Software Model Checking with SLAM

Thomas Ball
Software Reliability Research
MSR Redmond

Sriram K. Rajamani
Reliable Software Engineering
MSR Bangalore

http://research.microsoft.com/slam/

People behind SLAM

Partners in MS

- Ella Bounimova, Byron Cook, Nar Ganapathy, Shuvendu Lahiri, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bonus Ondrusek, Abdullah Ustuner

Summer interns

- Sagar Chaki, Satyaki Das, Rahul Kumar, Shuvendu Lahiri, Jakob Lichtenberg, Rupak Majumdar, Todd Millstein, Mayur Naik, Robby, Wes Weimer, George Weissenbacher, Fei Xie

Visitors

- Giorgio Delzanno, Albert Oliveras, Andreas Podelski, Stefan Schwoon

Outline

- Part I: Overview (30 min)
  - overview of SLAM process
  - demonstration (Static Driver Verifier)
  - lessons learned
- Part II: Basic SLAM (1 hour)
  - foundations
  - basic algorithms (no pointers)
- Part III: Advanced Topics (30 min)
  - pointers + procedures

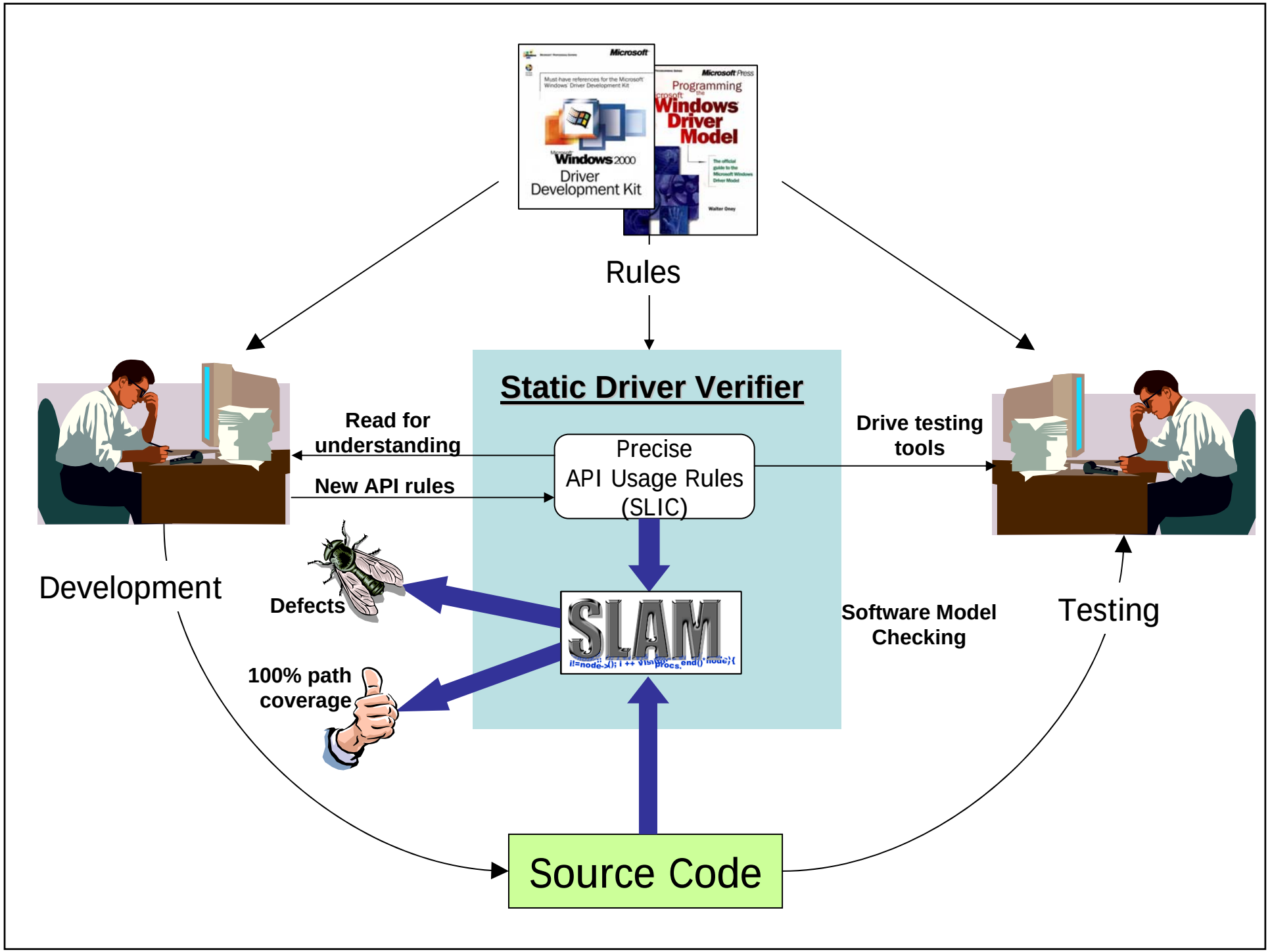
Part I: Overview of SLAM

What is SLAM?

SLAM is a software model checking project at Microsoft Research

- started in 2000
- check C programs against safety properties
- first application domain: device drivers
- still active!

Ships in the Windows Driver Kit, as the analysis engine in Static Driver Verifier


Microsoft
Windows 2000
Driver
Development Kit
Microsoft Press
Programming the
Windows
Driver
Model
Rules
Static Driver Verifier
Read for
understanding
New API rules
Precise
API Usage Rules
(SLIC)
Drive testing
tools
Development
Defects
SLAM
Software Model
Checking
Testing
100% path
coverage
Source Code

SLAM – Software Model Checking

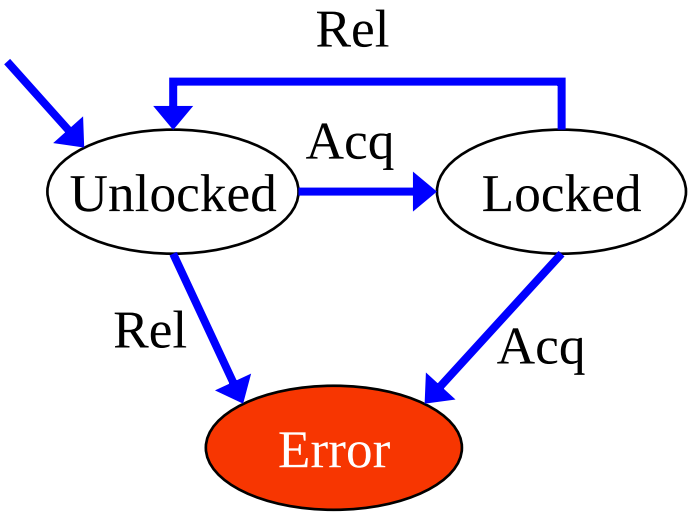
- Counterexample-driven abstraction of software
  - boolean programs
  - model creation (c2bp)
  - model checking (bebop)
  - model refinement (newton)

- Builds on
  - OCaml programming language
  - dataflow and pointer analysis
  - predicate abstraction
  - symbolic model checking
  - CUDD, SAT solver, SMT theorem prover

SLIC

- Finite state language for stating rules
  - monitors behavior of C code
  - temporal safety properties (security automata)
  - familiar C syntax
- Suitable for expressing control-dominated properties
  - e.g. proper sequence of events
  - can encode data values inside state

State Machine for Locking



Locking Rule in SLIC

```
state {
  enum {Locked,Unlocked}
    s := Unlocked;
}

KeAcquireSpinLock.entry {
  if (s=Locked) abort;
  else s := Locked;
}

KeReleaseSpinLock.entry {
  if (s=Unlocked) abort;
  else s := Unlocked;
}
```

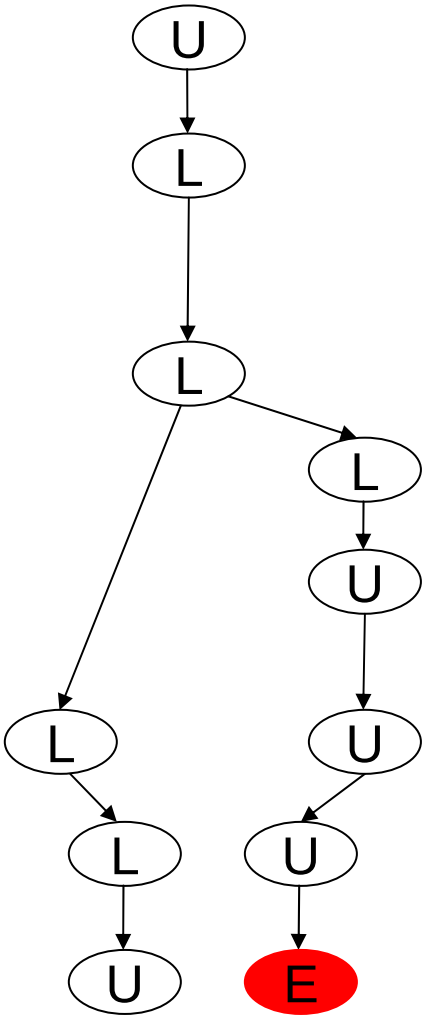
Example

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 (bebop)

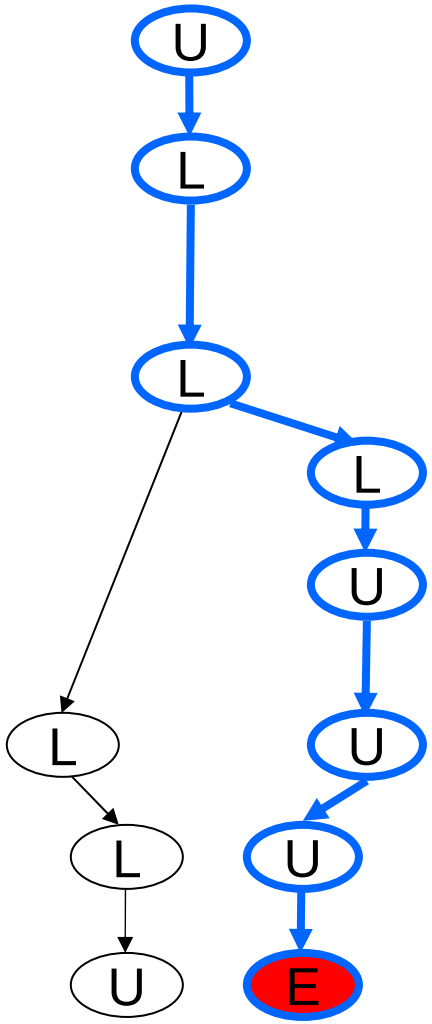



```
do {
   KeAcquireSpinLock();


   if(*){

       KeReleaseSpinLock();

   }
} while (*);

KeReleaseSpinLock();
```

Example

b : (nPacketsOld = nPackets)

Is error path feasible in C program? (newton)

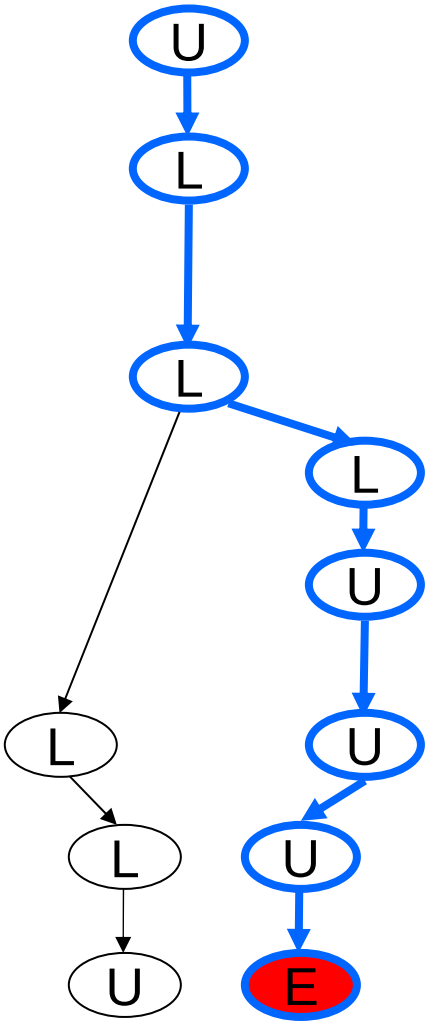



```
do {
   KeAcquireSpinLock();

   nPacketsOld := nPackets;

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

KeReleaseSpinLock();
```

Example

b : (nPacketsOld = nPackets)

Add new predicate to boolean program (c2bp)

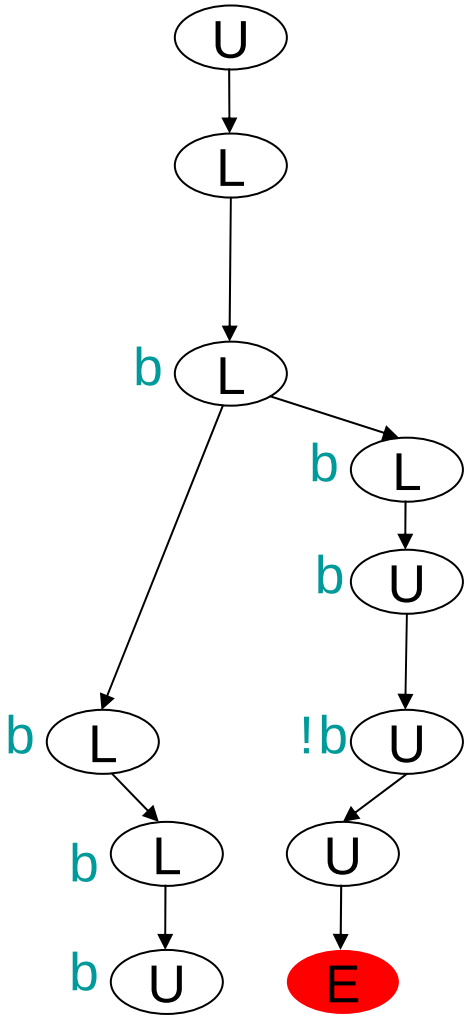



```
do {
   KeAcquireSpinLock();

   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 (bebop)

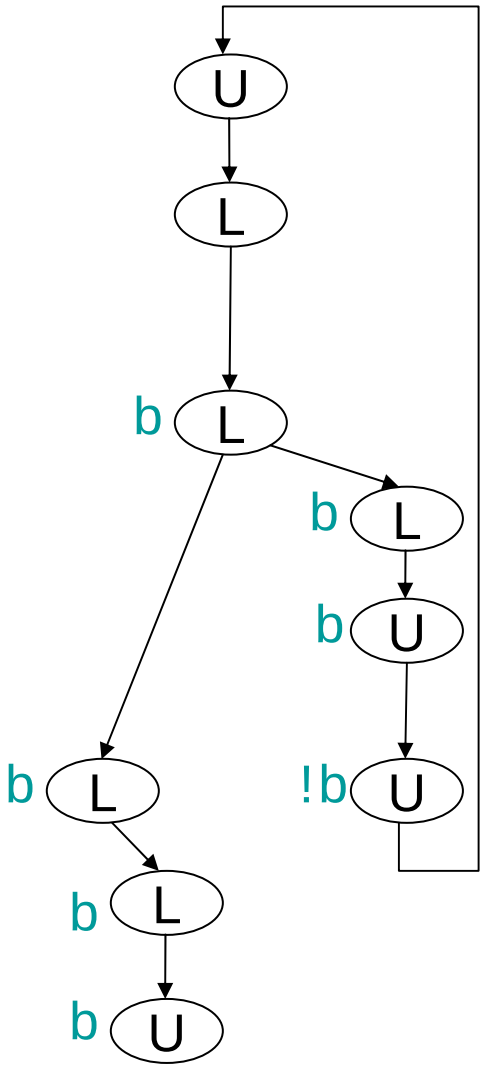



```
do {
   KeAcquireSpinLock();

   b := true;

   if(*){

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

KeReleaseSpinLock();
```

Example

b : (nPacketsOld = nPackets)

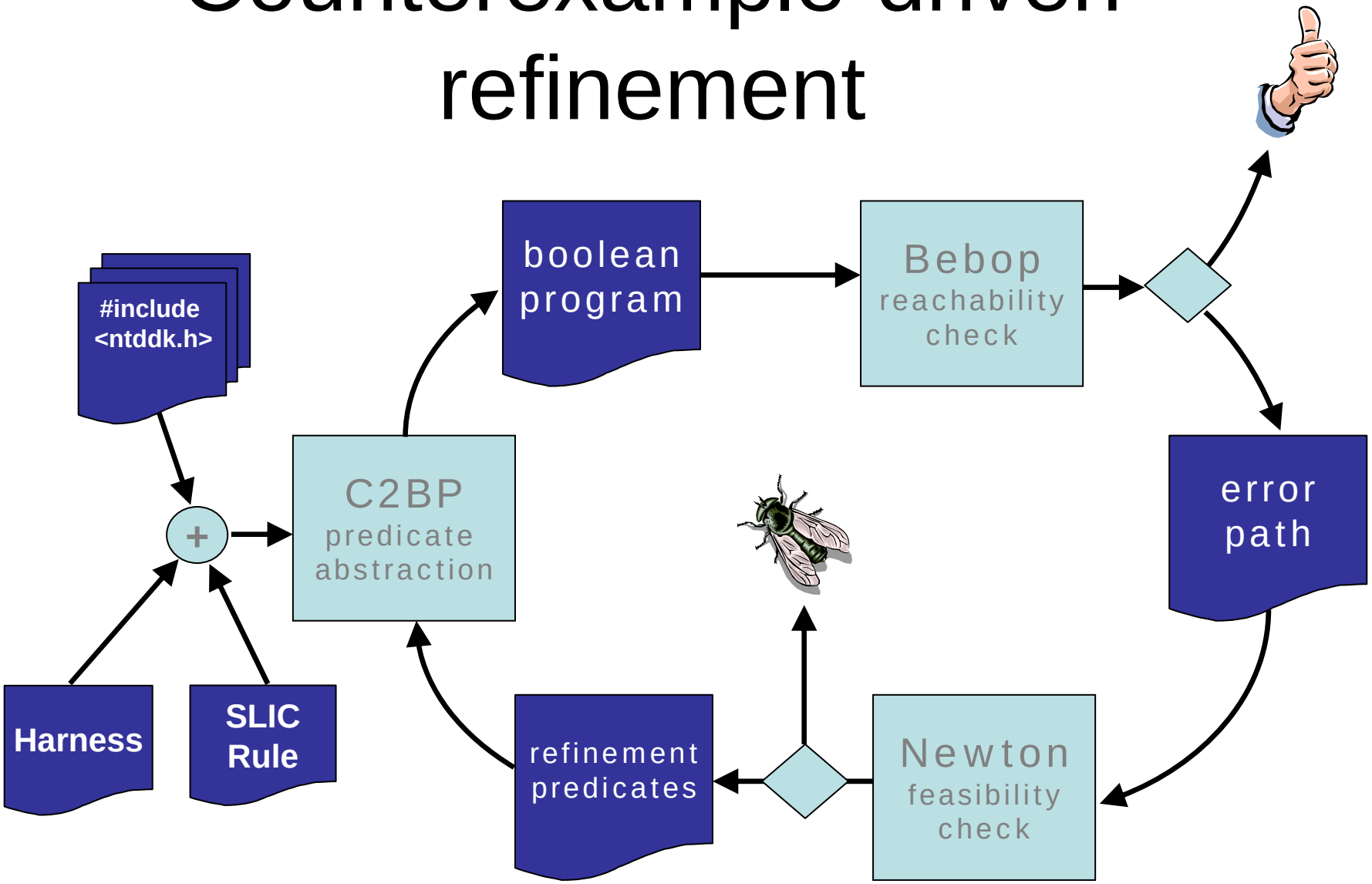
Model checking refined boolean program (bebop)




```
do {
   KeAcquireSpinLock();

   b := true;

   if(*){

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

KeReleaseSpinLock();
```

Inferred Invariant

Counterexample-driven refinement



Part I: Demo

Static Driver Verifier

src/kernel/serenum, lowerdriverreturn.slic

Part I: Lessons Learned

SLAM

- Boolean program model has proved itself
- Successful for domain of device drivers
  - control-dominated safety properties
  - few boolean variables needed to do proof or find real counterexamples
- Counterexample-driven refinement
  - terminates in practice
  - incompleteness of theorem prover not an issue

What Was Hard?

- Abstracting
  - from a language with pointers (C)
  - to one without pointers (boolean programs)
  - all side effects need to be modeled by copying
- Developing specifications, environment model
- False alarms
- Technology transfer!

Part II: Basic SLAM

C-

Types	τ	::=	void | bool | int | ref τ
Expressions	**e**	::=	c | x | $\mathbf{e_1}$ op $\mathbf{e_2}$ | &x | *x
LExpression	ℓ	::=	x | *x
Declaration	**d**	::=	τ $x_1, x_2, \ldots, x_n$
Statements	**s**	::=	skip | goto $L_1, L_2 \ldots L_n$ | L: s
		|	assume(**e**)
		|	ℓ := **e**
		|	ℓ := f ($\mathbf{e_1}, \mathbf{e_2}, \ldots, \mathbf{e_n}$)
		|	return x
		|	$\mathbf{s_1}$; $\mathbf{s_2}$; ... ; $\mathbf{s_n}$
Procedures	**p**	::=	τ f (x_1: τ_1, x_2 : τ_2, ..., x_n : τ_n)
Program	**g**	::=	$\mathbf{d_1}$ $\mathbf{d_2}$ ... $\mathbf{d_n}$ $\mathbf{p_1}$ $\mathbf{p_2}$ ... $\mathbf{p_n}$

C--

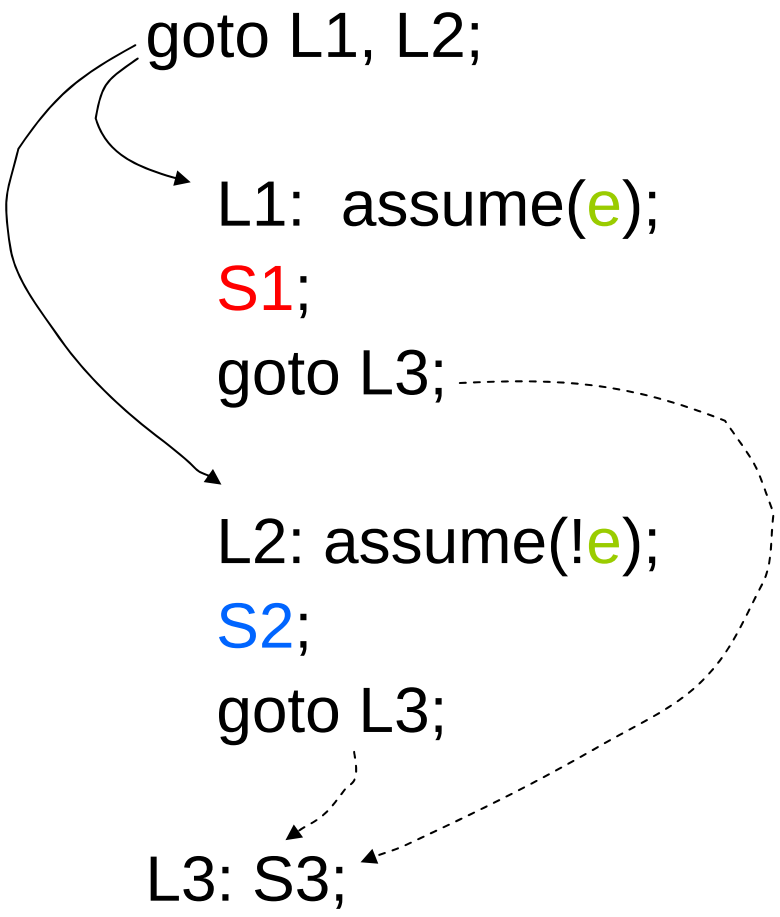
Types	τ	::=	void | bool | int
Expressions	**e**	::=	c | x | $\mathbf{e_1}$ op $\mathbf{e_2}$
LExpression	ℓ	::=	x
Declaration	**d**	::=	τ $x_1,x_2,\ldots,x_n$
Statements	**s**	::=	skip | goto $L_1,L_2 \ldots L_n$ | L: s
		|	assume(**e**)
		|	ℓ := **e**
		|	f ($\mathbf{e_1}$,$\mathbf{e_2}$,…,$\mathbf{e_n}$)
		|	return
		|	$\mathbf{s_1}$; $\mathbf{s_2}$;… ; $\mathbf{s_n}$
Procedures	**p**	::=	f (x_1: τ_1,x_2 : τ_2,…,x_n : τ_n)
Program	**g**	::=	$\mathbf{d_1}$ $\mathbf{d_2}$ … $\mathbf{d_n}$ $\mathbf{p_1}$ $\mathbf{p_2}$ … $\mathbf{p_n}$

BP

Types	τ	::=	void | bool
Expressions	**e**	::=	c | x | $\mathbf{e_1}$ op $\mathbf{e_2}$
LExpression	ℓ	::=	x
Declaration	**d**	::=	τ $x_1,x_2,\ldots,x_n$
Statements	**s**	::=	skip | goto $L_1,L_2 \ldots L_n$ | L: s
		|	assume(**e**)
		|	ℓ := **e**
		|	f ($\mathbf{e_1}$,$\mathbf{e_2}$,…,$\mathbf{e_n}$)
		|	return
		|	$\mathbf{s_1}$; $\mathbf{s_2}$;… ; $\mathbf{s_n}$
Procedures	**p**	::=	f (x_1: τ_1,x_2 : τ_2,…,x_n : τ_n)
Program	**g**	::=	$\mathbf{d_1}$ $\mathbf{d_2}$ … $\mathbf{d_n}$ $\mathbf{p_1}$ $\mathbf{p_2}$ … $\mathbf{p_n}$

Syntactic sugar

```
if (e) {
    S1;
} else {
    S2;
}
S3;
```



Example, in C

```
int g;

main(int x, int y){

  cmp(x, y);

  if (!g) {
      if (x != y)
          assert(0);
  }
}
```

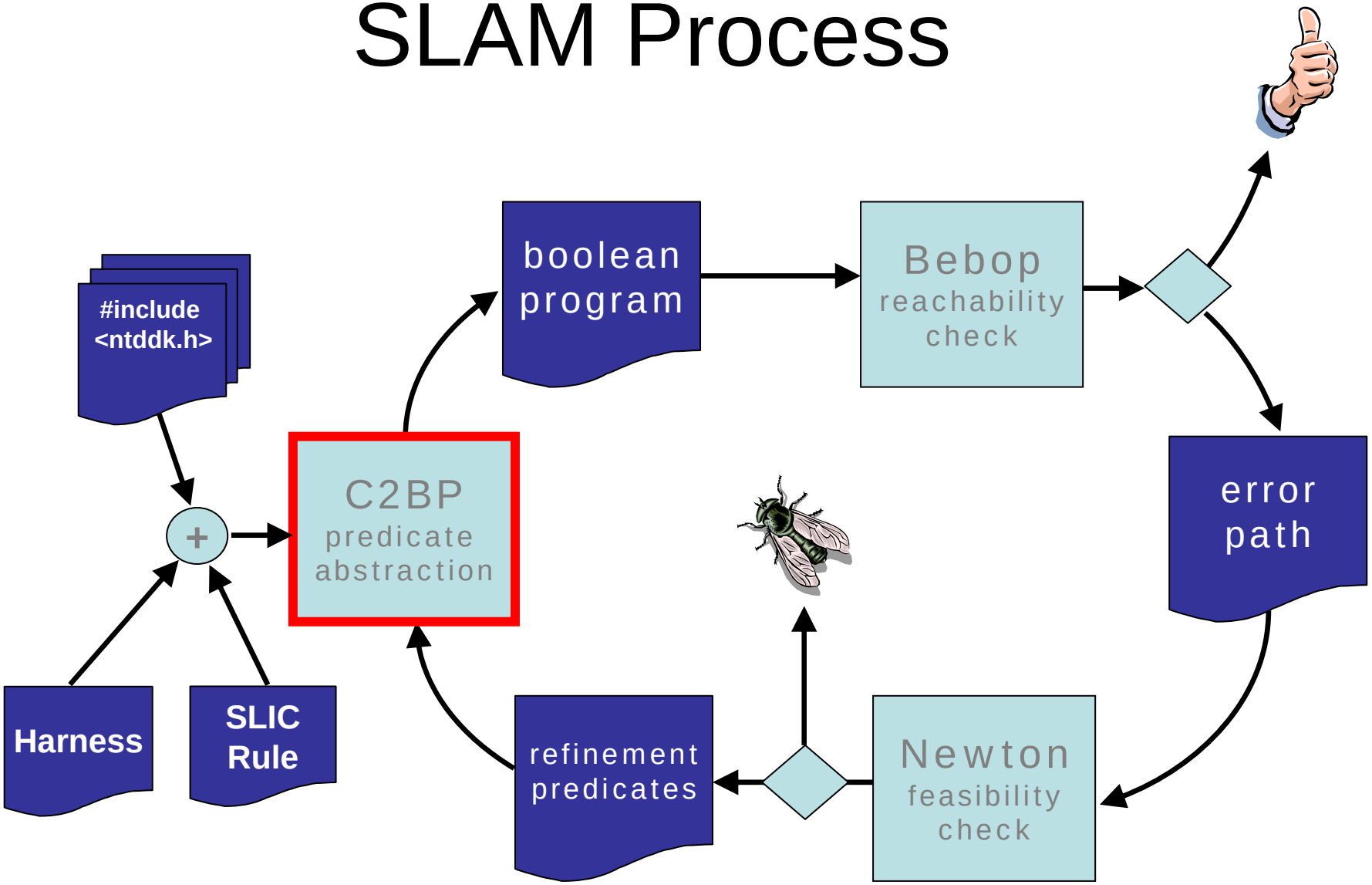
```
void cmp (int a , int b) {
  if (a = b)
        g := 0;
  else
        g := 1;
}
```

Example, in C--

```
int g;

main(int x, int y){

  cmp(x, y);

  assume(!g);
  assume(x != y)
  assert(0);
}
```

```
void cmp(int a , int b) {
   goto L1, L2;

   L1: assume(a=b);
       g := 0;
       return;

   L2: assume(a!=b);
       g := 1;
       return;
}
```

SLAM Process



c2bp: Predicate Abstraction for C Programs

Given

- P : a C program
- F : $\{e_1,...,e_n\}$
  - each e_i a pure boolean expression
  - each e_i represents set of states for which e_i is true

Produce a *boolean program* B(P,F)

- same control-flow structure as P
- boolean vars $\{b_1,...,b_n\}$ to match $\{e_1,...,e_n\}$
- B(P,F) simulates P

c2bp Algorithm

- Performs modular abstraction
  - abstracts each procedure in isolation
- Within each procedure, abstracts each statement in isolation
  - no control-flow analysis
  - no need for loop invariants

```
int g;

main(int x, int y){

  cmp(x, y);

  assume(!g);
  assume(x != y)
  assert(0);
}
```

```
void cmp (int a , int b) {
   goto L1, L2

   L1: assume(a=b);
        g := 0;
        return;

  L2: assume(a!=b);
        g := 1;
        return;
}
```

Preds: {x=y}

{g=0}

{a=b}

```
int g;

main(int x, int y){

  cmp(x, y);

  assume(!g);
  assume(x != y)
  assert(0);
}
```

```
decl {g=0} ;

main( {x=y} ) {

}
```

Preds: {x=y}

{g=0}

{a=b}

```
void cmp (int a , int b) {
   goto L1, L2

   L1: assume(a=b);
        g := 0;
        return;

  L2: assume(a!=b);
        g := 1;
        return;
}
```

```
void cmp ( {a=b} ) {

}
```

```
int g;

main(int x, int y){

  cmp(x, y);

  assume(!g);
  assume(x != y)
  assert(0);
}
```

```
void cmp (int a , int b) {
   goto L1, L2

   L1: assume(a=b);
        g := 0;
        return;

   L2: assume(a!=b);
        g := 1;
        return;
}
```

```
decl {g=0} ;

main( {x=y} ) {

  cmp( {x=y} );

  assume( {g=0} );
  assume( !{x=y} );
  assert(0);
}
```

```
void cmp ( {a=b} ) {
  goto L1, L2;

  L1: assume( {a=b} );
       {g=0} := T;
       return;

  L2: assume( !{a=b} );
       {g=0} := F;
       return;
}
```

Preds: {x=y}

{g=0}

{a=b}

C--

Types	τ	::=	void | bool | int
Expressions	**e**	::=	c | x | $\mathbf{e_1}$ op $\mathbf{e_2}$
LExpression	ℓ	::=	x
Declaration	**d**	::=	τ $x_1, x_2, \ldots, x_n$
Statements	**s**	::=	skip | goto $L_1, L_2 \ldots L_n$ | L: s
			| assume(**e**)
			| ℓ := **e**
			| f ($\mathbf{e_1}, \mathbf{e_2}, \ldots, \mathbf{e_n}$)
			| return
			| $\mathbf{s_1}$; $\mathbf{s_2}$; ... ; $\mathbf{s_n}$
Procedures	**p**	::=	f ($x_1 : \tau_1, x_2 : \tau_2, \ldots, x_n : \tau_n$)
Program	**g**	::=	$\mathbf{d_1}$ $\mathbf{d_2}$ ... $\mathbf{d_n}$ $\mathbf{p_1}$ $\mathbf{p_2}$ ... $\mathbf{p_n}$

Abstracting Assigns via WP

- Statement y:=y+1 and F={ y<4, y<5 }
  - {y<4}, {y<5} := *, {y<4} ;
- WP(x:=e,Q) = Q[x -> e]
- WP(y:=y+1, y<5) =
  (y<5) [y -> y+1] =
  (y+1<5) =
  (y<4)

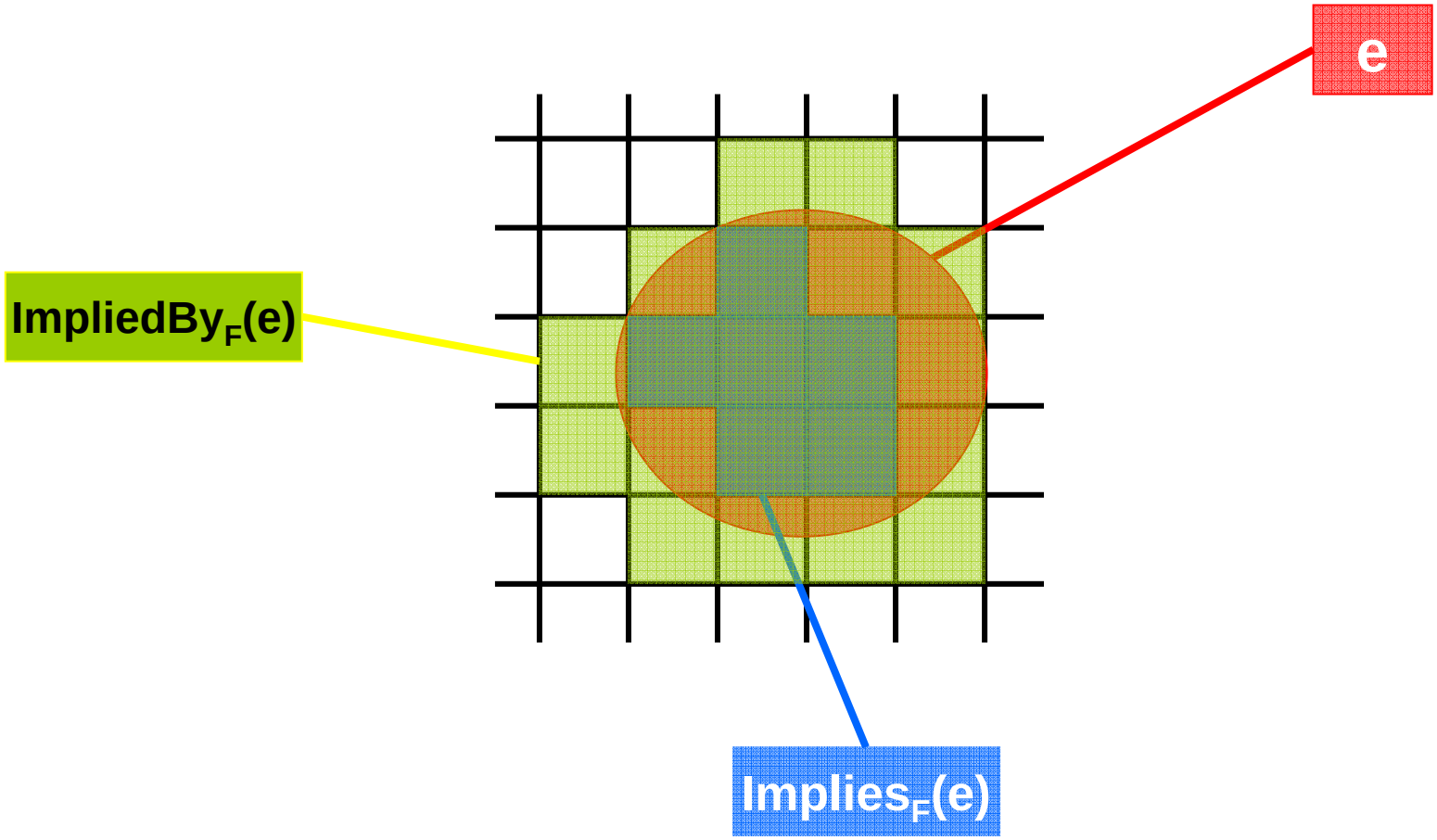
WP Problem

- WP(s, e_i) not always expressible via { e_1,...,e_n }
- Example
  - F = { x=0, x=1, x<5 }
  - WP(x:=x+1 , x<5) = x<4
  - Best possible: x=0 || x=1

Abstracting Expressions via F

- F = { e_1,...,e_n }
- $\text{Implies}_F(e)$
  - *best* boolean function over F *that implies* e
- $\text{ImpliedBy}_F(e)$
  - *best* boolean function over F *implied by* e
  - $\text{ImpliedBy}_F(e) = !\text{Implies}_F(!e)$

$Implies_F(e)$ and $ImpliedBy_F(e)$



Abstracting Assignments

- if $\text{Implies}_F(\text{WP}(s, e_i))$ is true before s then
  - e_i is true after s
- if $\text{Implies}_F(\text{WP}(s, !e_i))$ is true before s then
  - e_i is false after s

$$\{e_i\} := \begin{array}{ll} \text{Implies}_F(\text{WP}(s, e_i)) & ? \text{ true} : \\ \text{Implies}_F(\text{WP}(s, !e_i)) & ? \text{ false} \\ & : *; \end{array}$$

Assignment Example

Statement in P:

y := y+1;

Predicates in E:

{x=y}

Weakest Precondition:

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

$Implies_F$(x=y+1) = false

$Implies_F$(x!=y+1) = x=y

Abstraction of assignment:

{x=y} := {x=y} ? false : *;

Abstracting Assumes

- assume(e) is abstracted to:

  assume(ImpliedBy_F(e))

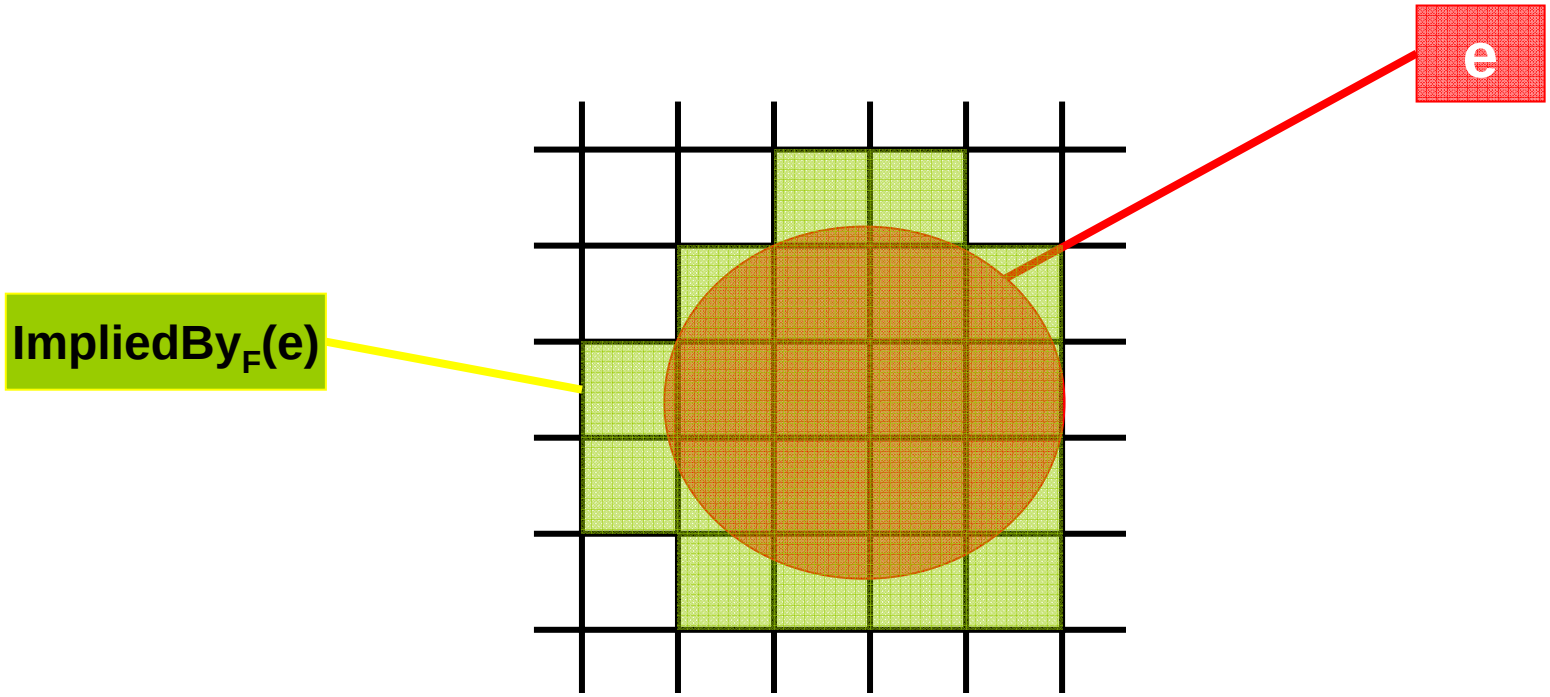
- Example:

  F = {x=2, x<5}

  assume(x < 2) is abstracted to:

  assume({x<5} && !{x=2})

Assume(e) and $\text{ImpliedBy}_F(e)$



Abstracting Procedures

- Each predicate in F is annotated as being either global or local to a particular procedure

- Procedures abstracted in two passes:
  - a *signature* is produced for each procedure in isolation
  - procedure calls are abstracted given the callees' signatures

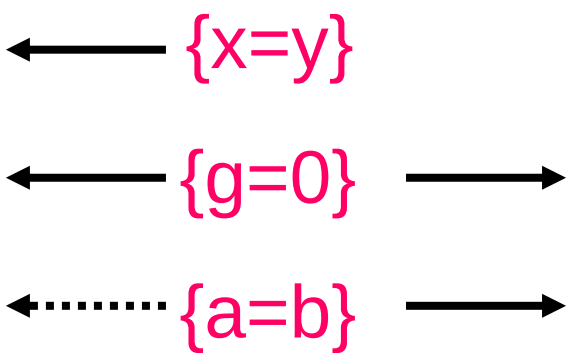
Abstracting a procedure call

- Procedure call
  - a sequence of assignments from actuals to formals
  - see assignment abstraction
- Procedure return
  - NOP for C-- with assumption that all predicates mention either only globals or only locals
  - with pointers and with mixed predicates:
    - most complicated part of c2bp
    - covered in the advanced topics section

```
int g;

main(int x, int y){

  cmp(x, y);

  assume(!g);
  assume(x != y)
  assert(0);
}
```

```
void cmp (int a , int b) {
goto L1, L2

   L1: assume(a=b);
        g := 0;
        return;

  L2: assume(a!=b);
        g := 1;
        return;
}
```

```
decl {g=0} ;

main( {x=y} ) {

  cmp( {x=y} );

  assume( {g=0} );
  assume( !{x=y} );
  assert(0);
}
```




```
void cmp ( {a=b} ) {
Goto L1, L2

  L1: assume( {a=b} );
        {g=0} := T;
        return;

  L2: assume( !{a=b} );
        {g=0} := F;
        return;
}
```

Computing $\text{Implies}_F(e)$

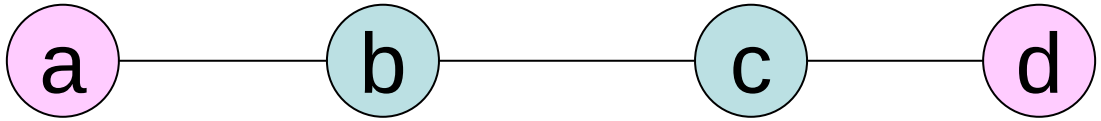
- *minterm* $m = d_1$ && ... && d_n
  - where $d_i = e_i$ or $d_i = !e_i$
- $\text{Implies}_F(e)$
  - disjunction of all minterms that imply e
- Naïve approach
  - generate all 2^n possible minterms
  - for each minterm m, use decision procedure to check *validity* of each implication $m \Rightarrow e$

Fast Predicate Abstraction

- Idea:
  - compute set of minterms m that imply e directly via theorem prover data structures

Consider Equalities

- Example
  - F = { a=b, b=c, c=d }
  - $\text{Implies}_F(a=d)$
- Equality graph induced by F:



$\text{Implies}_F(a=d) = (a=b) \wedge (b=c) \wedge (c=d)$

Efficient Implementation of Implies(e)

- Graph representation of
  - equalities
  - uninterpreted function symbols (*p = *q)
  - inequalities (x < c)
- Computation of good minterms via reachability query

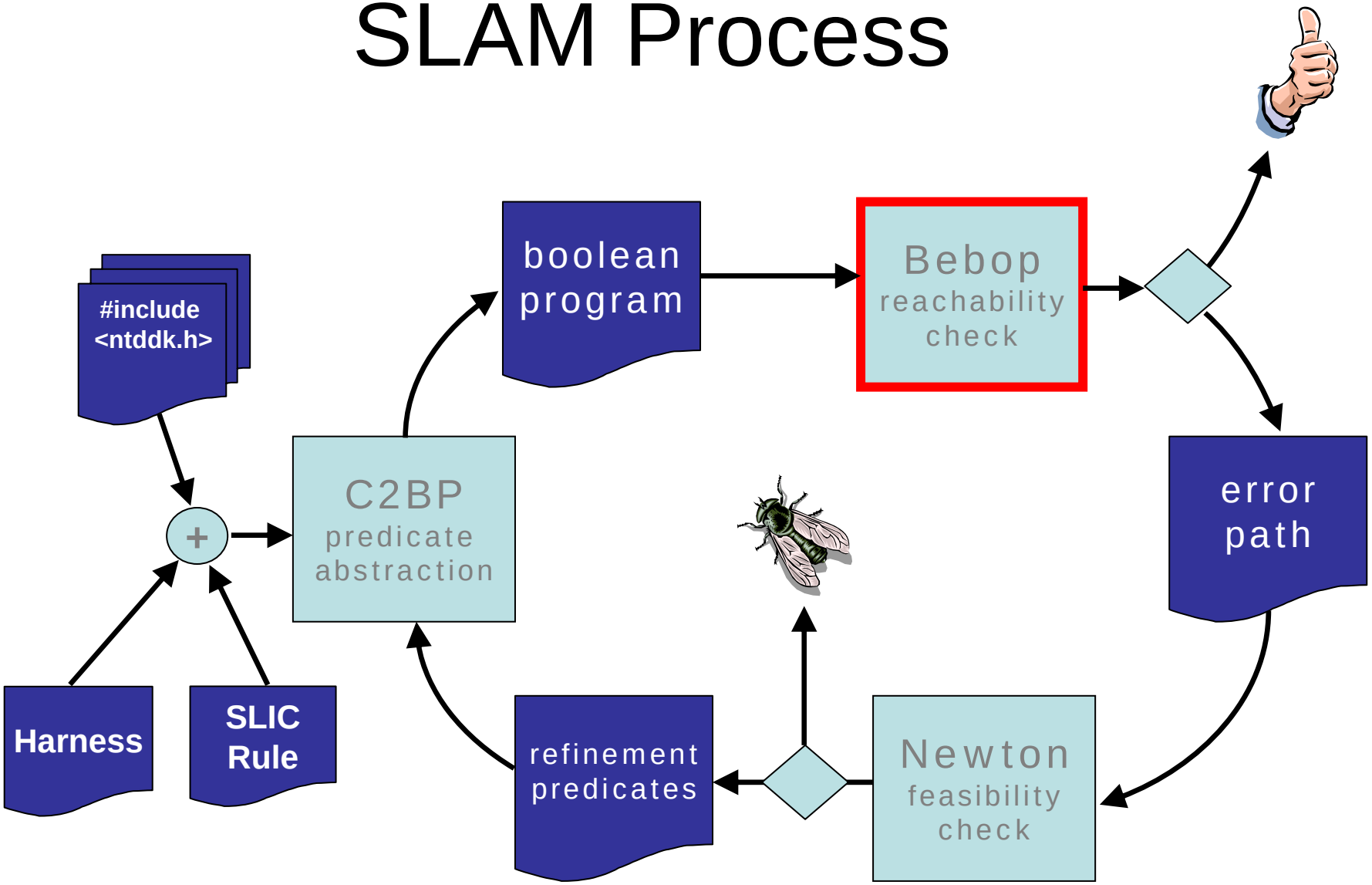
The Rest of the Story

- For predicates not in theory (i.e., x<y+c), call theorem prover as before
  - limit size of minterms for efficiency
- Use Das/Dill refinement to deal with approximation introduced by heuristics

c2bp Precision

- For program P and F = {e_1,...,e_n}, there exist two “ideal” abstractions:
  - Boolean(P,F) : most precise abstraction
  - Cartesian(P,F) : less precise abstraction, where each boolean variable is updated independently
  - [See Ball-Podelski-Rajamani, TACAS 00]
- Theory:
  - with an “ideal” theorem prover, c2bp can compute Cartesian(P,F)
- Practice:
  - c2bp computes a less precise abstraction than Cartesian(P,F)
  - we use Das/Dill’s technique to incrementally improve precision
  - the combination of c2bp + Das/Dill can compute Boolean(P,F)

SLAM Process



Bebop

- Symbolic reachability for boolean programs
- Based on CFL reachability
  - [Sharir-Pnueli 81]
  - [Reps-Sagiv-Horwitz 95]
- Uses BDDs to represent sets of paths

BDDs Summarize a Set of Paths

- Canonical representation of
  - boolean functions
  - set of (fixed-length) bitvectors
  - binary relations over finite domains
- Efficient algorithms for common analysis operations
  - transfer function
  - join/meet
  - subsumption test

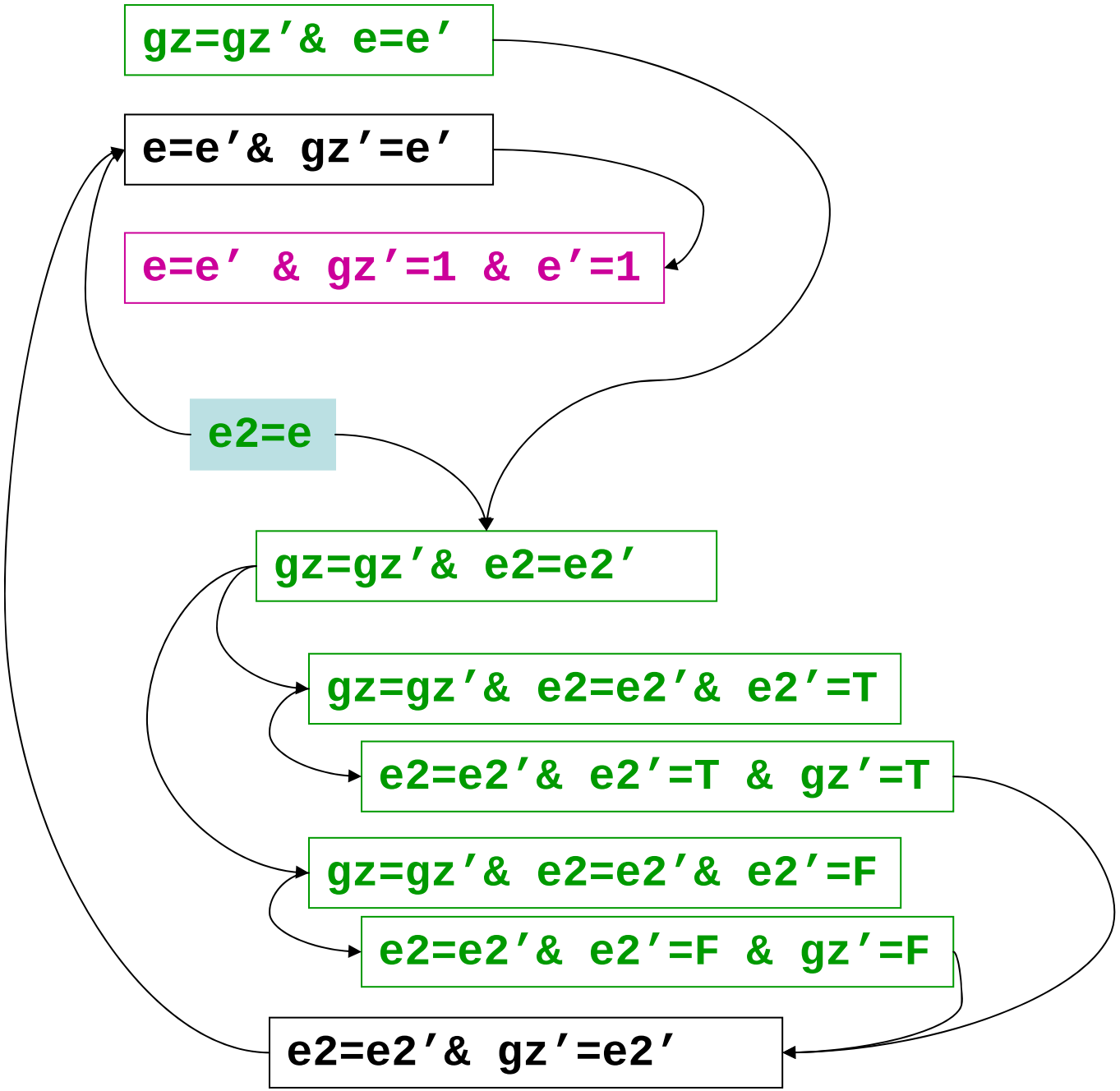
```
void cmp ( e2 ) {
[5]Goto L1, L2
   [6]L1: assume( e2 );
   [7] gz := T; goto L3;

   [8]L2: assume( !e2 );
   [9]gz := F; goto L3

[10] L3: return;
}
```

BDD at line [10] of cmp:

e2=e2′ & gz′=e2′

Read: “cmp leaves e2 unchanged and sets gz to have the same final value as e2”

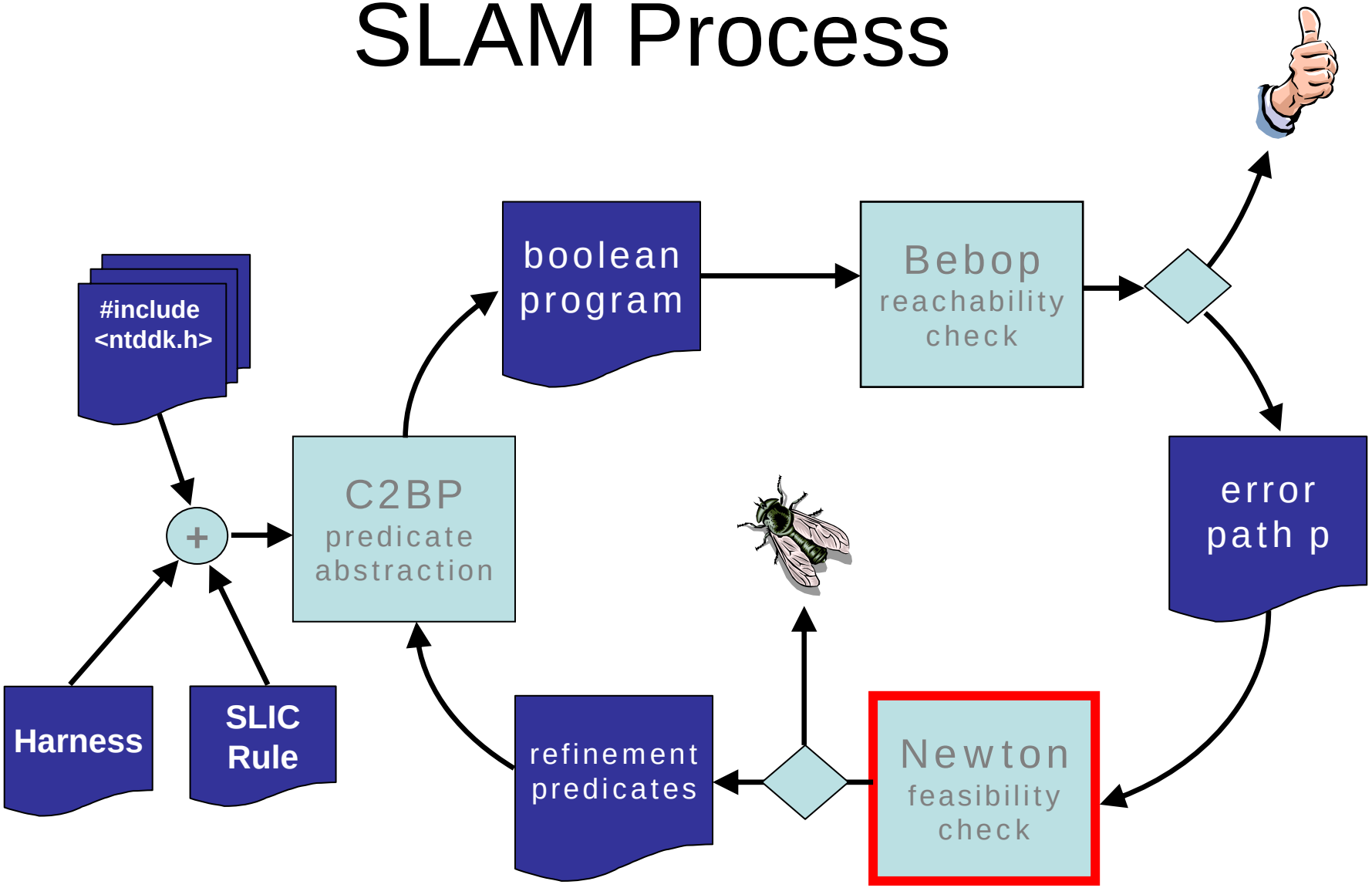
```
decl gz ;
main( e ) {

 [1] cmp( e );

 [2] assume( gz );

 [3] assume( !e );

 [4] assert(F);
}

void cmp ( e2 ) {
[5]Goto L1, L2

 [6]L1: assume( e2 );

 [7] gz := T; goto L3;

 [8]L2: assume( !e2 );

 [9]gz := F; goto L3

[10] L3: return;
}
```



Bebop: Summary

- Explicit representation of CFG
- Implicit representation of sets of paths via BDDs
- Generation of hierarchical error traces
- Complexity: $O(E * 2^{O(N)})$
  - E is the size of the CFG
  - N is the max. number of variables in scope

SLAM Process



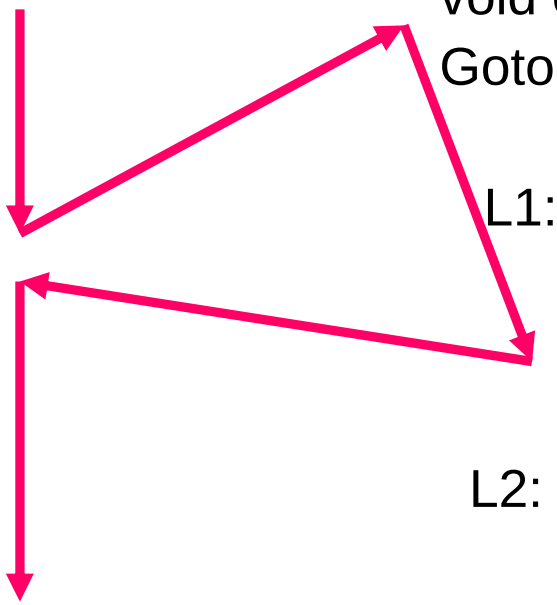
Newton

- Bebop produces potential error path p
- Is p a feasible path of the C program?
  - Yes: found an error
  - No: find predicates that explain the infeasibility
- Symbolic execution of a program path
  - feasibility testing with theorem prover

Newton

- Execute path symbolically
- Check conditions for inconsistency using theorem prover (satisfiability)
- After detecting inconsistency:
  - minimize inconsistent conditions
  - traverse dependencies
  - obtain predicates

```
int g;

main(int x, int y){

  cmp(x, y);

  assume(!g);
  assume(x != y)
  assert(0);
}
```



```
void cmp (int a , int b) {
Goto L1, L2

   L1: assume(a=b);
       g := 0;
       return;

   L2: assume(a!=b);
       g := 1;
       return;
}
```

```
int g;

main(int x, int y){

  cmp(x, y);

  assume(!g);
  assume(x != y)
  assert(0);
}
```

Global:

main:

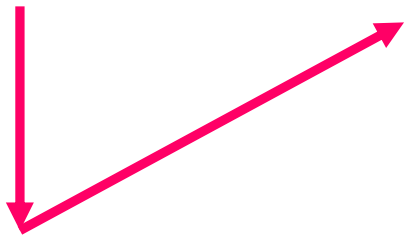
(1) x: X

(2) y: Y

```
void cmp (int a , int b) {
Goto L1, L2

   L1: assume(a=b);
       g := 0;
       return;

  L2: assume(a!=b);
       g := 1;
       return;
}
```

Conditions:

```
int g;

main(int x, int y){

  cmp(x, y);

  assume(!g);
  assume(x != y)
  assert(0);
}
```



```
void cmp (int a , int b) {
Goto L1, L2

  L1: assume(a=b);
      g := 0;
      return;

  L2: assume(a!=b);
      g := 1;
      return;
}
```

Global:

main:

(1) x: X

(2) y: Y

cmp:

(3) a: A

(4) b: B

Map:

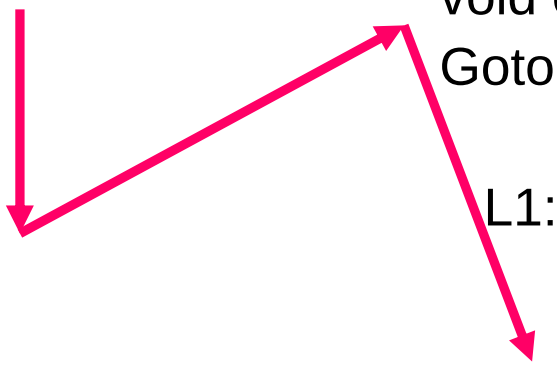
$X \rightarrow A$

$Y \rightarrow B$

Conditions:

```
int g;

main(int x, int y){

  cmp(x, y);

  assume(!g);
  assume(x != y)
  assert(0);
}
```

```
void cmp (int a , int b) {
Goto L1, L2

  L1: assume(a=b);
      g := 0;
      return;

  L2: assume(a!=b);
      g := 1;
      return;
}
```



Global:

(6) g: 0

main:

(1) x: X

(2) y: Y

cmp:

(3) a: A

(4) b: B

Map:

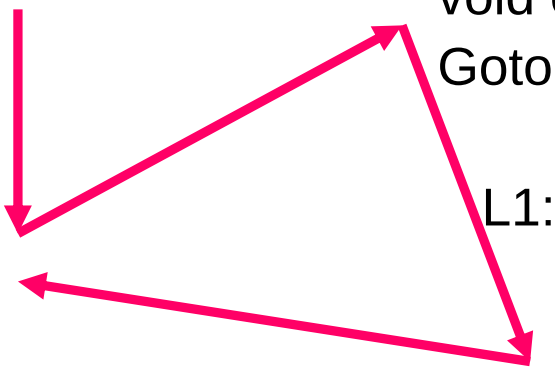
$X \rightarrow A$

$Y \rightarrow B$

Conditions:

(5) (A = B) [3, 4]

```
int g;

main(int x, int y){

  cmp(x, y);

  assume(!g);
  assume(x != y)
  assert(0);
}
```



```
void cmp (int a , int b) {
Goto L1, L2

   L1: assume(a=b);
       g := 0;
       return;

   L2: assume(a!=b);
       g := 1;
       return;
}
```

Global:

(6) g: 0

main:

(1) x: X

(2) y: Y

cmp:

(3) a: A

(4) b: B

Map:

$X \rightarrow A$

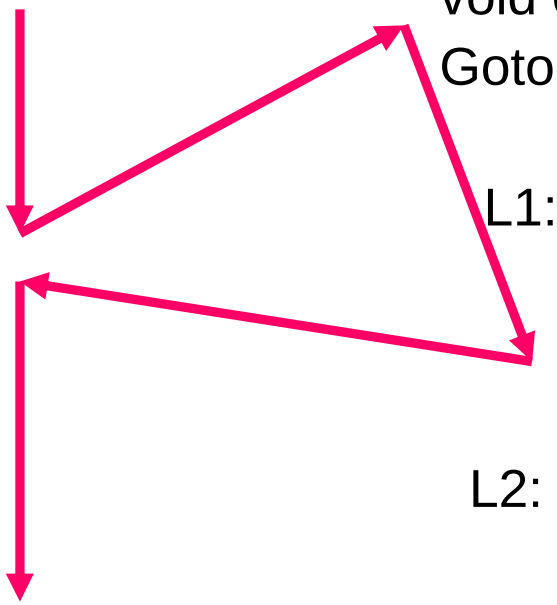
$Y \rightarrow B$

Conditions:

(5) (A = B) [3, 4]

(6) (X = Y) [5]

```
int g;

main(int x, int y){

  cmp(x, y);

  assume(!g);
  assume(x != y)
  assert(0);
}
```



```
void cmp (int a , int b) {
Goto L1, L2

   L1: assume(a=b);
       g := 0;
       return;

   L2: assume(a!=b);
       g := 1;
       return;
}
```

Global:

(6) g: 0

main:

(1) x: X

(2) y: Y

cmp:

(3) a: A

(4) b: B

Conditions:

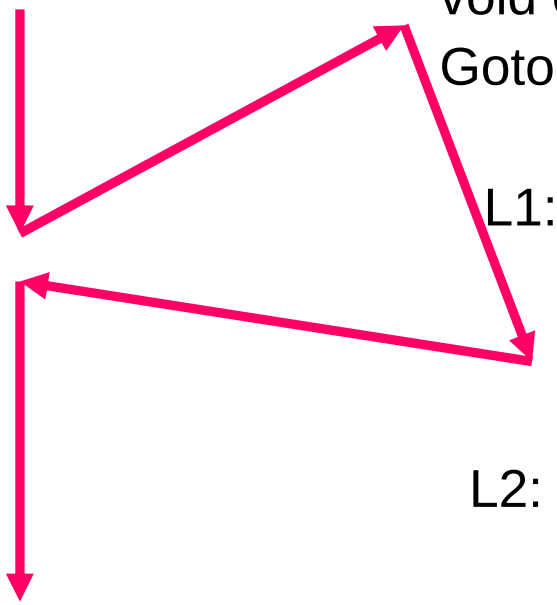
(5) (A = B) [3, 4]

(6) (X = Y) [5]

(7) (X != Y) [1, 2]

```
int g;

main(int x, int y){

  cmp(x, y);

  assume(!g);
  assume(x != y)
  assert(0);
}
```

```
void cmp (int a , int b) {
Goto L1, L2

   L1: assume(a=b);
       g := 0;
       return;

   L2: assume(a!=b);
       g := 1;
       return;
}
```



Global:

(6) g: 0

main:

(1) x: X

(2) y: Y

cmp:

(3) a: A

(4) b: B

Conditions:

(5) (A = B) [3, 4]

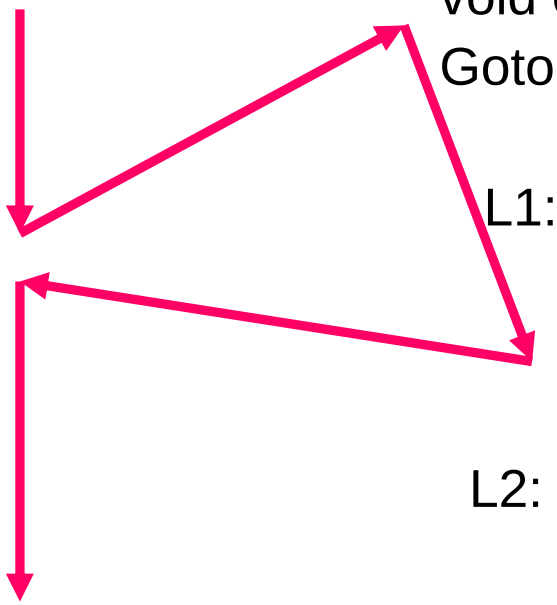
(6) (X = Y) [5]

(7) (X != Y) [1, 2]

Contradictory!

```
int g;

main(int x, int y){

  cmp(x, y);

  assume(!g);
  assume(x != y)
  assert(0);
}
```

```
void cmp (int a , int b) {
Goto L1, L2

   L1: assume(a=b);
       g := 0;
       return;

   L2: assume(a!=b);
       g := 1;
       return;
}
```



Global:

(6) g: 0

main:

(1) x: X

(2) y: Y

cmp:

(3) a: A

(4) b: B

Conditions:

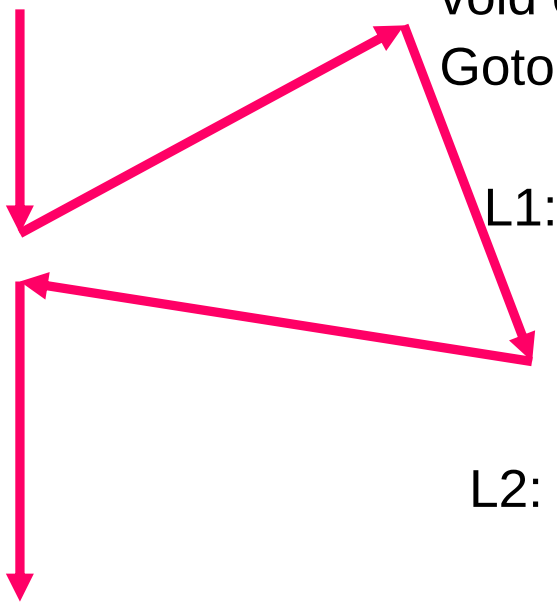
(5) (A = B) [3, 4]

(6) (X = Y) [5]

(7) (X != Y) [1, 2]

Contradictory!

```
int g;

main(int x, int y){

  cmp(x, y);

  assume(!g);
  assume(x != y)
  assert(0);
}
```



```
void cmp (int a , int b) {
Goto L1, L2

   L1: assume(a=b);
       g := 0;
       return;

    L2: assume(a!=b);
       g := 1;
       return;
}
```

Predicates after simplification:

{ x = y, a = b }

Part III: Advanced Topics

C-

Types	τ	::=	void | bool | int | ref τ
Expressions	**e**	::=	c | x | $\mathbf{e_1}$ op $\mathbf{e_2}$ | &x | *x
LExpression	ℓ	::=	x | *x
Declaration	**d**	::=	τ $x_1, x_2, \ldots, x_n$
Statements	**s**	::=	skip | goto $L_1, L_2 \ldots L_n$ | L: s
		|	assume(**e**)
		|	ℓ := **e**
		|	ℓ := f ($\mathbf{e_1}, \mathbf{e_2}, \ldots, \mathbf{e_n}$)
		|	return x
		|	$\mathbf{s_1}$; $\mathbf{s_2}$; ... ; $\mathbf{s_n}$
Procedures	**p**	::=	τ f (x_1: τ_1, x_2 : τ_2, ..., x_n : τ_n)
Program	**g**	::=	$\mathbf{d_1}$ $\mathbf{d_2}$ ... $\mathbf{d_n}$ $\mathbf{p_1}$ $\mathbf{p_2}$ ... $\mathbf{p_n}$

Pointers and SLAM

- With pointers, C supports call by reference
  - Strictly speaking, C supports only call by value
  - With pointers and the address-of operator, one can simulate call-by-reference
- Boolean programs support only call-by-value-result
  - SLAM mimics call-by-reference with call-by-value-result
- Extra complications:
  - address operator (&) in C
  - multiple levels of pointer dereference in C

What changes with pointers?

- C2bp
  - abstracting assignments
  - abstracting procedure returns
- Newton
  - simulation needs to handle pointer accesses
  - need to copy local heap across scopes to match Bebop's semantics
- Bebop
  - remains unchanged!

Assignments + Pointers

Statement in P:

*p := 3

Predicates in E:

{x=5}

Weakest Precondition:

WP(*p:=3 , x=5) = x=5

What if *p and x alias?

Correct Weakest Precondition:

(p=&x && 3==5) || (p!=&x && x=5)

We use Das's pointer analysis [PLDI 2000] to prune disjuncts representing infeasible alias scenarios.

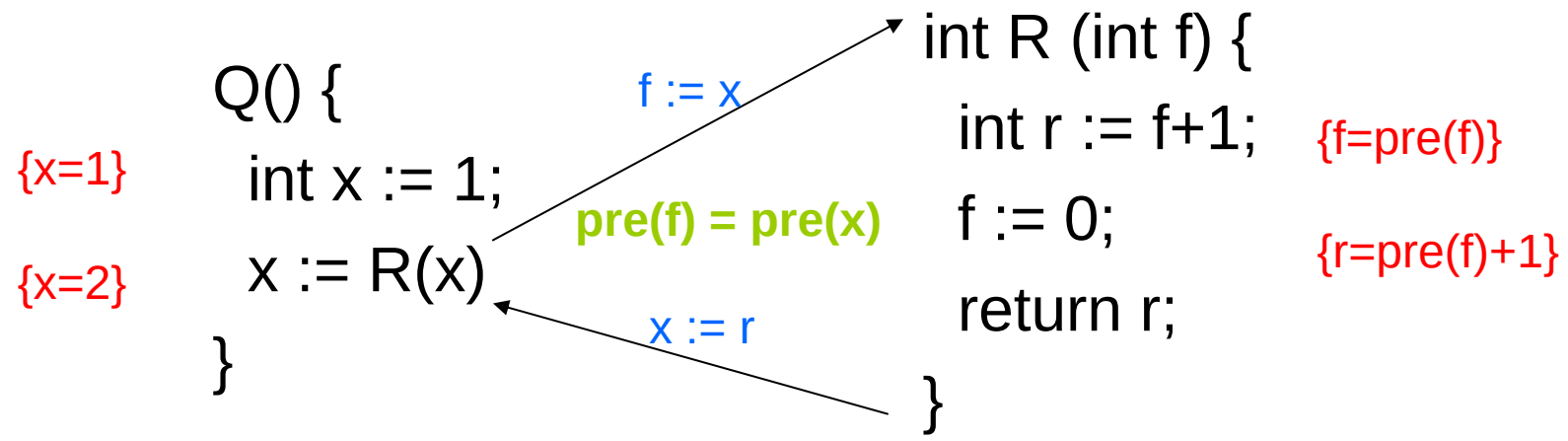
Abstracting Procedure Return

- Need to account for
  - lhs of procedure call
  - mixed predicates
  - side-effects of procedure
- Boolean programs support only call-by-value-result
  - c2bp models all side-effects using return processing

Abstracting Procedure Returns

- Let a be an actual at call-site P(…)
  - pre(a) = the value of a before transition to P
- Let f be a formal of procedure P
  - pre(f) = the value of f upon entry to P

predicate | call/return relation | call/return assign



WP(f:=x, f=pre(f)) = x=pre(f)

x=pre(f) is true at the call to R

WP(x:=r, x=2) = r=2

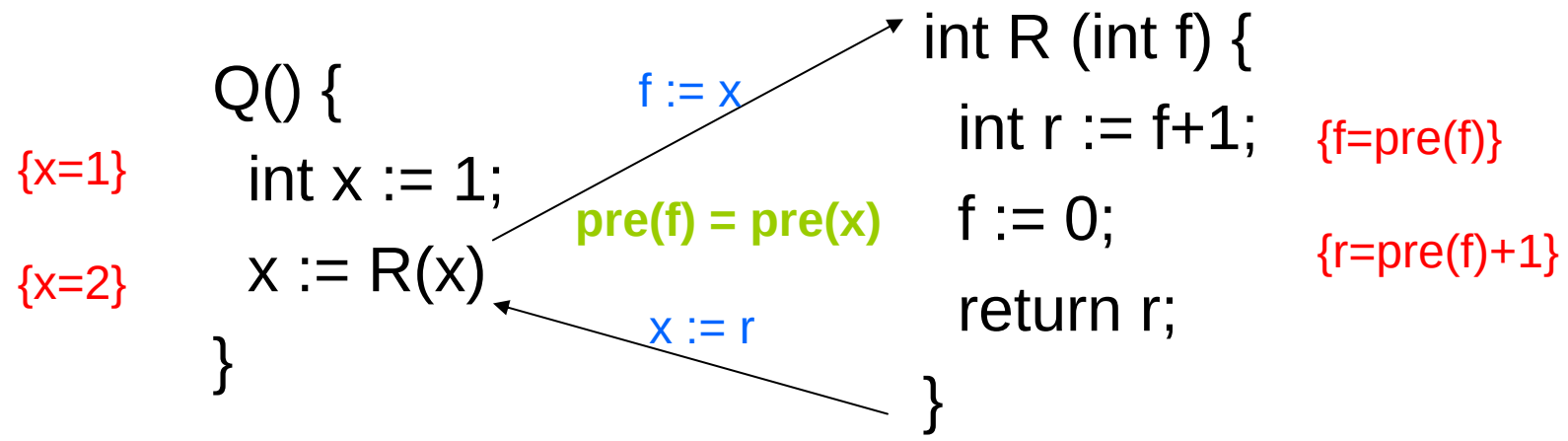
pre(f)=pre(x) and pre(x)=1 **and** r=pre(f)+1 **implies** r=2

{x=1} s

```
Q() {
 {x=1},{x=2} := T,F;
 s := R(T);
 {x=2} := s & {x=1};
}
```

```
bool R ( {f=pre(f)} ) {
  {r=pre(f)+1} := {f=pre(f)};
  {f=pre(f)} := *;
  return {r=pre(f)+1};
}
```

predicate | call/return relation | call/return assign



WP(f:=x, f=pre(f)) = x=pre(f)

x=pre(f) is true at the call to R

WP(x:=r, x=2) = r=2

pre(f)=pre(x) and pre(x)=1 **and** r=pre(f)+1 **implies** r=2

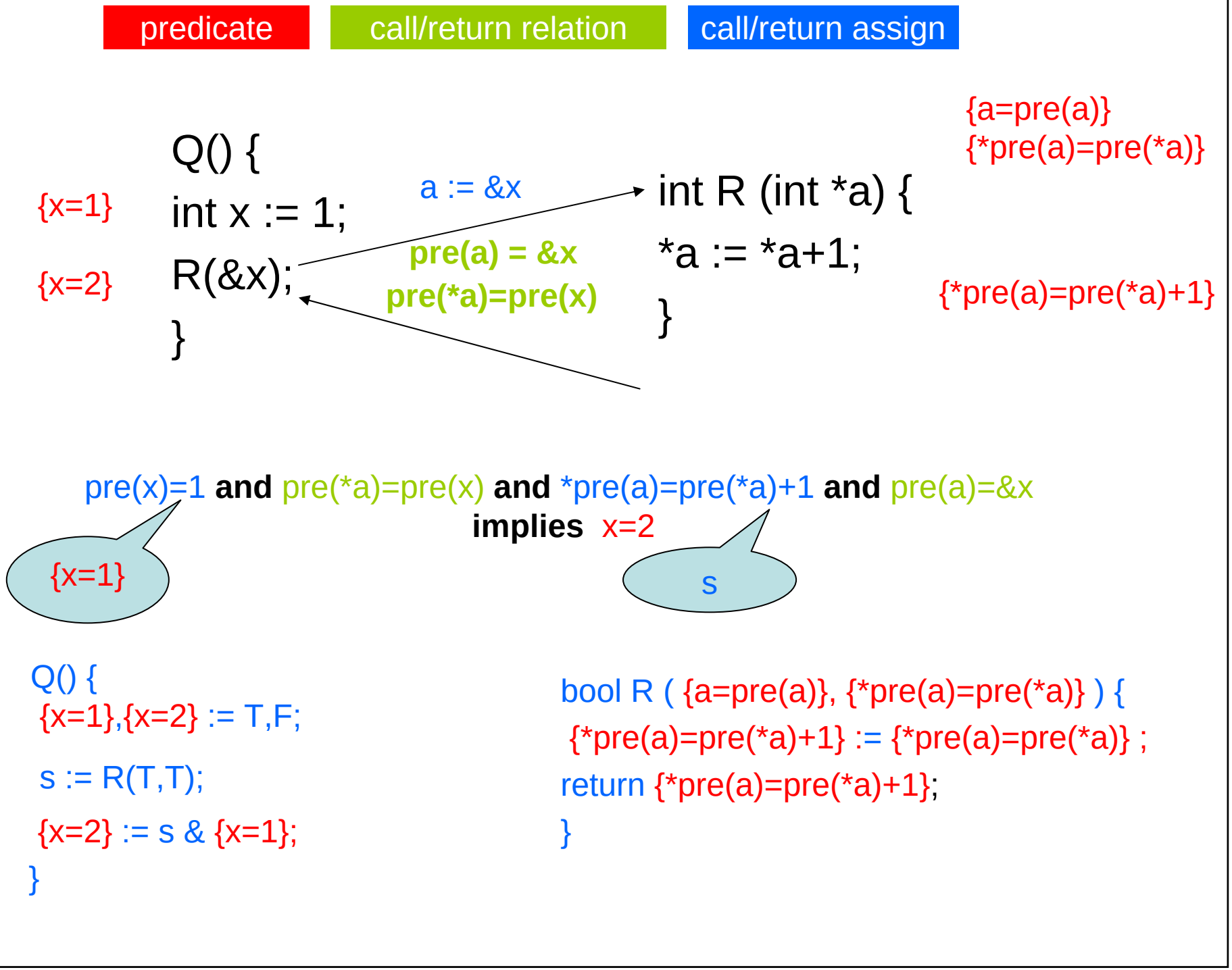
{x=1} s

```
Q() {
 {x=1},{x=2} := T,F;
 s := R(T);
 {x=1}, {x=2} := *, s & {x=1};
}
```

```
bool R ( {f=pre(f)} ) {
  {r=pre(f)+1} := {f=pre(f)};
  {f=pre(f)} := *;
  return {r=pre(f)+1};
}
```

Extending Pre-states

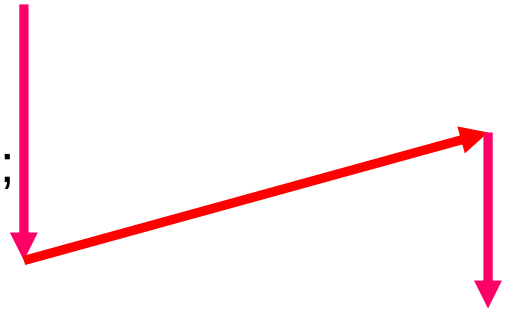
- Suppose formal parameter is a pointer
  - eg. P(int *f)
- pre(*f)
  - value of *f upon entry to P
  - can’t change during P
- * pre(f)
  - value of dereference of pre(f)
  - can change during P


predicate
call/return relation
call/return assign
{a=pre(a)}
{*pre(a)=pre(*a)}
Q() {
int x := 1;
R(&x);
}
{x=1}
{x=2}
a := &x
pre(a) = &x
pre(*a)=pre(x)
int R (int *a) {
*a := *a+1;
}
{*pre(a)=pre(*a)+1}
pre(x)=1 and pre(*a)=pre(x) and *pre(a)=pre(*a)+1 and pre(a)=&x
implies x=2
{x=1}
s
Q() {
{x=1},{x=2} := T,F;
s := R(T,T);
{x=2} := s & {x=1};
}
bool R ({a=pre(a)}, {*pre(a)=pre(*a)}) {
{*pre(a)=pre(*a)+1} := {*pre(a)=pre(*a)} ;
return {*pre(a)=pre(*a)+1};
}

Newton: what changes with pointers?

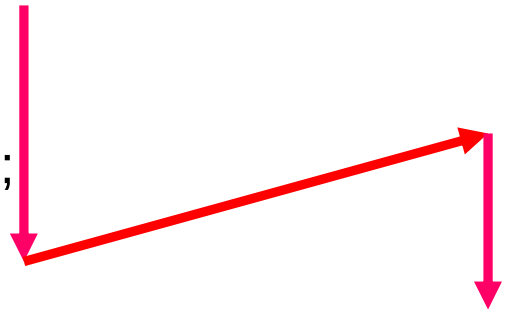
- Simulation needs to handle pointer accesses
- Need to copy local heap across scopes to match Bebop’s semantics

```
main(int  *x){

  assume(*x < 5);
  foo(x);

}
```



```
void foo (int *a) {
    assume(*a > 5);
    assert(0);
}
```

```
main(int  *x){

  assume(*x < 5);
  foo(x);

}
```



```
void foo (int *a) {
    assume(*a > 5);
    assert(0);
}
```

Predicates after simplification:

*x < 5 , *a < 5

main:

(1) x: X

(2) *X: Y [1]

foo:

(3) a: A

(4) *A: B [3]

Contradictory!

Conditions:

(5) (Y< 5) [1,2]

(6) (B < 5) [3,4,5]

(7) (B > 5) [3,4]

What worked well?

- Specific domain problem
- Safety properties
- Shoulders & synergies
- Separation of concerns
- Summer interns & visitors
- Windows partners

Further Reading

See papers, slides from:

http://research.microsoft.com/slam