Software Model Checking:

predicate discovery

Thomas Ball

Testing, Verification and Measurement

Microsoft Research

Counterexample-driven Refinement

```
F := {};
loop
    $T^\#$ := predAbs(T,F)
    if unsafe $\notin$ lfp($T^\#$, init) then
        return SUCCESS
    else
        find min k s.t. unsafe $\in$ ${T^\#}^k$(init)
        if unsafe $\in$ $T^k$ (init) then
            return FAILURE
        else
            find G s.t. unsafe $\notin$ ${U^\#}^k$(init)
                where $U^\#$ = predAbs(T,G)
            F := F $\cup$ G
forever
```

Path Feasibility and Predicate Discovery

- Given an error path p in boolean program
  - is p a feasible path of the corresponding C program?
    - Yes: found an error
    - No: find predicates that explain the infeasibility

- Goals of predicate discovery
  - want to rule out many infeasible paths at once
  - predicates should be properly scoped

Predicate Discovery Goals

- Rule out the given spurious counterexample
- Scope predicates precisely
  - interpolants
  - [Jhala et al, POPL 2003]
- Rule out many paths at once
  - “weak” proofs of infeasibility

Trace Feasibility Formulas

Trace	SSA Trace	Trace Feasibility Formula
pc_1: x = ctr	pc_1: $x_1 = ctr_0$	$x_1 = ctr_0$
pc_2: ctr = ctr+1	pc_2: $ctr_1 = ctr_0 + 1$	$\wedge\ ctr_1 = ctr_0 + 1$
pc_3: y = ctr	pc_3: $y_1 = ctr_1$	$\wedge\ y_1 = ctr_1$
pc_4: assume(x=i-1)	pc_4: assume($x_1 = i_0 - 1$)	$\wedge\ x_1 = i_0 - 1$
pc_5: assume(y≠i)	pc_5: assume($y_1 \neq i_0$)	$\wedge\ y_1 \neq i_0$

Theorem: Trace is *Feasible* ⇔ TFF is *Satisfiable*

Newton

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

Symbolic simulation for C--

Domains

- variables: names in the program
- values: constants + symbols

State of the simulator has 3 components:

- store: map from variables to values
- conditions: predicates over symbols
- history: past valuations of the store

Symbolic simulation Algorithm

Input: path ***p***

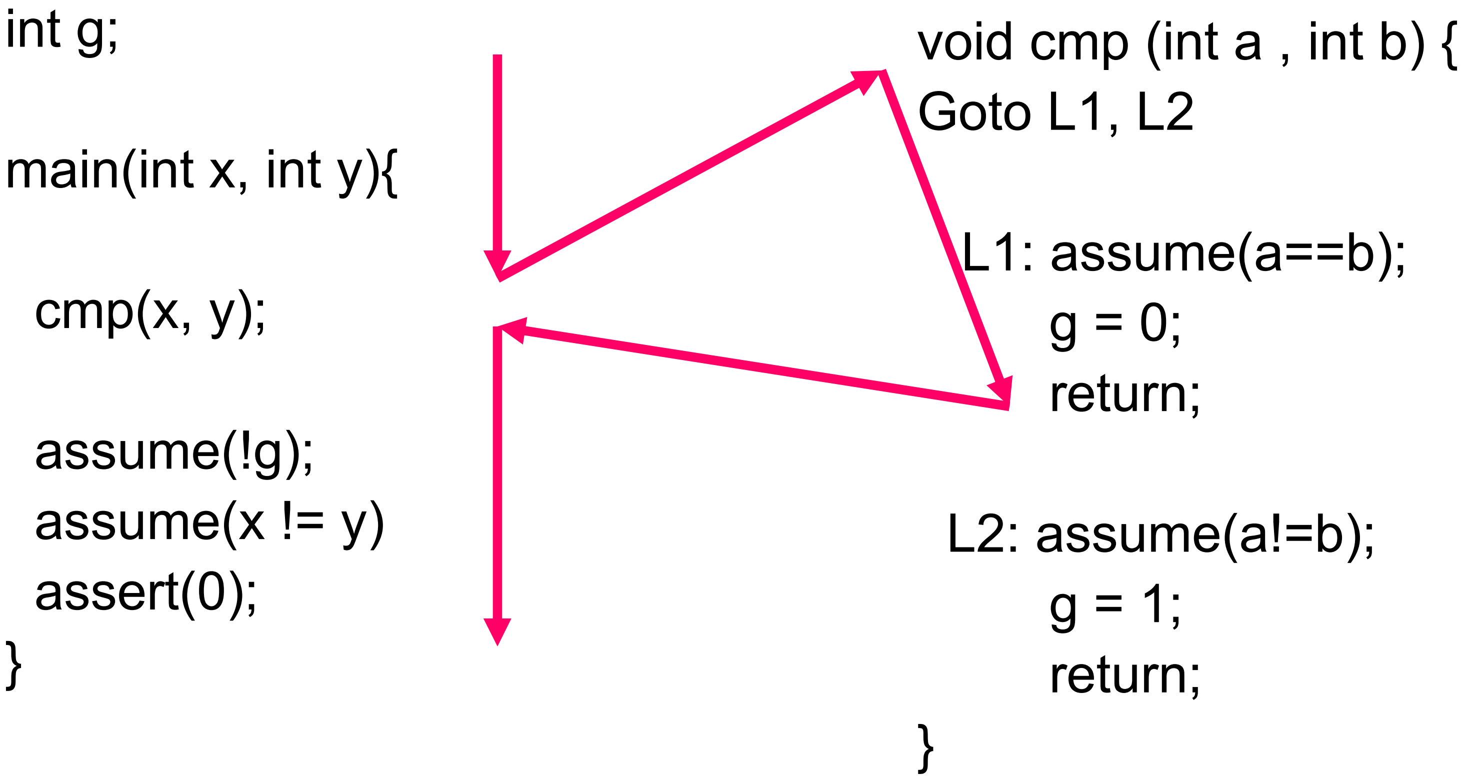
```
For each statement s in p do
    match s with
       Assign(x,e):
                   let val = Eval(e) in
                   if (Store[x]) is defined then
                               History[x] := History[x] ⊕ Store[x]
                   Store[x] := val

      Assume(e):
                   let val = Eval(e) in
                               Cond := Cond and val
                               let result = CheckConsistency(Cond) in
                               if (result  == "inconsistent") then
                                           GenerateInconsistentPredicates()
End

Say "Path p is feasible"
```

Symbolic Simulation: Caveats

- Procedure calls
  - add a stack to the simulator
  - push and pop stack frames on calls and returns
  - implement mappings to keep values “in scope” at calls and returns
- Dependencies
  - for each condition or store, keep track of which values where used to generate this value
  - traverse dependency during predicate generation



```
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);
}
```

```
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

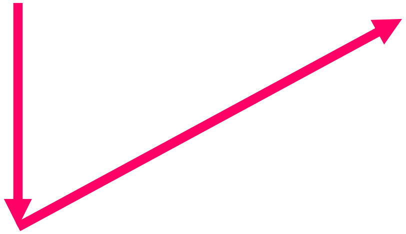
Conditions:

```
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 → A

Y → B

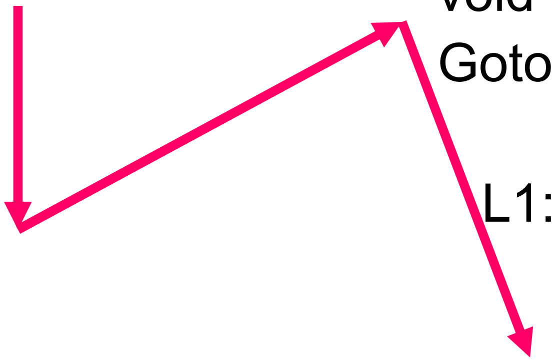
Conditions:

```
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

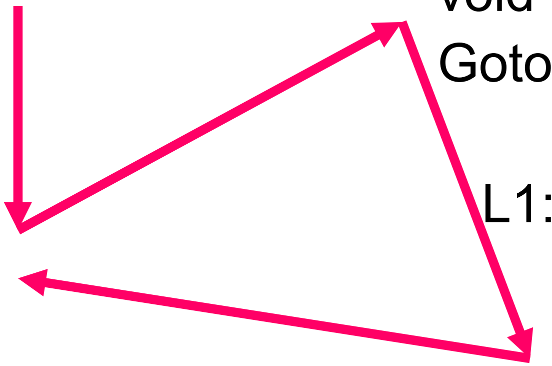
Map:

X → A

Y → 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 → A

Y → B

Conditions:

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

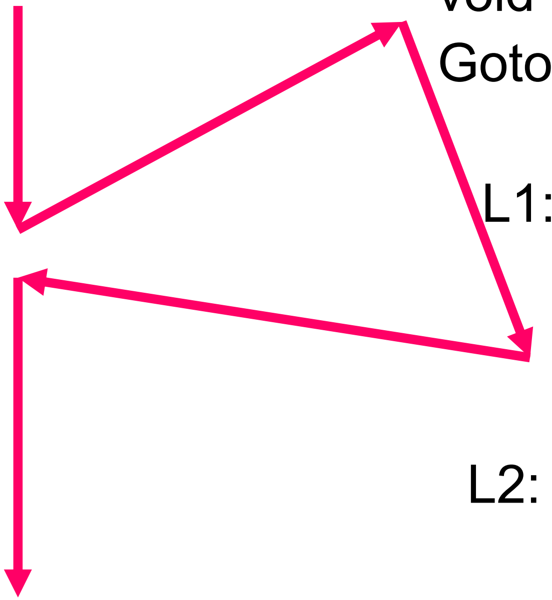
(6) (X == Y) [5]

```
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

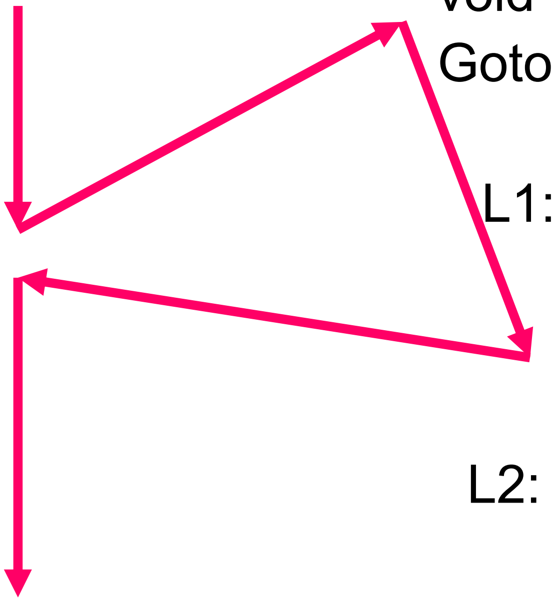
(4) b: B

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:

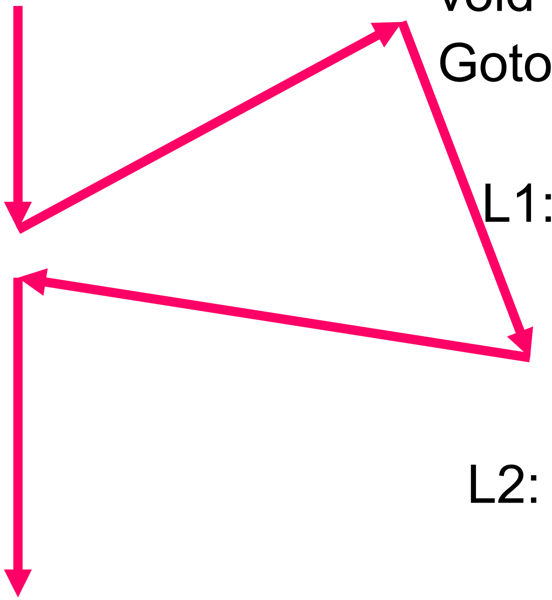
(3) a: A

(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

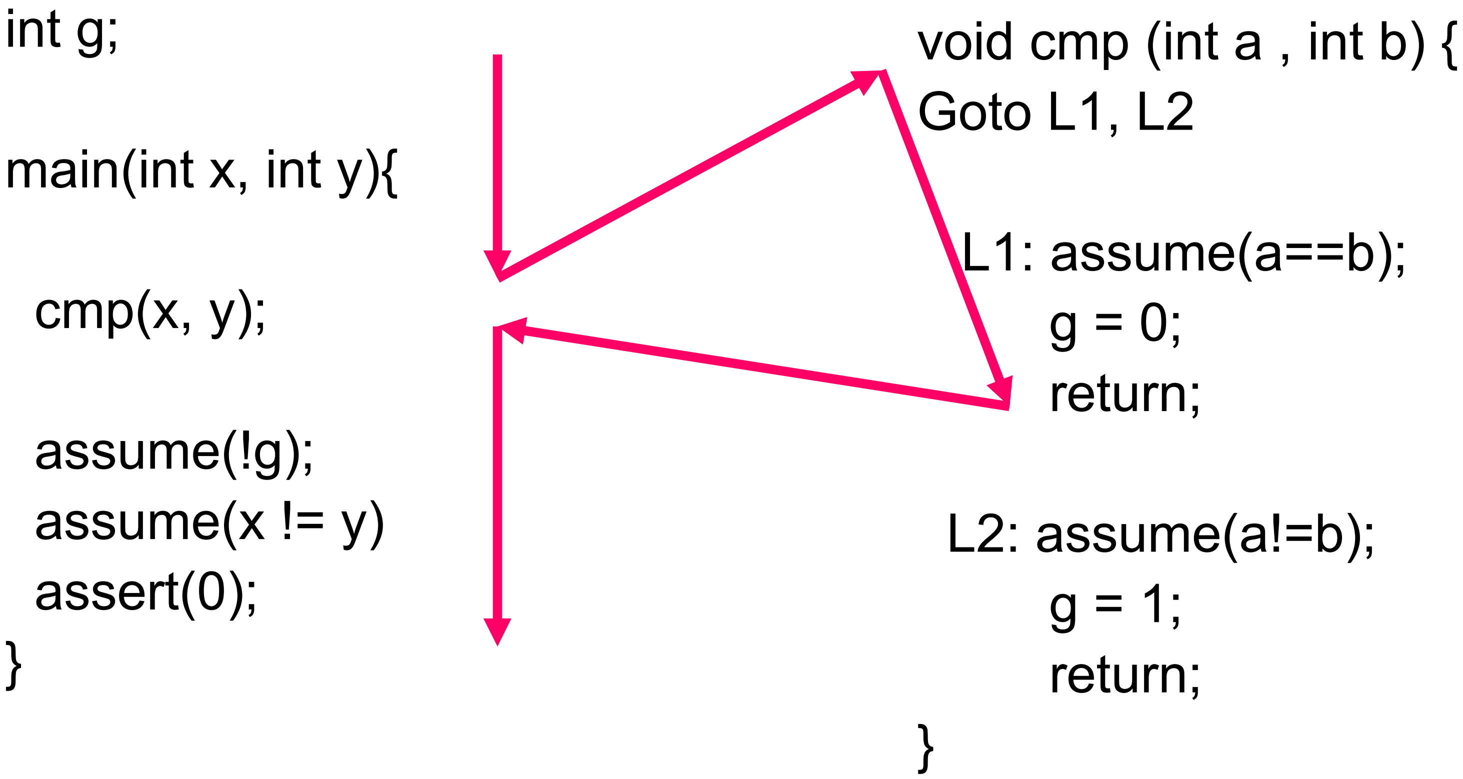
Conditions:

(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 }

Basic Question

Trace

pc_1: `x = ctr`

pc_2: `ctr = ctr + 1`

pc_3: `y = ctr`

pc_4: `assume(x = i-1)`

pc_5: `assume(y ≠ i)`

At ***pc_4***, which predicate on *present state* shows infeasibility of *suffix* ?

State…

1. … after executing trace *prefix*
2. … knows *present values* of variables
3. … makes trace *suffix* infeasible

Craig's Interpolation Theorem [Craig '57]

Given formulas ψ^- , ψ^+ s.t. $\psi^- \wedge \psi^+$ is ***unsatisfiable***

There exists an ***Interpolant*** Φ for ψ^- , ψ^+ , s.t.

1. ψ^- ***implies*** Φ
2. Φ has symbols ***common*** to ψ^-, ψ^+
3. $\Phi \wedge \psi^+$ is ***unsatisfiable***

Φ computable from ***Proof of Unsat.*** of $\psi^- \wedge \psi^+$

[Krajicek '97] [Pudlak '97]

(boolean) SAT-based Model Checking [McMillan '03]

Example

$$\psi^- = (x = y \wedge y = z)$$

$$\psi^+ = (x \neq z)$$

$$\phi = ?$$

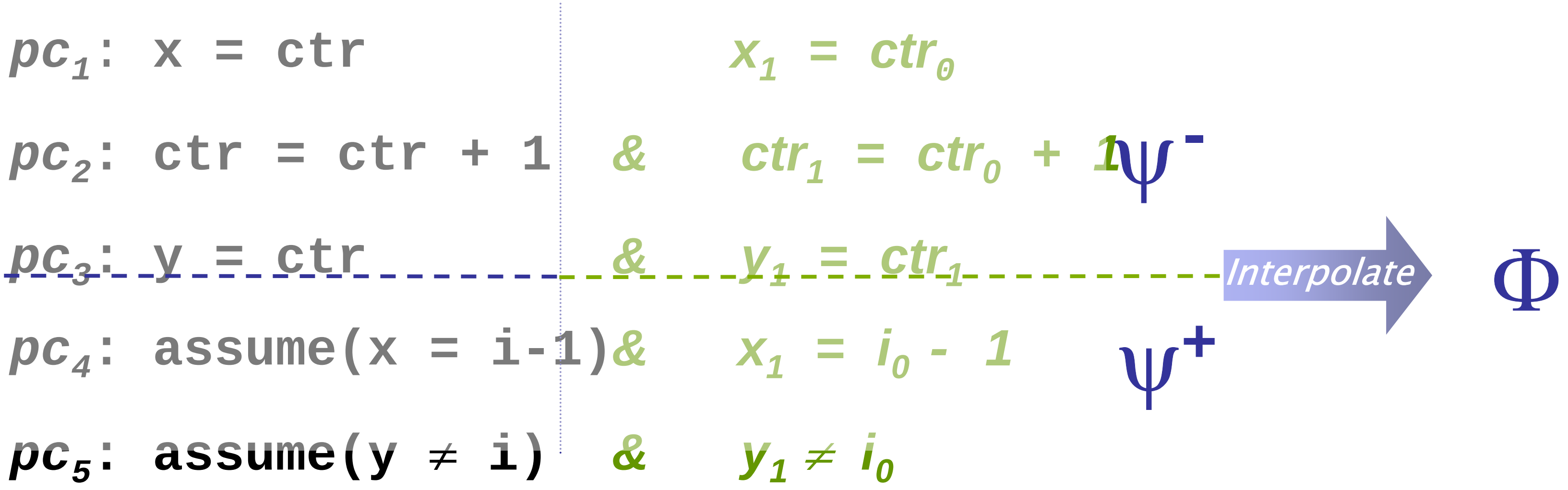
Example

$$\psi^- = (x = y \wedge y = z)$$

$$\psi^+ = (x \neq z)$$

$$\phi = (x = z)$$

Interpolant = Predicate !



Trace		Trace Formula
pc_1: `x = ctr`		$x_1 = ctr_0$
pc_2: `ctr = ctr + 1`	&	$ctr_1 = ctr_0 + 1$
pc_3: `y = ctr`	&	$y_1 = ctr_1$
pc_4: `assume(x = i-1)`	&	$x_1 = i_0 - 1$
pc_5: `assume(y ≠ i)`	&	$y_1 \neq i_0$

Interpolant:

1. ψ^- *implies* Φ
2. Φ has symbols *common* to ψ^-, ψ^+
3. $\Phi \wedge \psi^+$ is *unsatisfiable*

Interpolant = Predicate !

Trace		Trace Formula
`pc`$_1$`: x = ctr`		$x_1 = ctr_0$
`pc`$_2$`: ctr = ctr + 1`	&	$ctr_1 = ctr_0 + 1$
`pc`$_3$`: y = ctr`	&	$y_1 = ctr_1$
`pc`$_4$`: assume(x = i-1)`	&	$x_1 = i_0 - 1$
`pc`$_5$`: assume(y ≠ i)`	&	$y_1 \neq i_0$

ψ^-

ψ^+

Interpolate

Φ

$y_1 = x_1 + 1$

Interpolant:

1. ψ^- *implies* Φ
2. Φ has symbols *common* to ψ^-, ψ^+
3. $\Phi \wedge \psi^+$ is *unsatisfiable*

Interpolant = Predicate !

Trace		Trace Formula
pc_1: `x = ctr`		$x_1 = ctr_0$
pc_2: `ctr = ctr + 1`	&	$ctr_1 = ctr_0 + 1$
pc_3: `y = ctr`	&	$y_1 = ctr_1$
pc_4: `assume(x = i-1)`	&	$x_1 = i_0 - 1$
pc_5: `assume(y ≠ i)`	&	$y_1 \neq i_0$

pc_4

ψ^-

ψ^+

Interpolate

Φ

$y_1 = x_1 + 1$

Predicate at pc_4:
$y = x+1$

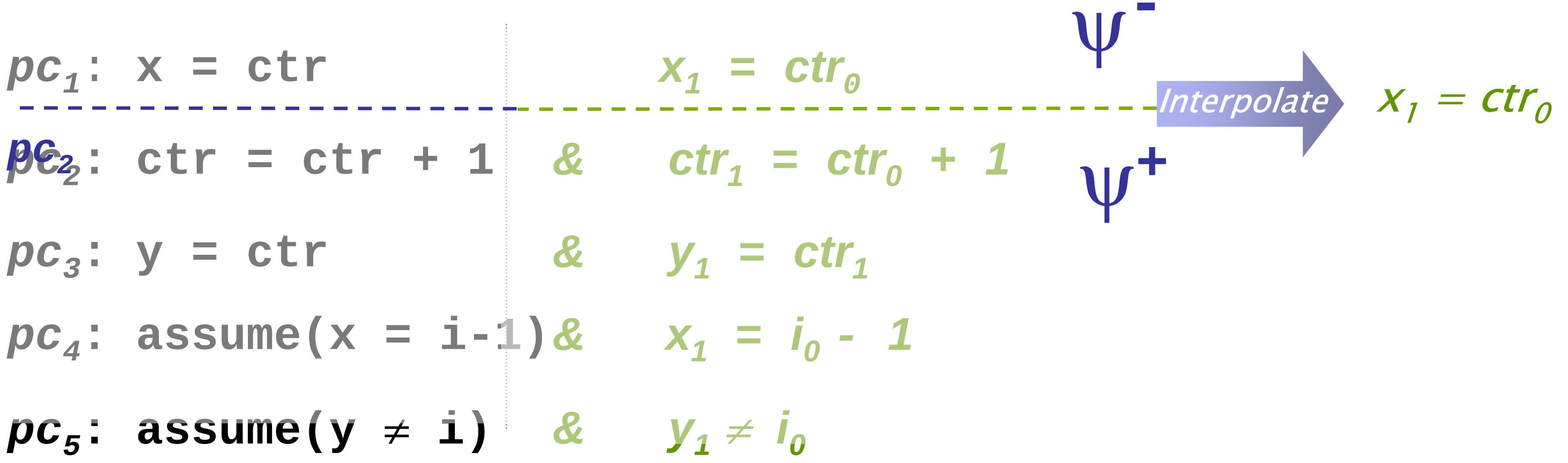
Interpolant:

1. ψ^- *implies* Φ
2. Φ has symbols *common* to ψ^-, ψ^+
3. $\Phi \wedge \psi^+$ is *unsatisfiable*

Building Predicate Maps

Predicate Map

pc_2: $x = ctr$



Trace		Trace Formula
pc_1: `x = ctr`		$x_1 = ctr_0$
pc_2: `ctr = ctr + 1`	&	$ctr_1 = ctr_0 + 1$
pc_3: `y = ctr`	&	$y_1 = ctr_1$
pc_4: `assume(x = i-1)`	&	$x_1 = i_0 - 1$
pc_5: `assume(y ≠ i)`	&	$y_1 \neq i_0$

- Cut + Interpolate at *each* point
- Pred. Map: $pc_i \mapsto$ Interpolant from cut i

Building Predicate Maps

Predicate Map

pc_2: $x = ctr$

pc_3: $x = ctr - 1$

Trace	Trace Formula
pc_1: `x = ctr`	$x_1 = ctr_0$
pc_2: `ctr = ctr + 1`	& $ctr_1 = ctr_0 + 1$
pc_3: `y = ctr`	& $y_1 = ctr_1$
pc_4: `assume(x = i-1)`	& $x_1 = i_0 - 1$
pc_5: `assume(y ≠ i)`	& $y_1 \neq i_0$

pc_3

ψ^-

Interpolate $x_1 = ctr_1 - 1$

ψ^+

- Cut + Interpolate at *each* point
- Pred. Map: $pc_i \mapsto$ Interpolant from cut i

Building Predicate Maps

Predicate Map

pc_2: $x = ctr$

pc_3: $x = ctr-1$

pc_4: $y = x+1$

pc_5: $y = i$

Trace		Trace Formula
pc_1: `x = ctr`		$x_1 = ctr_0$
pc_2: `ctr = ctr + 1`	&	$ctr_1 = ctr_0 + 1$
pc_3: `y = ctr`	&	$y_1 = ctr_1$
pc_4: `assume(x = i-1)`	&	$x_1 = i_0 - 1$
pc_5 pc_5: `assume(y ≠ i)`	&	$y_1 \neq i_0$

ψ^-

Interpolate → $y_1 = i_0$

ψ^+

- Cut + Interpolate at *each* point
- Pred. Map: $pc_i \mapsto$ Interpolant from cut i

Building Predicate Maps

Predicate Map
pc_2: $x = ctr$
pc_3: $x = ctr - 1$
pc_4: $y = x + 1$
pc_5: $y = i$

Trace		Trace Formula
pc_1: `x = ctr`		$x_1 = ctr_0$
pc_2: `ctr = ctr + 1`	&	$ctr_1 = ctr_0 + 1$
pc_3: `y = ctr`	&	$y_1 = ctr_1$
pc_4: `assume(x = i-1)`	&	$x_1 = i_0 - 1$
pc_5: `assume(y ≠ i)`	**&**	**$y_1 \neq i_0$**

Theorem: *Predicate map* makes trace *abstractly infeasible*

SLAM Predicate Generation

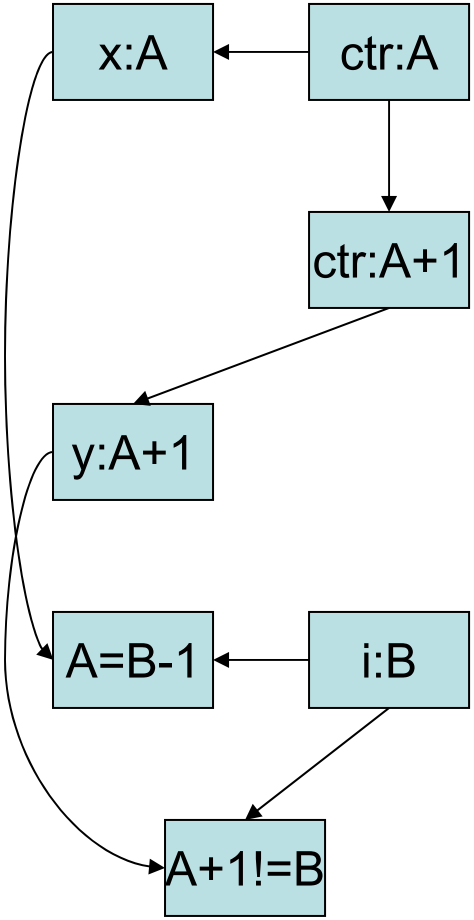
pc_1: **`x = ctr`**

pc_2: **`ctr = ctr + 1`**

pc_3: **`y = ctr`**

pc_4: **`assume(x = i-1)`**

pc_5: **`assume(y ≠ i)`**



ctr = A
x = A
ctr = A+1
y = A+1
A = B-1
i = B

eliminate A and B

ctr = x
ctr = x+1
y = x+1
x = i-1

Weakness of explanation

```
a := b;
a := a-1;
c := 2*b;
assume(b>0);
assume(a<b);
assume(c==a);
```

⇨

```
a := b;
d := a-1;
c := 2*b;
assume(b>0);
assume(d<b);
assume(c==d);
```

⇨

$a=b \wedge d=a-1 \wedge c=2b \wedge b>0 \wedge d<b \wedge c=d$

unsatisfiable because

$b>0 \wedge c=2b \wedge a=b \wedge d=a-1$

$\Rightarrow b>0 \wedge c=2b \wedge d=b-1$

$\Rightarrow c \neq d$

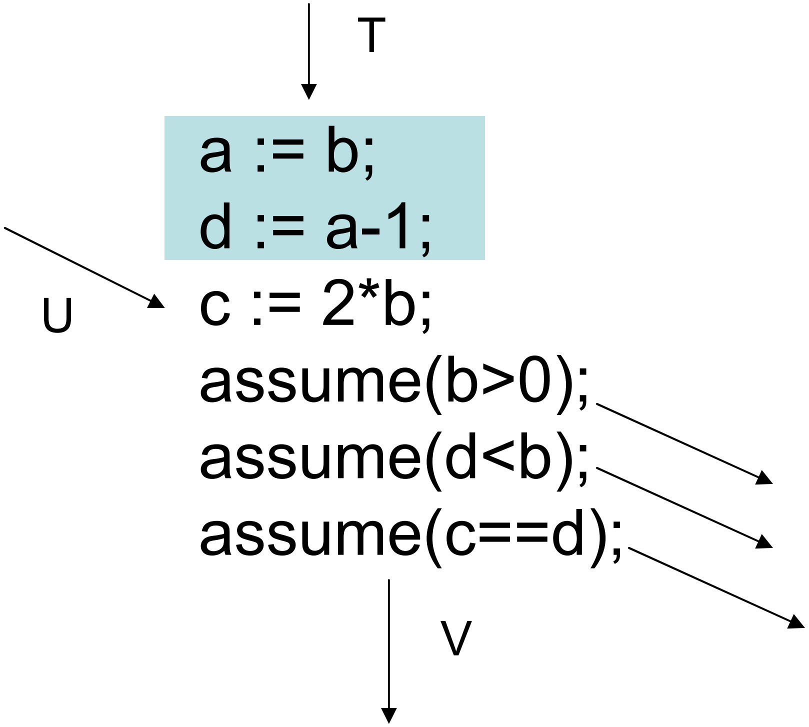
$\Rightarrow$

also unsatisfiable because

$b>0 \wedge c=2b \wedge d<b$

$\Rightarrow c \neq d$

Weakness of explanation



```
a := b;
d := a-1;
c := 2*b;
assume(b>0);
assume(d<b);
assume(c==d);
```

{ b>0, c=2b, a=b, d=a-1 }
eliminates paths from
T to V

{ b>0, c=2b, d<b }
eliminates paths from
{T,U} to V

Idea: “redundant predicates”

```
a := b;
d := a-1;
c := 2*b;
assume(b>0);
assume(d<b);
assume(c==d);
```

```


c := 2*b;
assume(b>0);
assume(d<b);
assume(c==d);
```

$a=b \wedge d=a-1 \Rightarrow d<b$

Summary

- Predicate discovery the key to making abstraction/refinement work
- Two goals:
  - proper scoping of predicates improves efficiency
  - weak explanations rule out many infeasible paths in a single swipe