Software Model Checking

Aditya V. Nori
Microsoft Research India

Thanks to Tom Ball & Sriram Rajamani for material from their lectures

Software Model Checking via Abstraction Refinement

- Model checking = exhaustive exploration of state space
- Challenge: realistic software has a huge state space?
- Approach: Abstraction-refinement
  - Construct an abstraction
    - a "simpler model" of the software that only contains the variables and relationships that are important to the property being checked
  - Model check the abstraction
    - easier because state space of the abstraction is smaller
  - Refine the abstraction
    - to reduce false errors

SLAM – Software Model Checking

SLAM models

- boolean programs: a new model for software

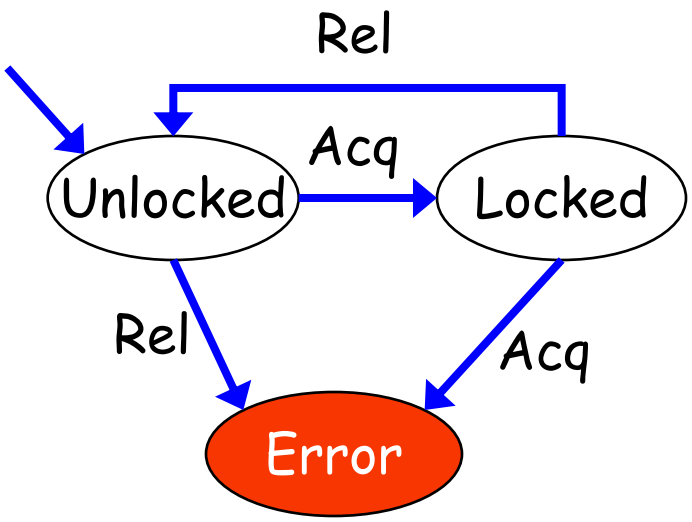
SLAM components

- model creation (c2bp)
- model checking (bebop)
- model refinement (newton)

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

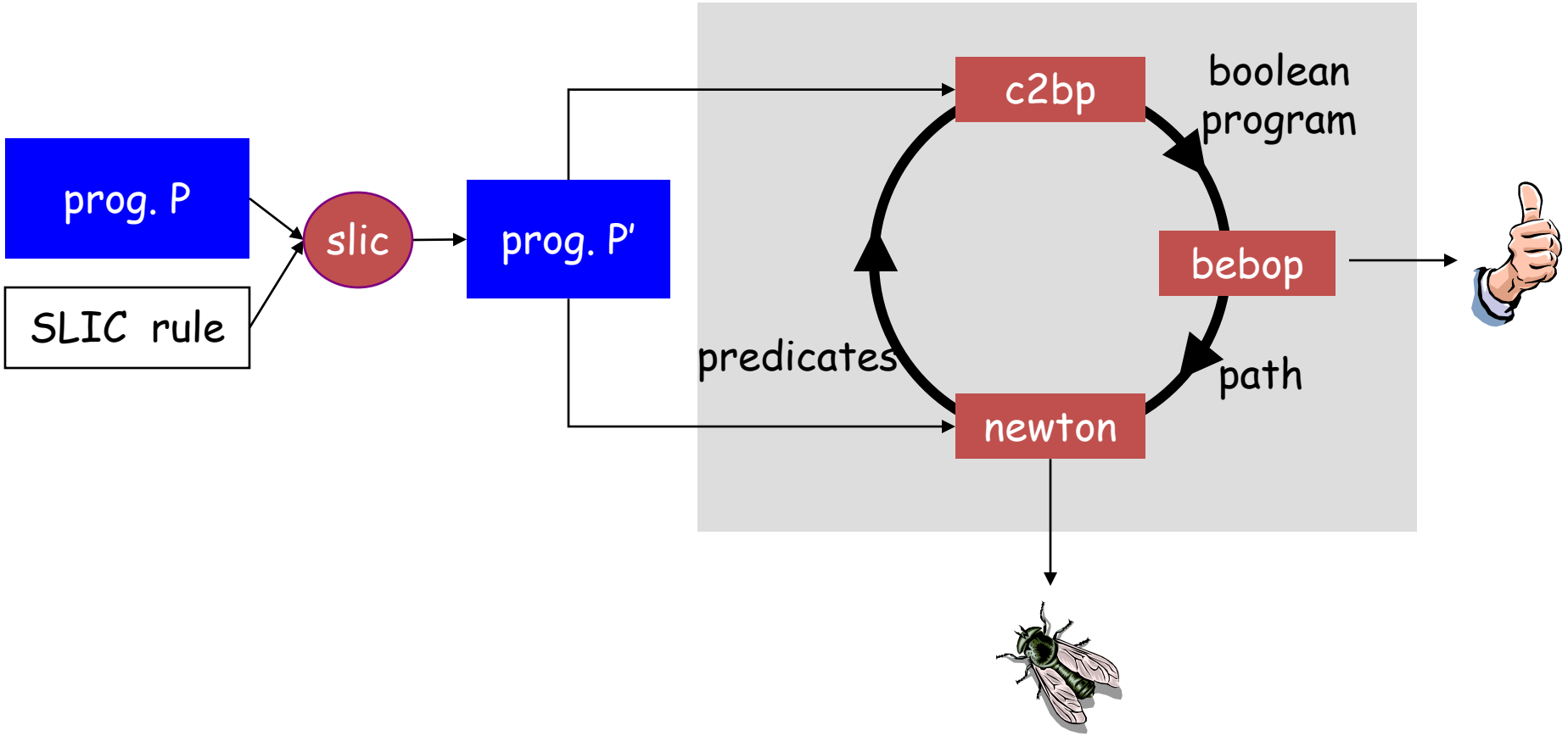
State Machine for Locking



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;
}
```

The SLAM Process



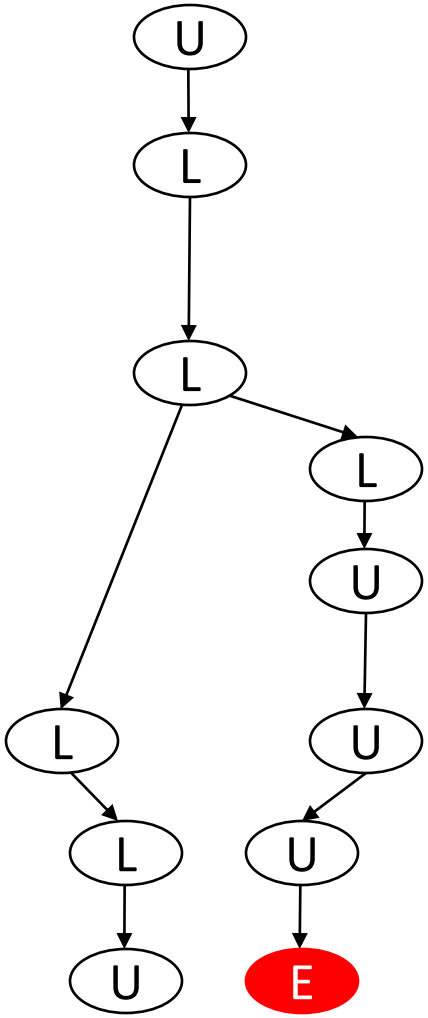
Example

Does this code obey the locking rule?

```
do {
    KeAcquireSpinLock();

    nPacketsOld = nPackets;

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

KeReleaseSpinLock();
```

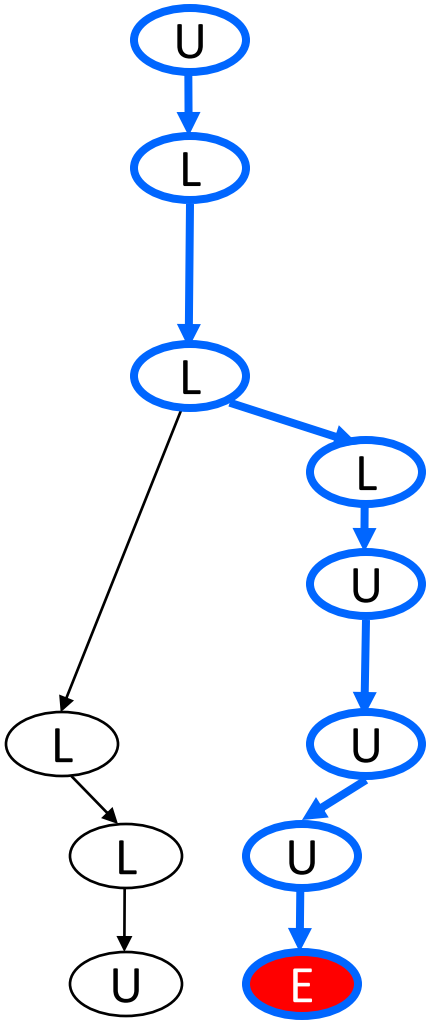
Example

Model checking boolean program (bebop)




```
do {
    KeAcquireSpinLock();

    if(*){
        KeReleaseSpinLock();
    }
} while (*);

KeReleaseSpinLock();
```

Example



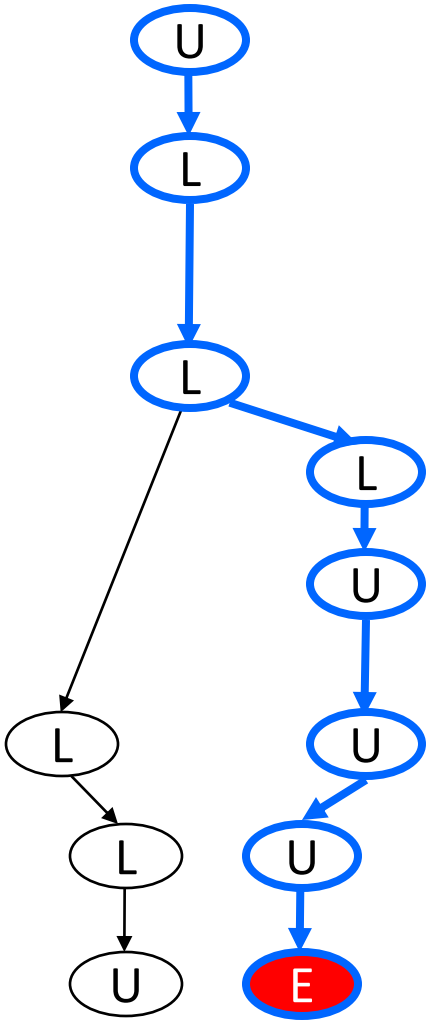

```
do {
    KeAcquireSpinLock();

    nPacketsOld = nPackets;

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

KeReleaseSpinLock();
```

Is error path feasible in C program? (newton)

Example

b : (nPacketsOld == nPackets)

Add new predicate to boolean program (c2bp)

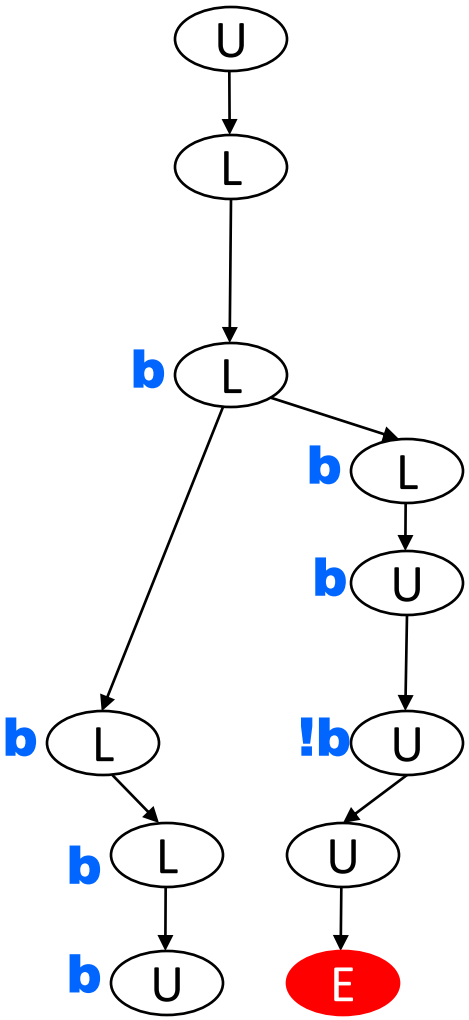



```
do {
    KeAcquireSpinLock();

    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)

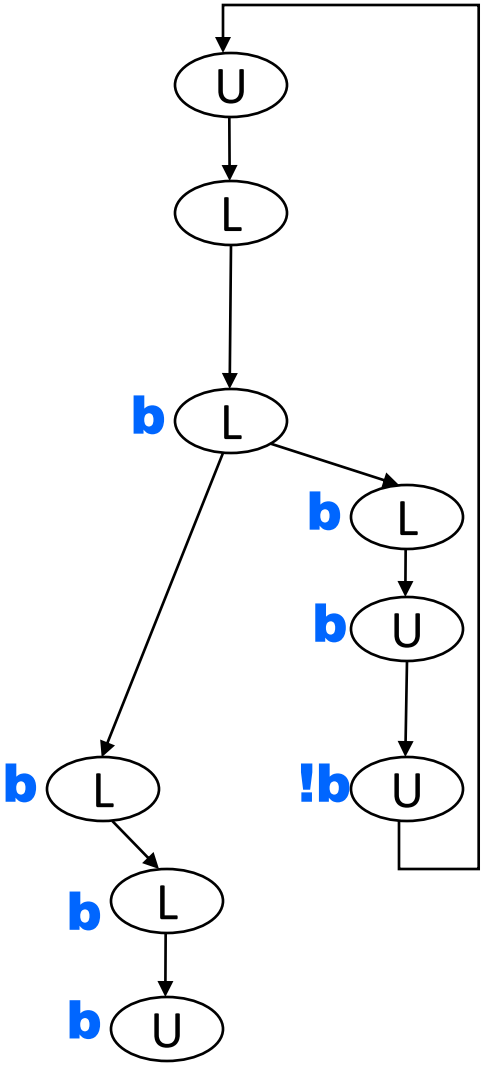



```
do {
    KeAcquireSpinLock();

    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();
```

Observations about SLAM

- Automatic discovery of invariants
  - driven by property and a finite set of (false) execution paths
  - predicates are ***not*** invariants, but observations
  - abstraction + model checking computes inductive invariants (boolean combinations of observations)
- A hybrid dynamic/static analysis
  - newton executes path through C code symbolically
  - c2bp+bebop explore all paths through abstraction
- A new form of program slicing
  - program code and data not relevant to property are dropped
  - non-determinism allows slices to have more behaviors

SLAM internals
(with some simplifications)

C-

Types	τ	::=	void | bool | int | ref τ
Expressions	e	::=	c | x | e_1 op 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)
		|	$l = e$
		|	$l = f(e_1, e_2, \ldots, e_n)$
		|	return x
		|	$s_1 ; s_2 ; \ldots ; s_n$
Procedures	p	::=	τ f (x_1: τ_1, x_2 : τ_2, …, x_n : τ_n) d s
Program	g	::=	d_1 d_2 … d_n p_1 p_2 … p_n

C--

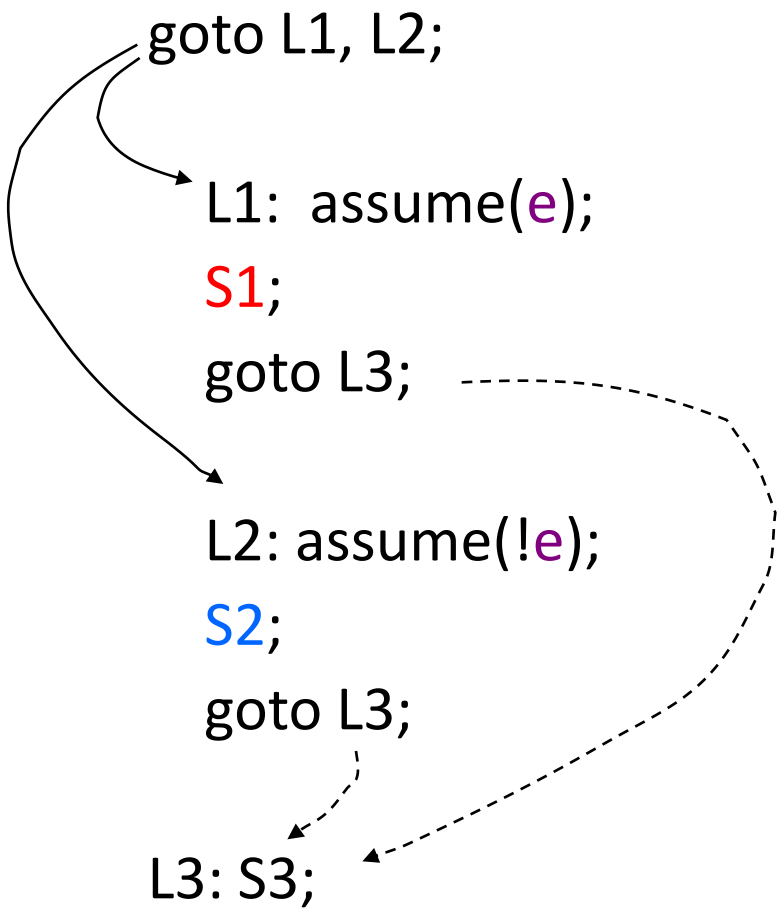
Types	τ	::=	void | bool | int
Expressions	e	::=	c | x | e_1 op e_2
LExpression	l	::=	x
Declaration	d	::=	τ $x_1, x_2, \ldots, x_n$
Statements	s	::=	skip | goto $L_1, L_2 \ldots L_n$ | L: s
		|	assume(e)
		|	l = e
		|	f ($e_1, e_2, \ldots, e_n$)
		|	return
		|	s_1 ; s_2 ; … ; s_n
Procedures	p	::=	f (x_1: τ_1, x_2 : $\tau_2, \ldots, x_n$: τ_n) d s
Program	g	::=	d_1 d_2 … d_n p_1 p_2 … p_n

BP

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

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;
}
```

c2bp: Predicate Abstraction for C Programs

Given

- P : a C program
- F = $\{e_1,\ldots,e_n\}$
  - 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 $\{b_1,\ldots,b_n\}$ to match $\{e_1,\ldots,e_n\}$
- properties true of B(P,F) are true of 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);
}
```

```
decl {g==0} ;

main( {x==y} ) {

  cmp( {x==y} );

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

Preds: {x==y}

{g==0}

{a==b}

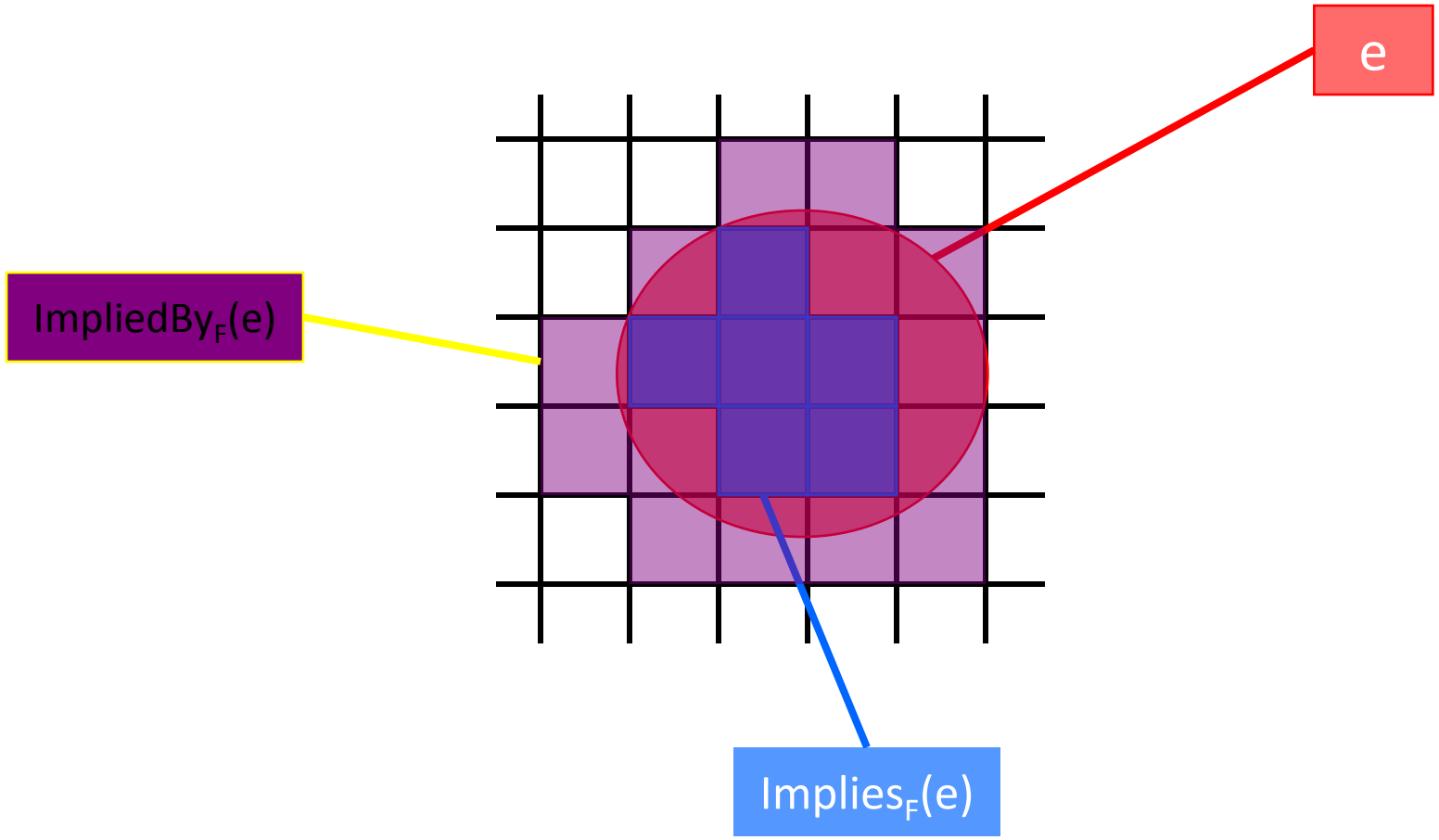
```
void equal (int a , int b) {
  goto L1, L2

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

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

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

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

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

Abstracting Expressions via F

- $F = \{ e_1, \ldots, e_n \}$
- $\text{Implies}_F(e)$
  - *weakest* boolean function over F *that implies* e
- $\text{ImpliedBy}_F(e)$
  - *strongest* boolean function over F *implied by* e
  - $\text{ImpliedBy}_F(e) = \neg\text{Implies}_F(\neg e)$

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



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

- $F = \{ e_1, \ldots, e_n \}$
- Minterm: $m = d_1 \wedge \ \ldots \wedge d_n$
  - where $d_i = e_i$ or $d_i = \neg e_i$
- $\text{Implies}_F(e)$
  - 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 $m \Rightarrow e$
- Many optimizations possible

Abstracting Assignments

Suppose you are given an assignment s

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

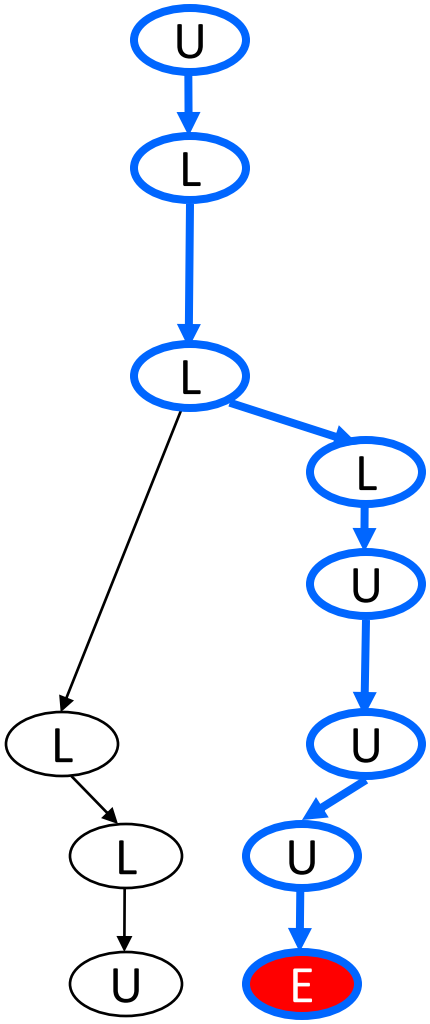
- if $\text{Implies}_F(WP(s, !e_i))$ is true before s then
  - e_i is false after s

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

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();
```

Assignment Example

Statement in P:

y = y+1;

Predicates in F:

{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 in B:

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

Abstracting Assumes

- WP(assume(e) , Q) = $e \Rightarrow Q$

- assume(e) is abstracted to:

  assume($\text{ImpliedBy}_F(e)$)

- Example:

  F = {x==2, x<5}

  assume(x<2) is abstracted to:

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

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

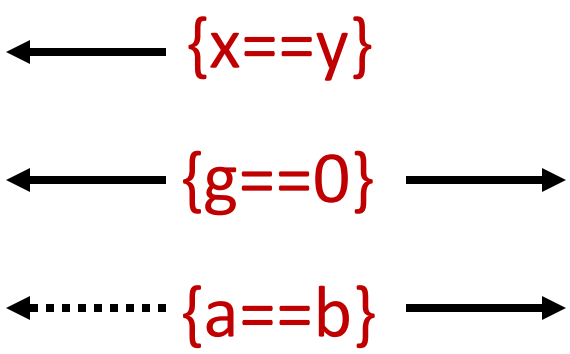
- Procedure call
  - a sequence of assignments from actuals to formals
  - see assignment abstraction

- 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

```
int g;

main(int x, int y){

  cmp(x, y);

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

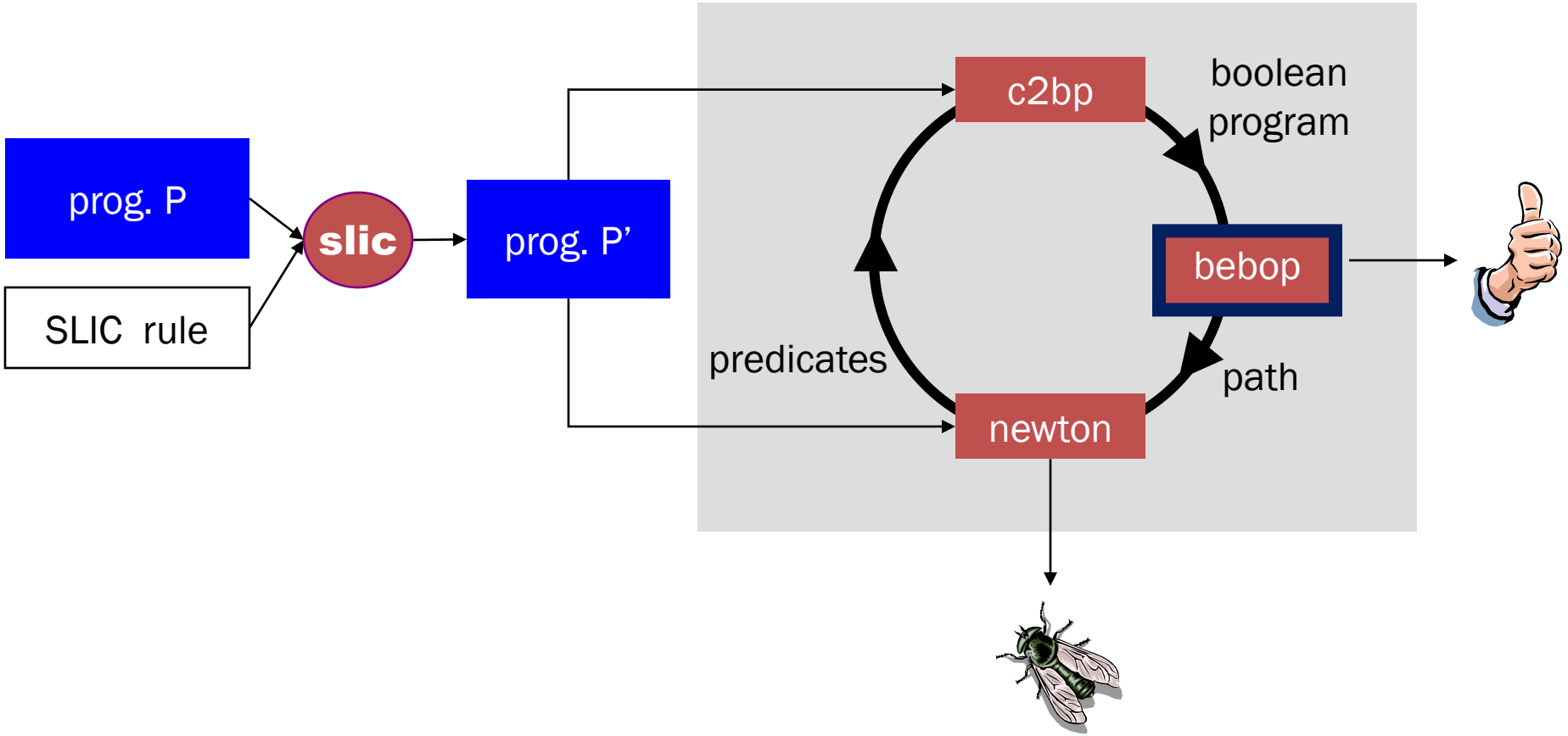
```
void cmp (int a , int b) {
Goto L1, L2

   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;
}
```

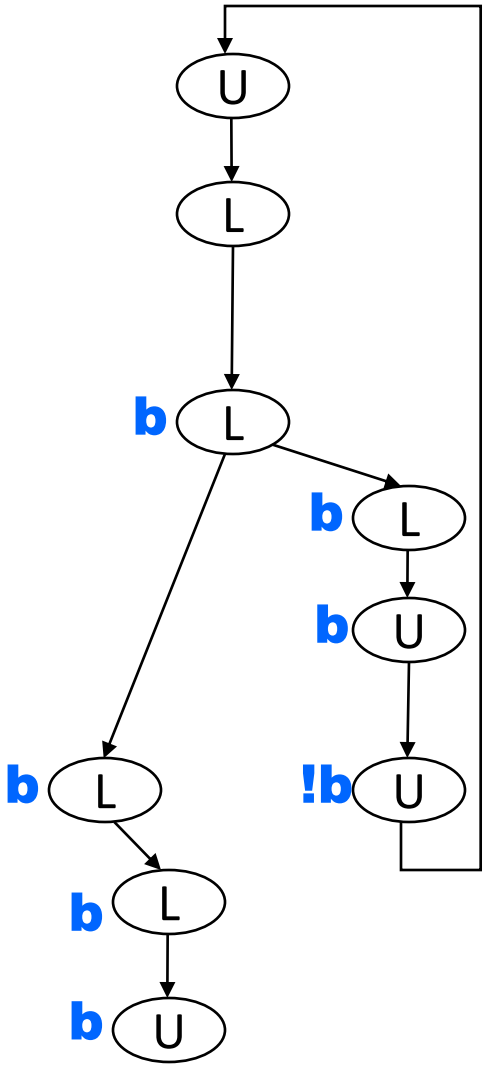
The SLAM Process



Bebop

- Model checker for boolean programs
- Based on CFL reachability
- Explicit representation of CFG
- Implicit representation of path edges and summary edges
- Generation of hierarchical error traces

Bebop: single procedure case




```
do {
    KeAcquireSpinLock();

    b = true;

    if(*){

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

KeReleaseSpinLock();
```

Bebop: interprocedural case

- Based on CFL reachability or Sharir-Pnueli's "functional" approach

- Key idea: instead of computing fixpoints over "states", compute fixpoints over two kinds of "edges":
  - Path edges
  - Summary edges

Domains

$g \in Global$

$l \in Local$

$f \in Frame$

$s \in Stack = Frame^*$

$State = Global \times Local \times Stack$

Transitions:

$T \subseteq (Global \times Local) \times (Global \times Local)$

$T^+ \subseteq Local \times (Local \times Frame)$

$T^- \subseteq (Local \times Frame) \times Local$

Transition Relation

$g \in$ Global
$l \in$ Local
$f \in$ Frame
$s \in$ Stack = Frame*
State = Global X Local X Stack

Transitions:
$T \subseteq$ (Global X Local) X (Global X Local)
$T^+ \subseteq$ Local X (Local X Frame)
$T^- \subseteq$ (Local X Frame) X Local

STEP

$$\frac{T(g, l, g', l')}{(g,l,s) \rightarrow (g',l', s)}$$

PUSH

$$\frac{T^+(l, l', f)}{(g,l,s) \rightarrow (g',l', s.f)}$$

POP

$$\frac{T^-(l, f, l'')}{(g,l,s.f) \rightarrow (g',l', s)}$$

Naïve model checking

Start with initial state: (g_0, l_0, ε)

Find all the states s such that
$(g_0, l_0, \varepsilon) \rightarrow^* s$

Even if Globals and Locals are finite, the set of reachable states could be infinite since the stack can grow infinite

No guarantee for termination

Still, assertion checking is decidable

Need to use a different algorithm (CFL reachability)

$$\frac{\text{STEP}\quad T(g, l, g', l')}{(g,l,s) \rightarrow (g',l', s)}$$

$$\frac{\text{PUSH}\quad T^{+}(l, l', f)}{(g,l,s) \rightarrow (g',l', s.f)}$$

$$\frac{\text{POP}\quad T^{-}(l, f, l'')}{(g,l,s.f) \rightarrow (g',l', s)}$$

CFL reachability

$P \subseteq (Global \times Local) \times (Global \times Local)$

$Sum \subseteq (Global \times Local) \times Frame \times (Global \times Local)$

CFL-INIT

$$\frac{}{P(g_0, l_0, g_0, l_0)}$$

CFL-STEP

$$\frac{P(g_1, l_1, g_2, l_2) \quad T(g_2, l_2, g_3, l_3)}{P(g_1, l_1, g_3, l_3)}$$

CFL-SUM

$$\frac{P(g_1, l_1, g_2, l_2) \quad T^{-}(l_2, f, l_3)}{Sum(g_1, l_1, f, g_2, l_3)}$$

CFL-PUSH

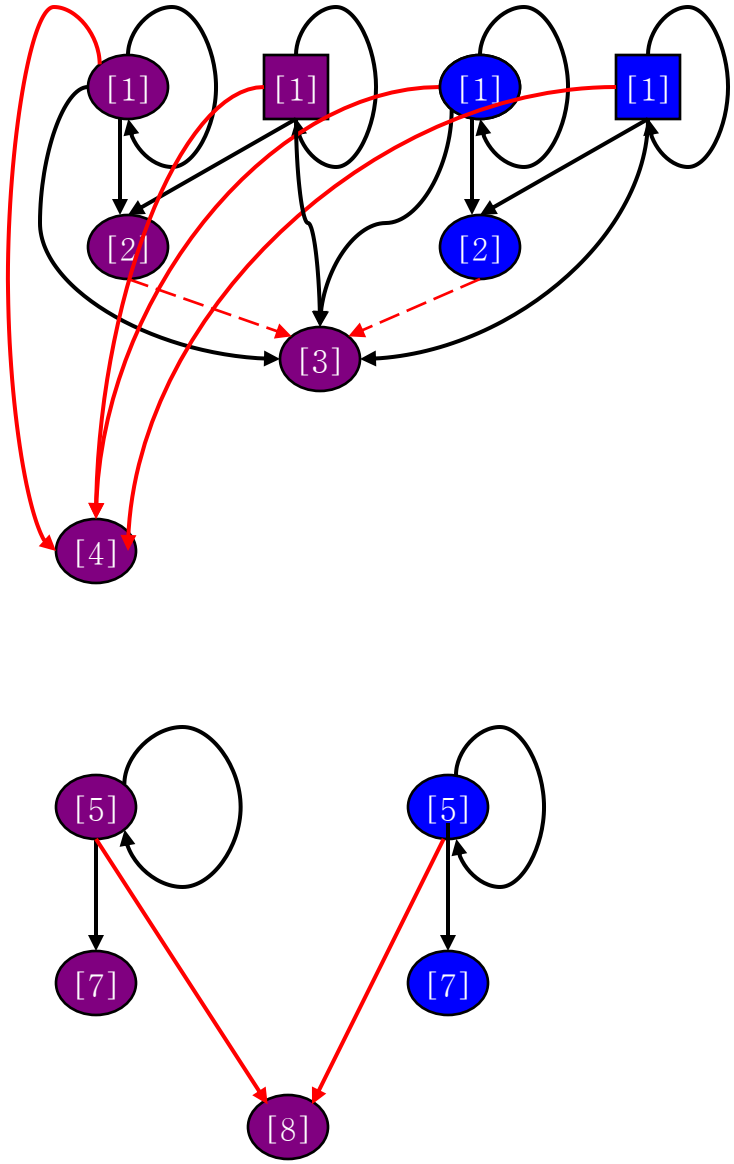
$$\frac{P(g_1, l_1, g_2, l_2) \quad T^{+}(l_2, l_3, f)}{P(g_2, l_3, g_2, l_3)}$$

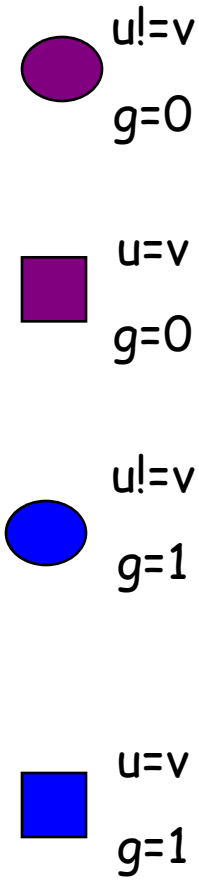
CFL-POP

$$\frac{P(g_1, l_1, g_2, l_2) \quad T^{+}(l_2, l_3, f) \quad Sum(g_2, l_3, f, g_3, l_4)}{P(g_1, l_1, g_3, l_4)}$$

[Sharir-Pnueli 81] [Reps-Sagiv-Horwitz 95]

```
decl g;
void main()
decl u,v;

 [1] u := !v;

 [2] equal(u,v);

 [3] if (g) then
              R:  skip;
     fi
 [4] return;

void equal(a, b)

 [5]  if (a = b) then
 [6]          g := 1;
      else
 [7]          g := 0;
      fi
 [8]  return;
```

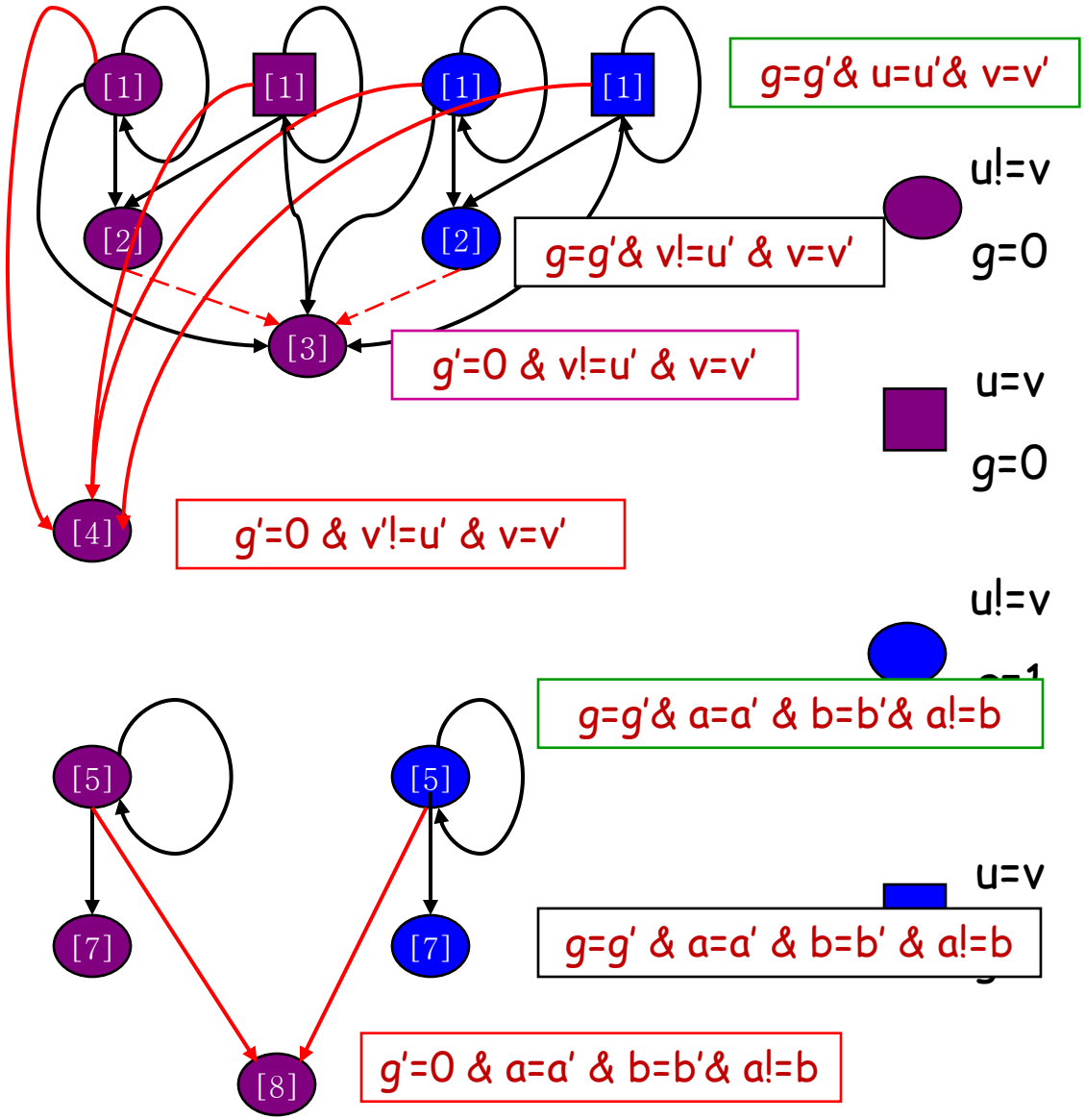




Symbolic CFL reachability

- Partition path edges by their "target"
  - PE(v) = { <d1,d2> | <entry,d1> $\rightarrow$ <v,d2> }
- What is <d1,d2> for boolean programs?
  - A bit-vector!
- What is PE(v)?
  - A set of bit-vectors
- Use a BDD (attached to v) to represent PE(v)

```
decl g;
void main()
decl u,v;

 [1] u := !v;

 [2] equal(u,v);

 [3] if (g) then
              R:  skip;
     fi
 [4] return;

void equal(a, b)

 [5]  if (a = b) then
 [6]           g := 1;
       else
 [7]           g := 0;
      fi
 [8]  return;
```



Bebop: summary

- Explicit representation of CFG
- Implicit representation of path edges and summary edges
- 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