDS

Given PDS $= (G, L, (g_0, l_0), \rightarrow)$

and $g \in G$,

Does there exist a stack $w.l_0 \in L^*$ such that

$$(g_0, l_0) \rightarrow^* (g, w.l_0)$$

Naïve Algorithm

$R(g_0, l_0).$

$R(G_2, (NTOP.REST))$:-

$R(G1, TOP::REST), (G1, TOP) \rightarrow (G2, NTOP).$

Finite Reachability Relations

$Step(g, \ell, h, m)$

$Push(g, \ell, h, nm)$

$Pop(g, \ell, h)$

Initially

$Step(g_0, \ell_0, g_0, \ell_0)$

$Push = Pop = \phi$

$g, h \in G$

$\ell, m, n \in L$

Meaning of Step(g, ℓ, h, m)

$$\overbrace{\underbrace{(g_0, \ell_0) \to^* (\_, \_)}_{\text{calls/returns match}} \xrightarrow[\text{call}]{}}^{*} \underbrace{(g, \ell) \to^* (h, m)}_{\text{calls/returns match}}$$

$$S \to M \, [_i \, S$$

$$S \to M$$

$$M \to M M$$

$$M \to [_i \, M \,]_i$$

$$M \to \varepsilon$$

Matching Path

$(_C \;)_C \; (_D \; (_E \;)_E \;)_D$

$(_A \; (_B \; (_C \;)_C \; (_D \; (_E \;)_E \;)_D$

Step Rule

$$\frac{\text{Step}(g, \ell, h, m) \quad (h, m) \rightarrow (h', m')}{\text{Step}(g, \ell, h', m')}$$

or

Step(G, L, H', M') :-
    Step(G, L, H, M), (H, M) -> (H', M').

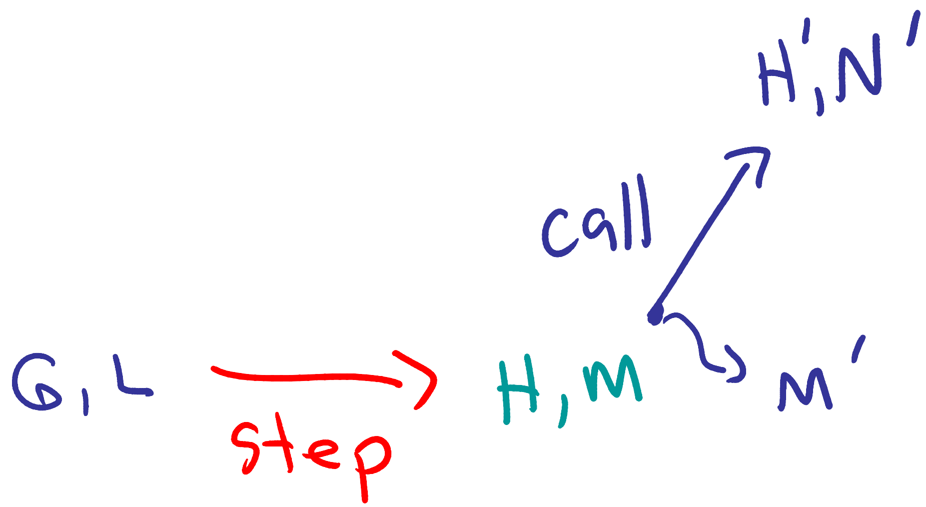
(Basic transitive closure)

Call Rule $\nearrow$

$$\mathrm{Push}(G, L, H', N'.M'),\ \mathrm{Step}(H', N', H', N')$$

$$:-$$

$$\mathrm{Step}(G, L, H, M),\quad (H, M) \to (H', N'.M').$$



Return Rule

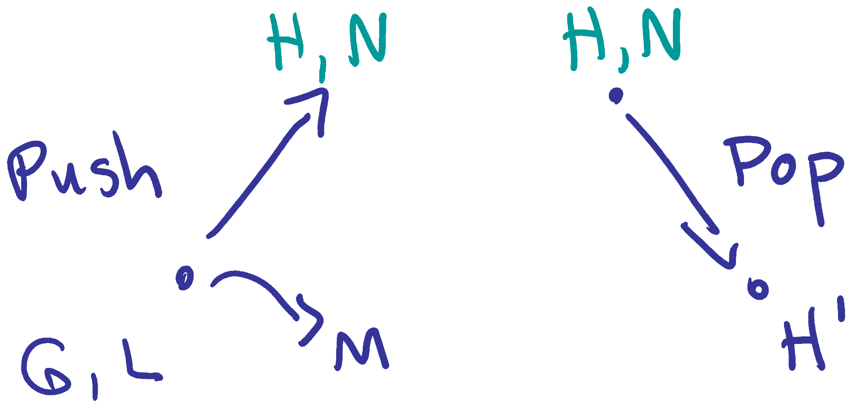
$$\mathrm{Pop}(G, L, H') \text{ :- }$$
$$\quad \mathrm{Step}(G, L, H, M),\ (H, M) \to (H', \varepsilon).$$

$$\begin{tikzcd} G, L \arrow[r] & H, M \arrow[d] \\ & H', \varepsilon \end{tikzcd}$$

Match Rule ↗ ↘

$$Step(G, L, H', M) :-$$

$$Push(G, L, H, N, M),\ Pop(H, N, H').$$



Termination

- $|Step| \le |G|^2|L|^2$
- $|Push| \le |G|^2|L|^3$
- $|Pop| \le |G|^2|L|$

Reachability in a PDS

$\forall\, g \in G$

$\exists\, w \in L^* \text{ s.t. } (g_0, l_0) \rightsquigarrow^* (g, w.l_0)$

iff

$\exists\, x, y, z : \mathrm{Step}(x, y, g, z)$

Proof

$\Leftarrow$. straightforward

$\Rightarrow$ Induction on length of $(g_0, l_0) \rightsquigarrow^* (g, l_s)$

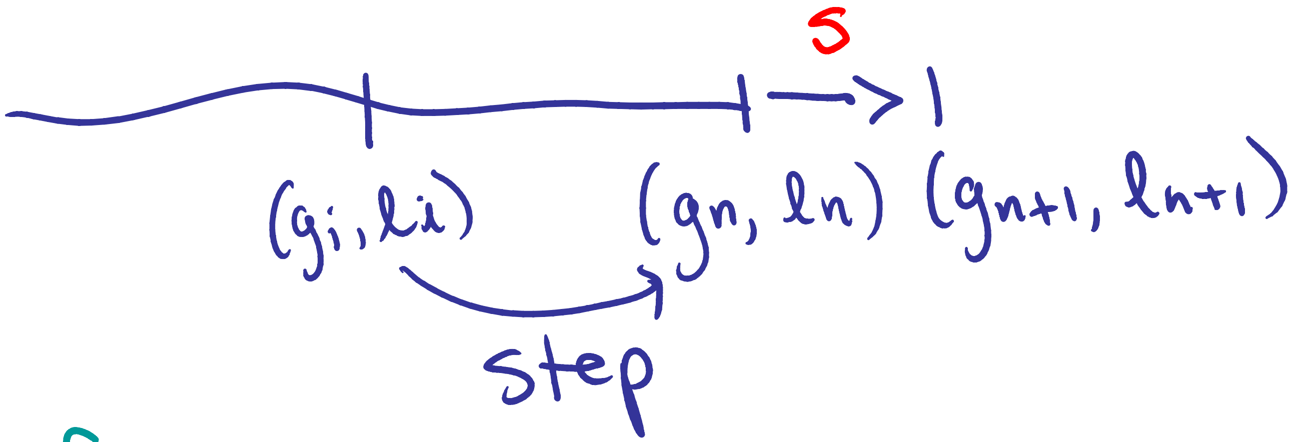
Base case: $n=0$. $g = g_0$, $l = l_0$

$Step(g_0, l_0, g_0, l_0)$.

Ind. Hyp: $n > 0$, $\exists i, 0 \le i \le n$ s.t.

$Step(g_i, l_i, g_n, l_n)$.

Induction Step



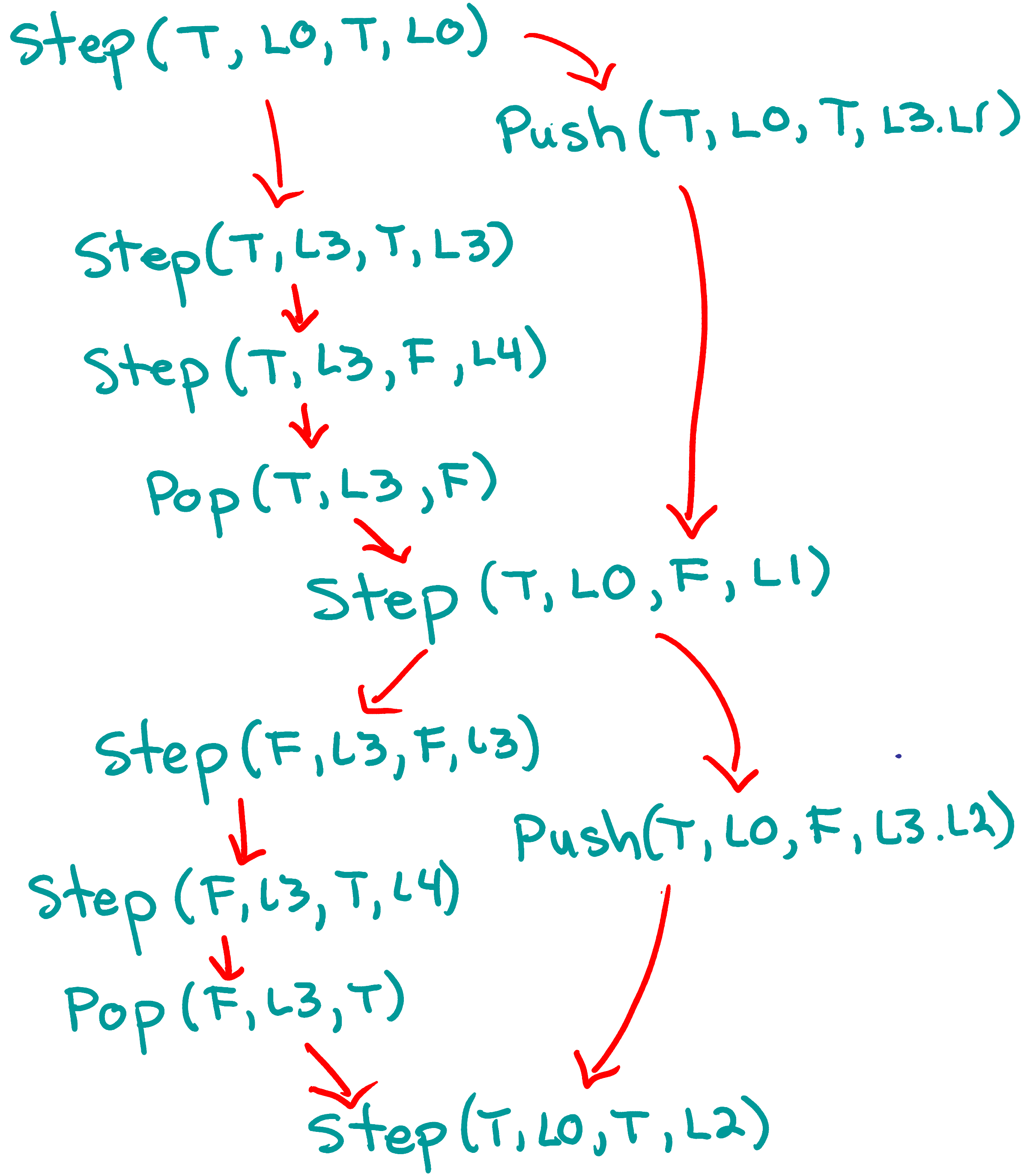
case s of

block:

call :

return:

```
bool g := T;
main() {
L0: flip();
L1: flip();
L2: assert(g);
}

flip() {
L3: g := !g;
L4: ;
}
```



Summary

- Global state reachability decidable for pushdown sys.

- Boolean programs can be compiled to PDS

- Decidability extends to regular properties of stack (+ more)

Static

analysis of

Sequential

programs on

Safety

properties

Dynamic
analysis of
Concurrent
programs on
Liveness
properties

CHESS: Systematic Concurrency Testing

Thomas Ball, Sebastian Burckhardt,
Madan Musuvathi, Shaz Qadeer

Software Reliability Research
Microsoft Research

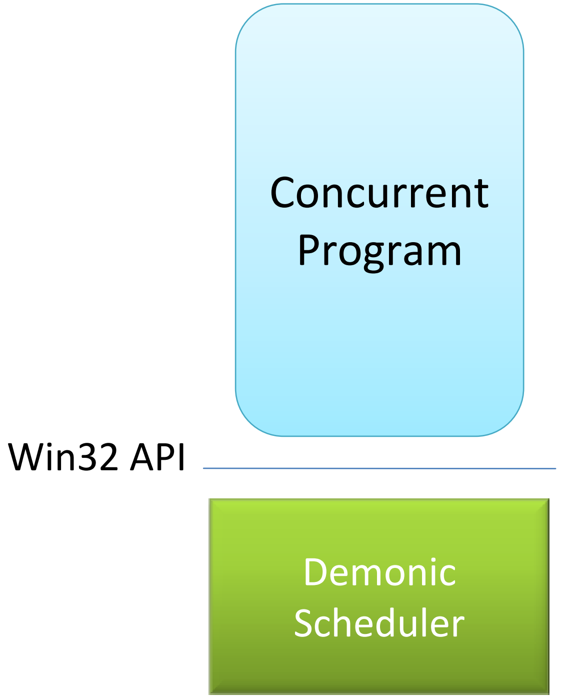
Testing concurrent programs is HARD

- Bugs hidden in rare thread interleavings

- Today, concurrency testing == stress testing
  - Poor coverage of interleavings
  - Unpredictable coverage results in "Heisenbugs"

- The mark of reliability of the system still remains its ability to withstand stress

CHESS in a nutshell



- Replace the OS scheduler with a demonic scheduler
- Systematically explore all scheduling choices

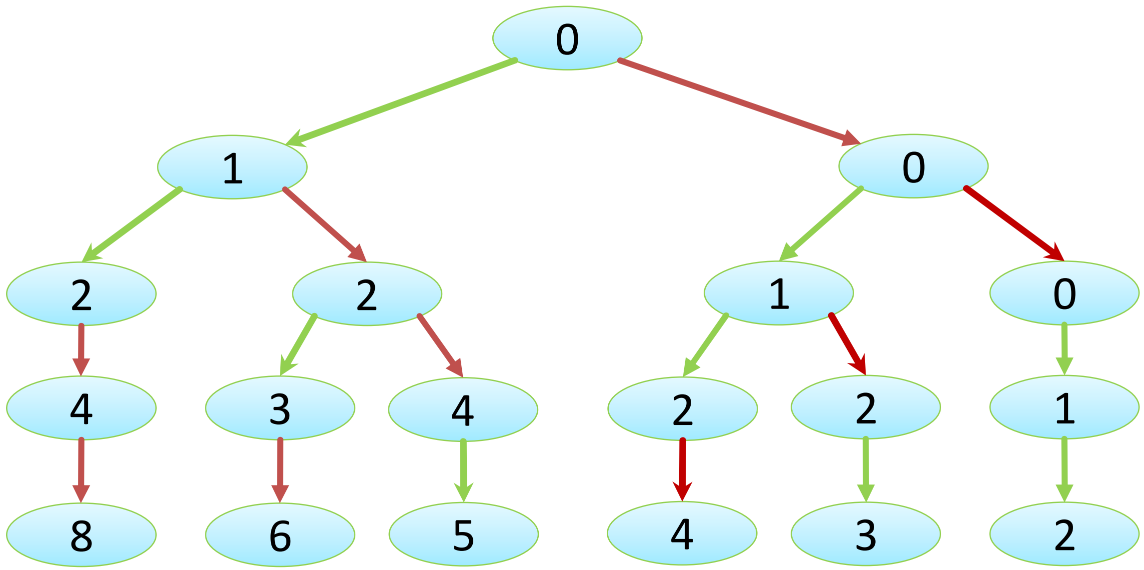
Enumerating thread interleavings

Thread 1

```
x++;
x++;
```

Thread 2

```
x*=2;
x*=2;
```

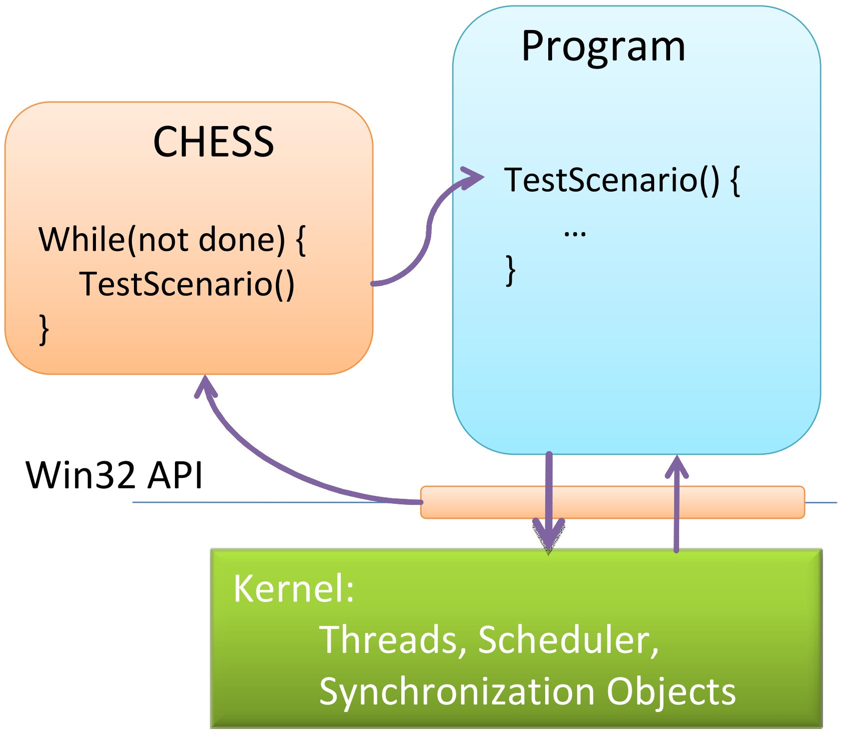


Demo: Don’t stress, use CHESS

CHESS goals

- Scale to large programs
- In the limit, verify that the program is correct for a given input
- Provide qualified coverage guarantees

CHESS architecture



CHESS runs the scenario in a loop

- Every run takes a different interleaving
- Every run is repeatable

Intercept synch. & threading calls

- To control and introduce nondeterminism

Detect

- Assertion violations
- Deadlocks
- Dataraces
- Livelocks

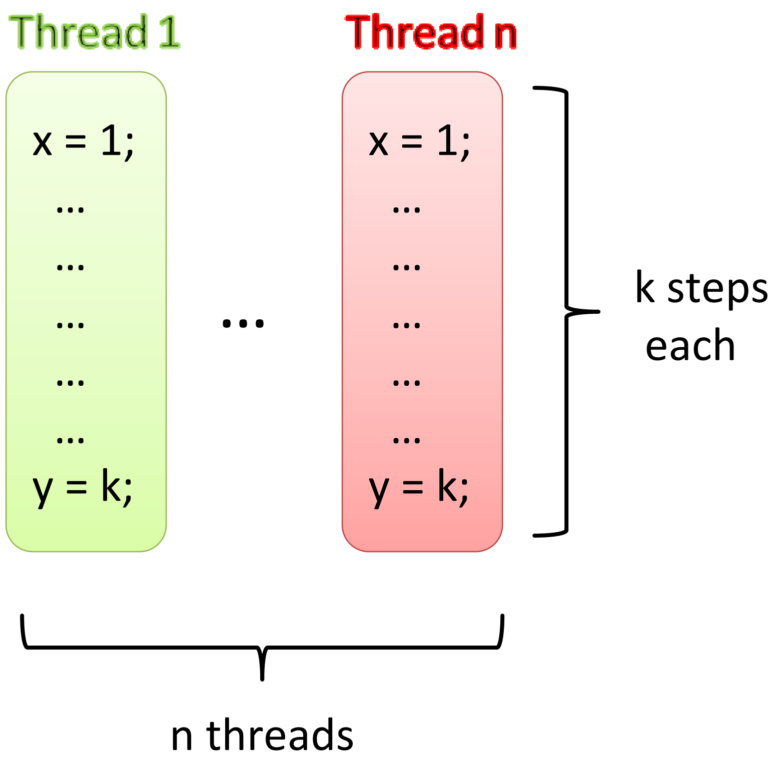
Stateless model checking [Verisoft ‘97]

- Systematically enumerate all paths in a state-space graph
- Don’t capture program states
  - Capturing states is extremely hard for large programs
- Effective for message-passing programs

Outline

- Preemption bounding
  - Makes CHESS effective on deep state spaces
- Fair stateless model checking

State space explosion



- Number of executions

  $= O(n^{nk})$

- Exponential in both n and k
  - Typically: $n < 10$ $k > 100$
- Limits scalability to large programs

Goal: Scale CHESS to large programs (large k)

Preemption bounding

- Prioritize executions with small number of preemptions
- Two kinds of context switches:
  - Preemptions – forced by the scheduler
    - e.g. Time-slice expiration
  - Non-preemptions – a thread voluntarily yields
    - e.g. Blocking on an unavailable lock, thread end

Thread 1		Thread 2
x = 1; if (p != 0) {		p = 0;
	← preemption	
x = p->f; }		

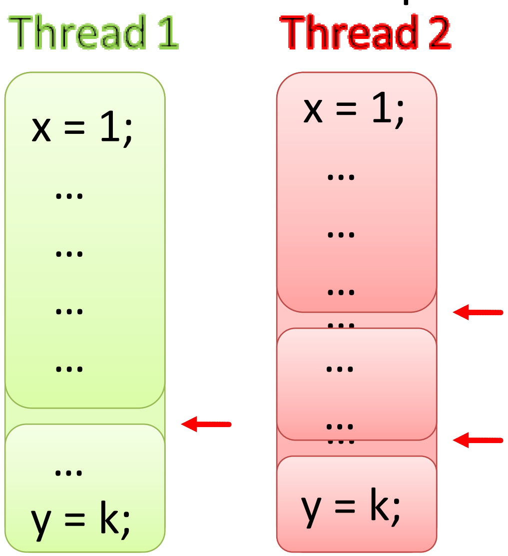
← non-preemption

Polynomial state space

- Terminating program with fixed inputs and deterministic threads
  - n threads, k steps each, c preemptions
- Number of executions <= ${}_{nk}C_c \,.\, (n+c)!$

$$= O(\,(n^2k)^c .\, n!\,)$$

Exponential in n and c, but not in k



- Choose c preemption points
- Permute n+c atomic blocks

Find lots of bugs with 2 preemptions

Program	Lines of code	Bugs
Work Stealing Q	4K	4
CDS	6K	1
CCR	9K	3
ConcRT	16K	4
Dryad	18K	7
APE	19K	4
STM	20K	2
TPL	24K	9
PLINQ	24K	1
Singularity	175K	2
		37 (total)

Acknowledgement: testers from PCP team

Outline

- Preemption bounding
  - Makes CHESS effective on deep state spaces
- Fair stateless model checking
  - Makes CHESS effective on cyclic state spaces
  - Enables CHESS to find liveness violations (livelocks)

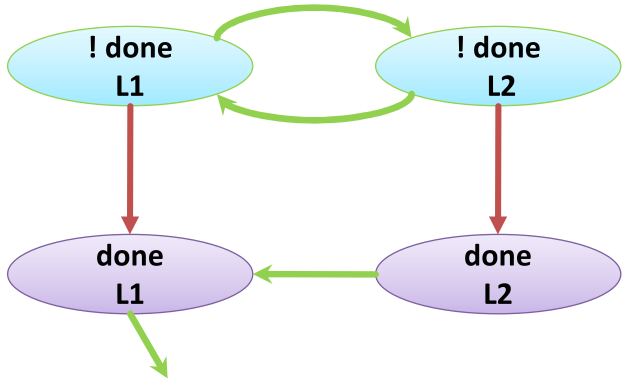
Concurrent programs have cyclic state spaces

Thread 1

```
L1:  while( ! done) {
L2:     Sleep();
     }
```

Thread 2

```
M1: done = 1;
```



- Spinlocks
- Non-blocking algorithms
- Implementations of synchronization primitives
- Periodic timers
- …

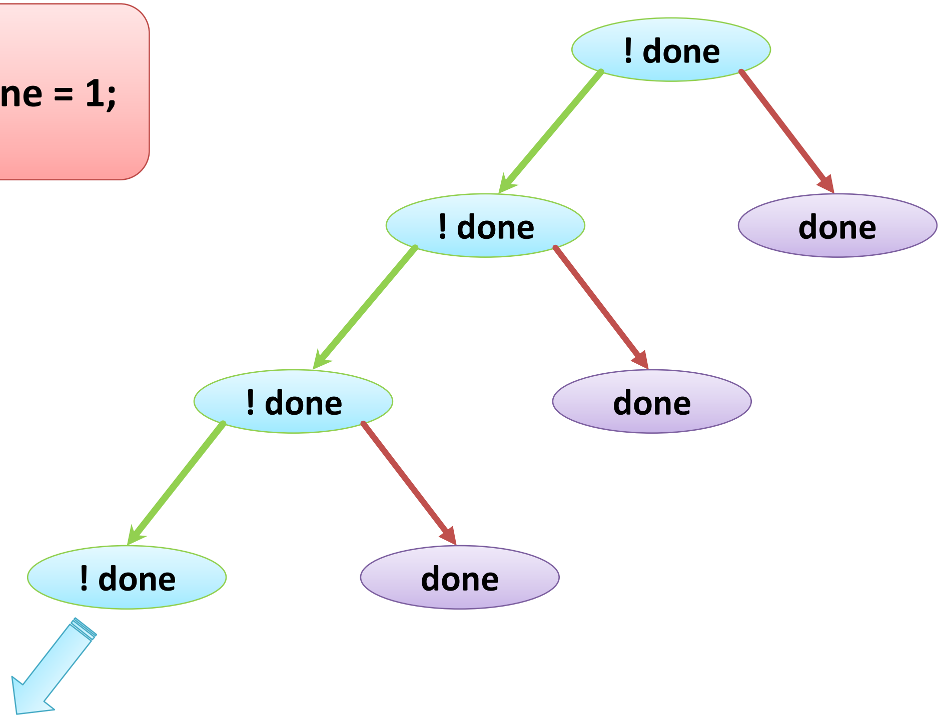
A demonic scheduler unrolls any cycle ad-infinitum

Thread 1

```
while( ! done)
{
    Sleep();
}
```

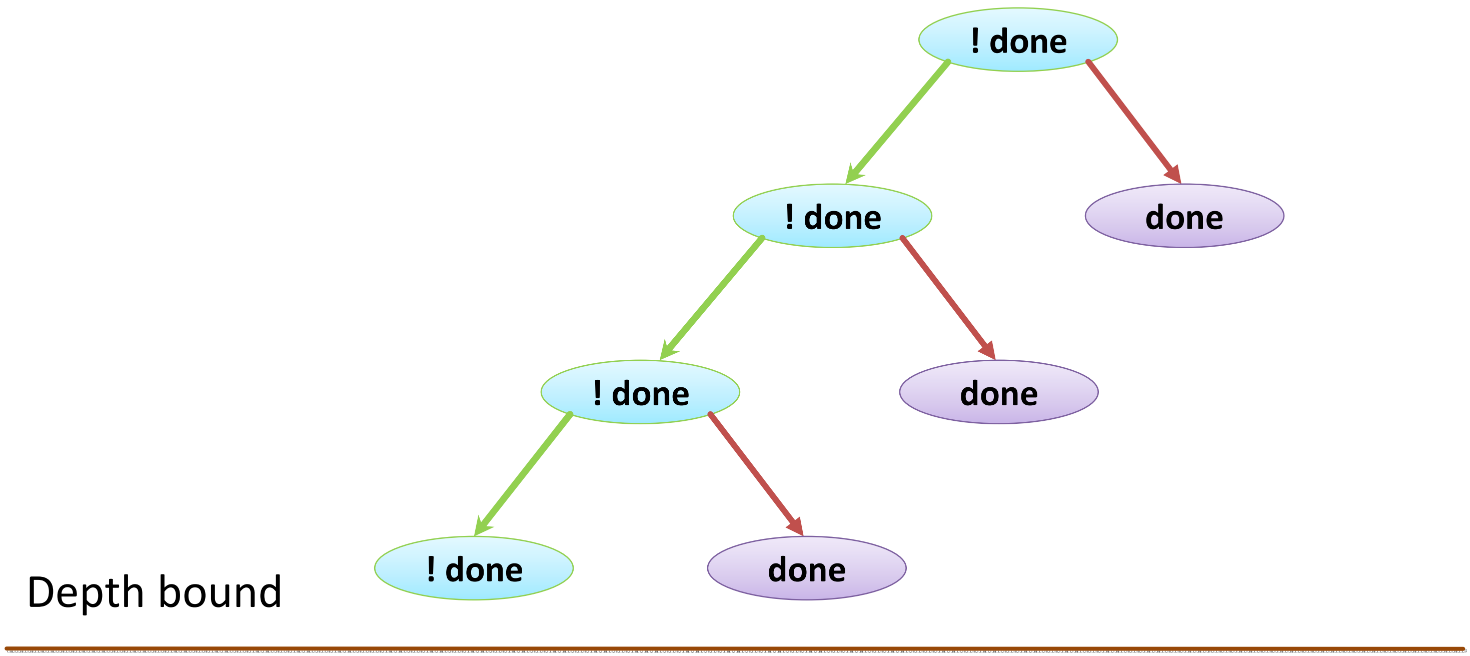
Thread 2

```
done = 1;
```



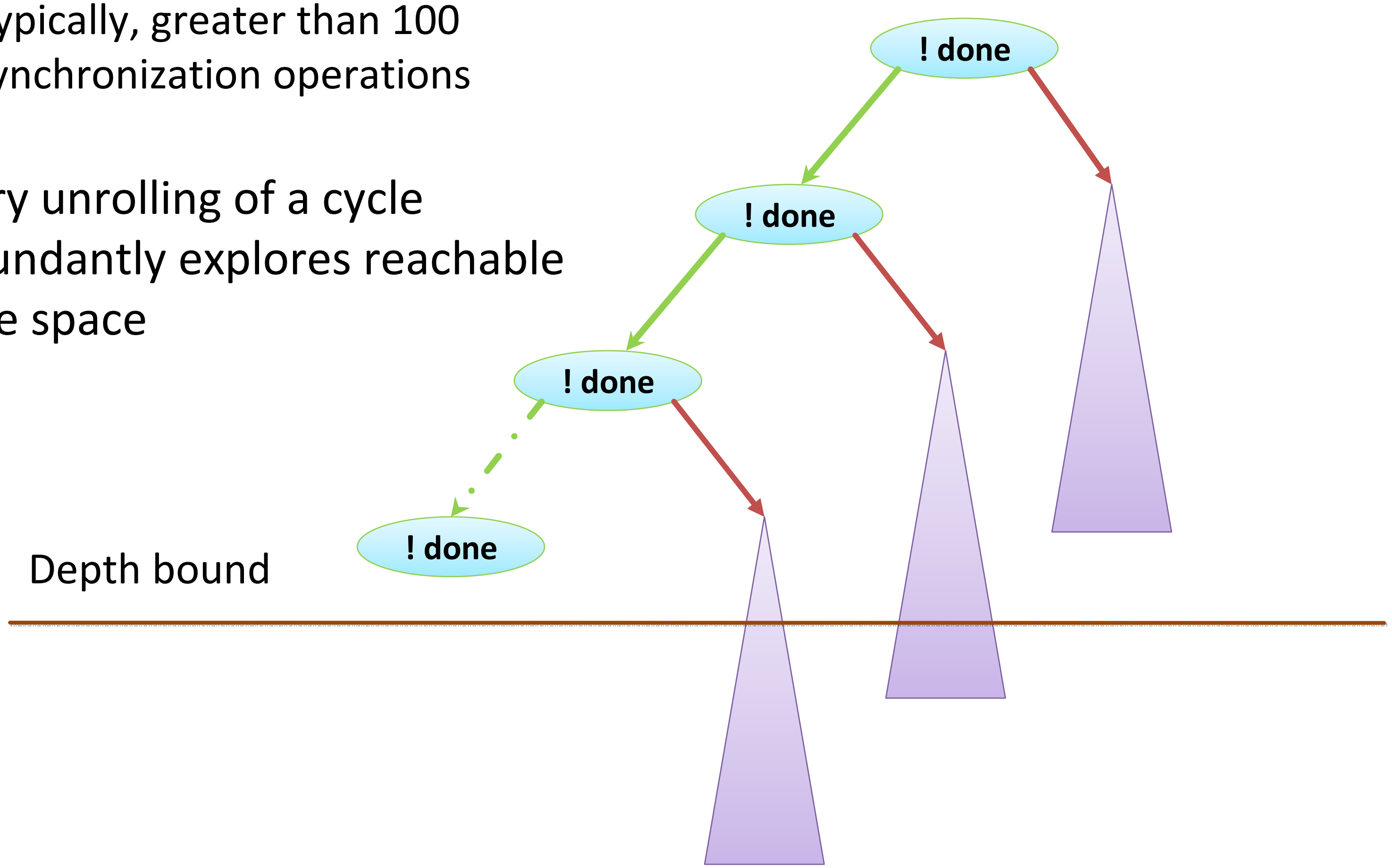
Depth bounding

- Prune executions beyond a bounded number of steps



Problem 1: Ineffective state coverage

- Bound has to be large enough to reach the deepest bug
  - Typically, greater than 100 synchronization operations
- Every unrolling of a cycle redundantly explores reachable state space



Problem 2: Cannot find livelocks

- Livelocks : lack of progress in a program

Thread 1

```
temp = done;
while( ! temp)
{
    Sleep();
}
```

Thread 2

```
done = 1;
```

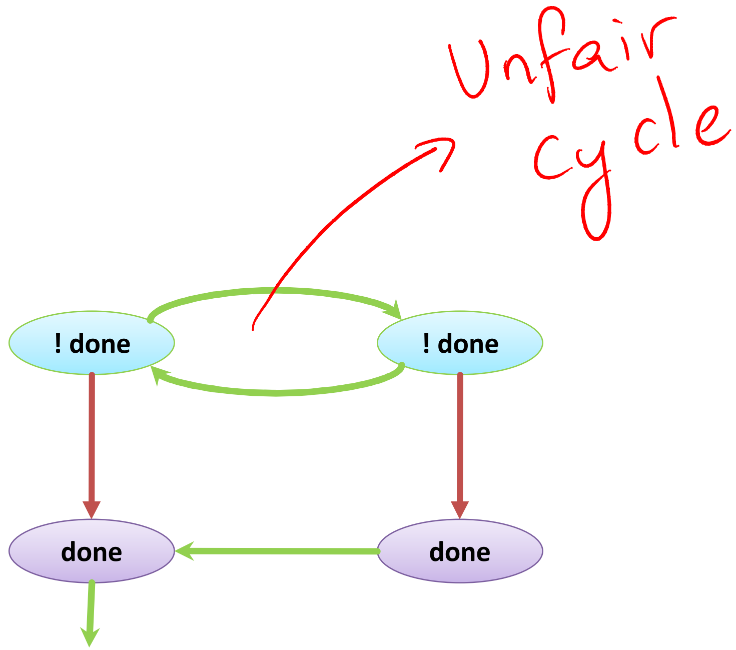
Key idea

Thread 1

```
while( ! done)
{
    Sleep();
}
```

Thread 2

```
done = 1;
```

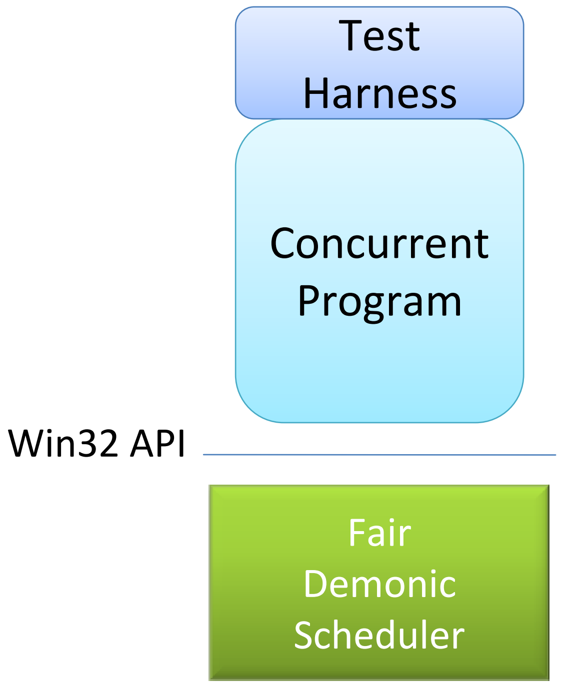


- This test terminates only when the scheduler is fair
- Fairness is assumed by programmers

All cycles in correct programs are unfair

A fair cycle is a livelock

We need a fair demonic scheduler



- Avoid unrolling unfair cycles
  - Effective state coverage
- Detect fair cycles
  - Find livelocks

Good Samaritan violation

- Thread yield the processor when not making progress
  - Forall threads t : GF scheduled(t) → GF yield(t)

Thread 1

```
while( ! done)
{
     ;
}
```

Thread 2

```
done = 1;
```

- Found many such violations, including one in the Singularity boot process
  - Results in “sluggish I/O” behavior during bootup

Conclusion

- Don't stress, use CHESS
- CHESS binary and papers available at http://research.microsoft.com/CHESS

Questions