Counter Example Guided Refinement CEGAR

Mooly Sagiv

A problem has been detected and Windows has been shut down to prevent damage to your computer.

IRQL_NOT_LESS_OR_EQUAL

If this is the first time you have seen this stop error screen, restart your computer. If this screen appears again, follow these steps:

Check to be sure any new hardware or software is properly installed. If this is a new installation, ask your hardware or software manufacturer for any Windows updates you might need. If problems continue, disable or remove any newly installed hardware or software. Disable BIOS memory options such as caching or shadowing.

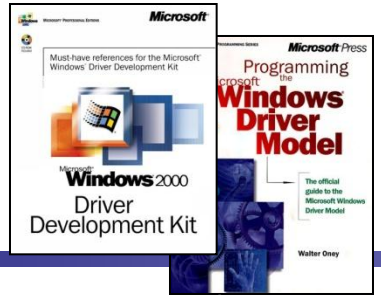
If you need to use Safe Mode to remove or disable components, restart your computer, press F8 to select Advanced Startup Options, and then select Safe Mode.

Technical information:

*** STOP: 0x0000000A 0xFFFFFA802880010A, 0x0000000000000002, 0x0000000000000000, 0xFFFFF8000185E251)

SLAM

- Microsoft blames most Windows crashes on third party device drivers
- The Windows device driver API is quite complicated
- Drivers are low level C code
- SLAM: Tool to automatically check device drivers for certain errors
- SLAM is shipped with Device Driver Development Kit
- Full detail available at http://research.microsoft.com/slam/



Rules

Development

Static Driver Verifier

Precise
API Usage Rules
(SLAM)

Testing

"Things like even software verification, this has been the Holy Grail of computer science for many decades but now in some very key areas, for example, verification we're building tools that can

Source Code

Recap

- Many abstract domains
  - Signs
  - Odd/Even
  - Constant propagation
  - Intervals
  - [Polyhedra]
  - Canonic abstraction
  - Domain constructors
  - …
- Static Algorithms
  - Iterative Chaotic Iterations
  - Widening/Narrowing
  - Interprocedural Analysis
  - Concurrency
  - Modularity
  - Non-Iterative methods

A Lattice of Abstractions

- Every element is an abstract domain
- $A \sqsubseteq A'$ if there exists a Galois Connection from A to A'

But how to find the appropriate abstract domain

- Precision vs. Scalability
- Sometimes precision improves scalability
- Specialize the abstraction for the desired property

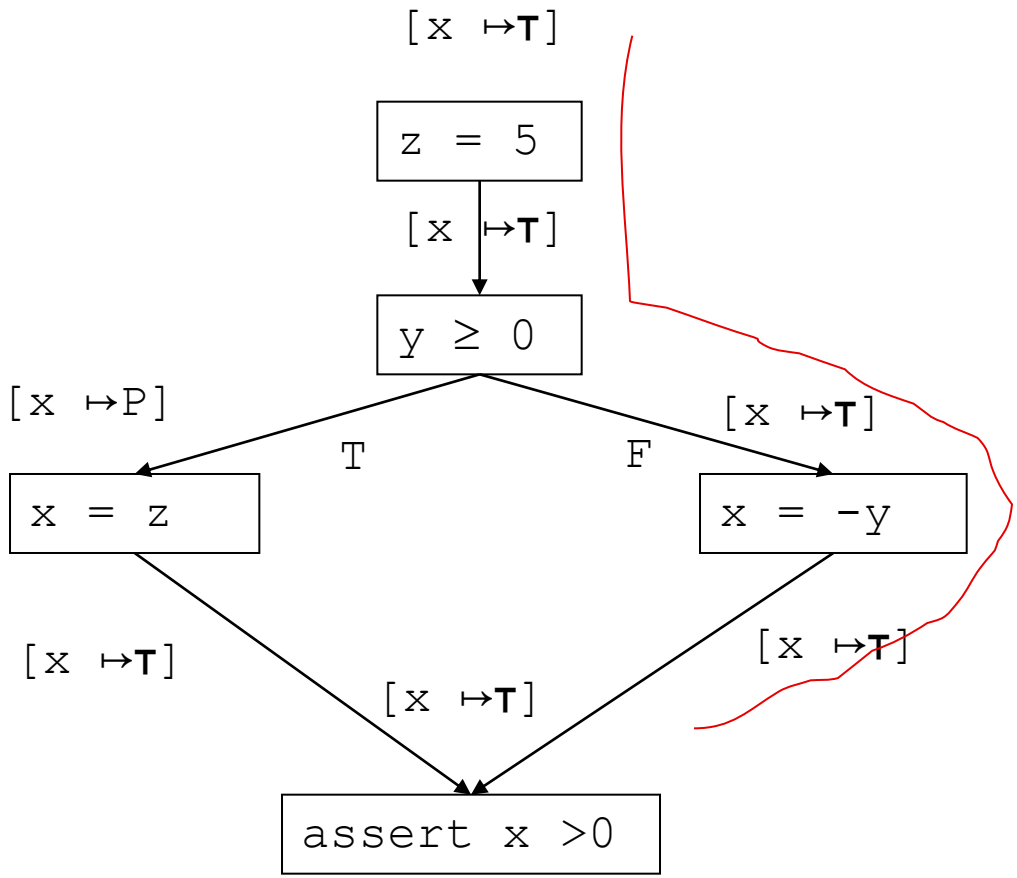
Counter Example Guided Refinement (CEGAR)

- Run the analysis with a simple abstract domain
- When the analysis verifies the property declare done
- If the analysis reports an error employs a theorem prover to identify if the error is feasible
  - If the error is feasible generate a concrete trace
  - If the error is spurious refine the abstract domain and repeat

A Simple Example

```
z =5
if (y >0)
   x = z;
 else
  x = -y;
assert x >0
```

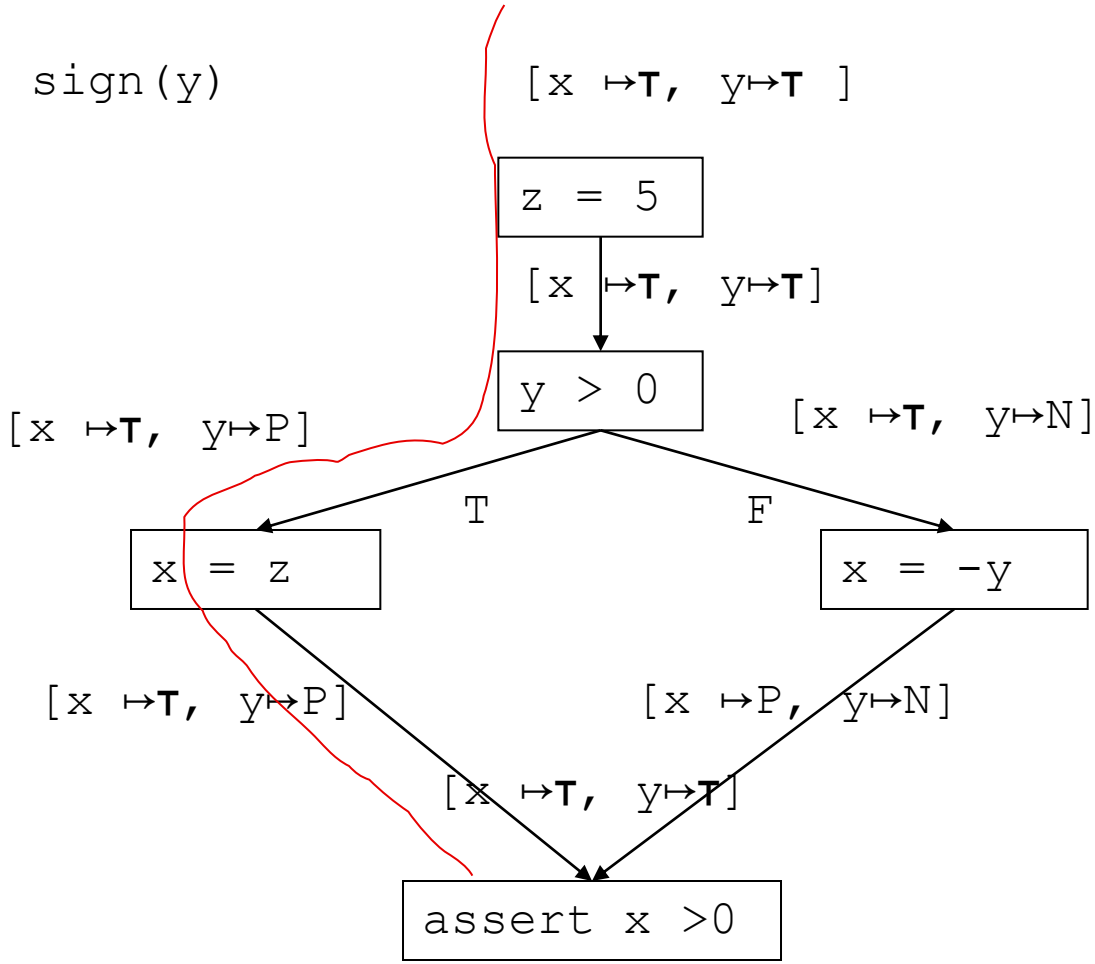
sign(x)



A Simple Example

```
z =5
if (y >0)
   x = z;
 else
  x = -y;
assert x >0
```

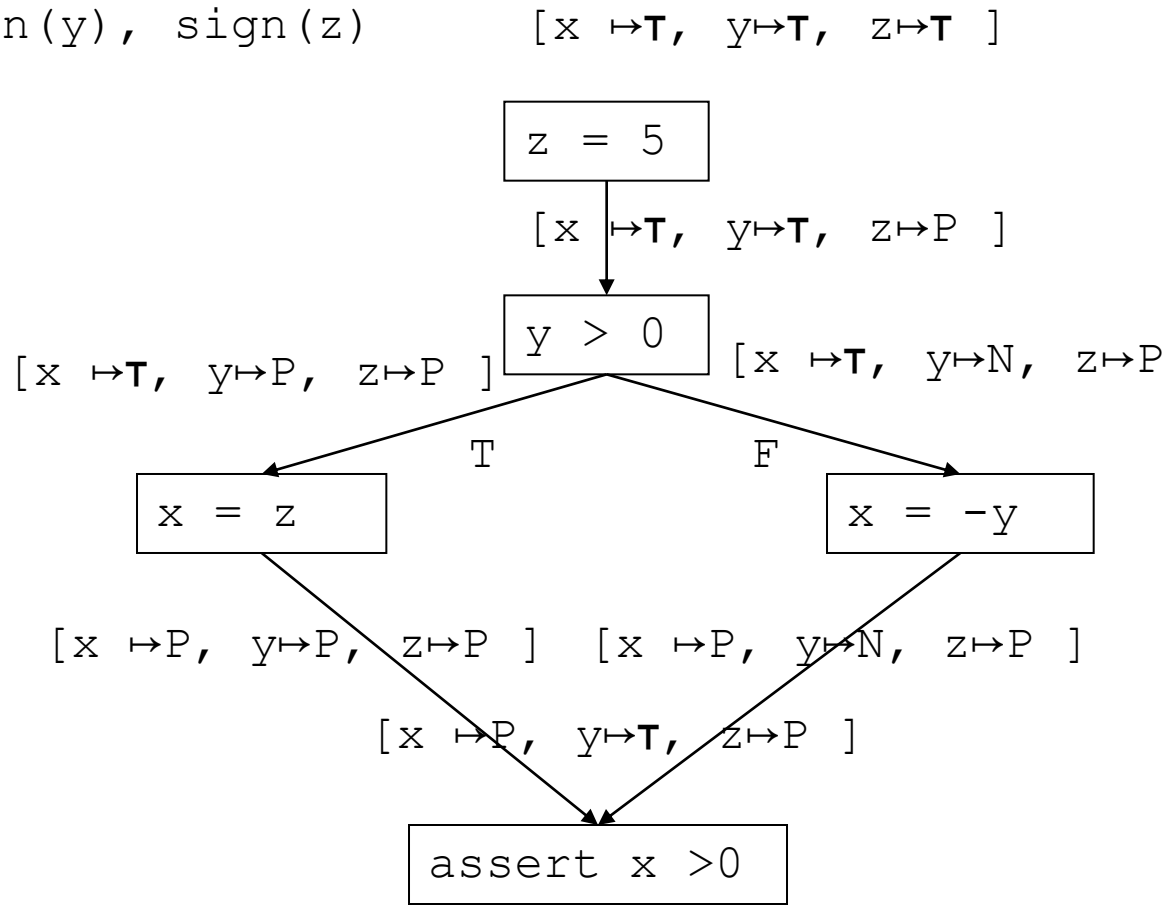
sign(x), sign(y)



A Simple Example

```
z =5
if (y >0)
   x = z;
 else
  x = -y;
assert x >0
```

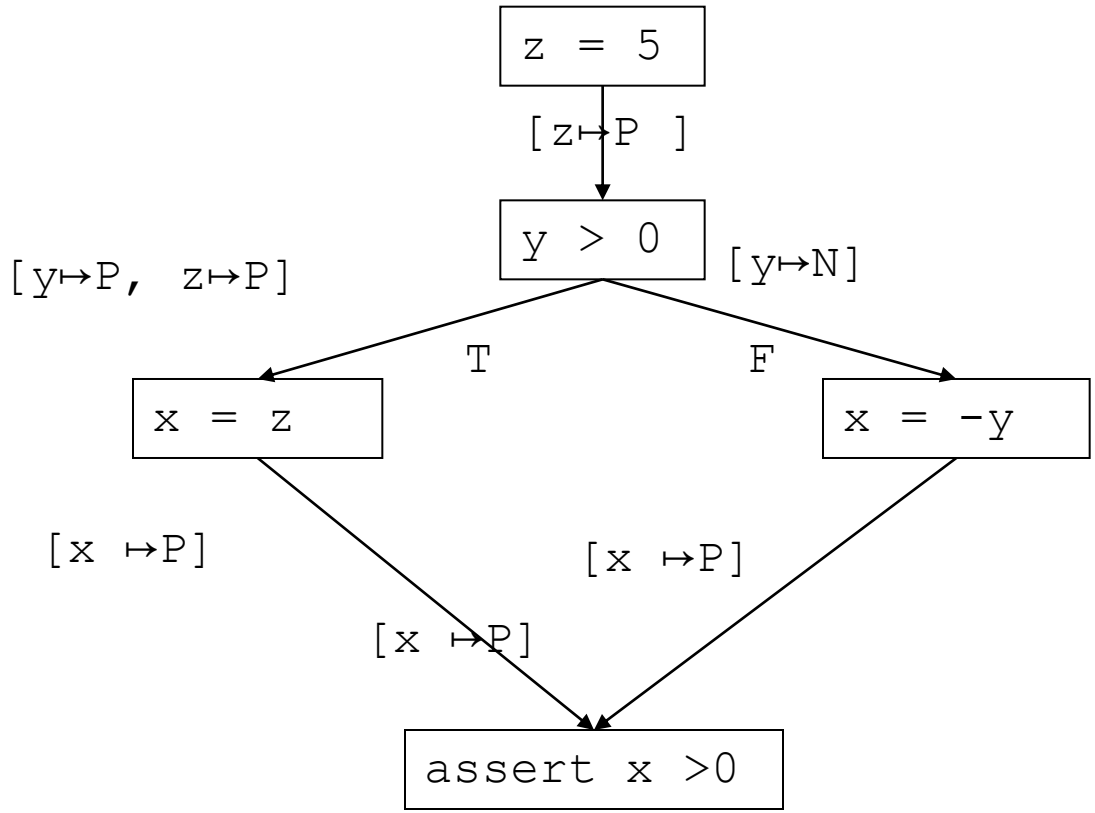
sign(x), sign(y), sign(z)



Simple Example (local abstractions)

```
z =5
if (y >0)
   x = z;
 else
  x = -y;
assert x >0
```

sign(x), sign(y), sign(z) []



Plan

- Predicate Abstraction
- CEGAR in BLAST (inspired by SLAM) POPL'04
- Limitations

BLAST

Berkeley Lazy Abstraction Software Tool

www.eecs.berkeley.edu/~blast/

Abstractions from Proofs: POPL'04

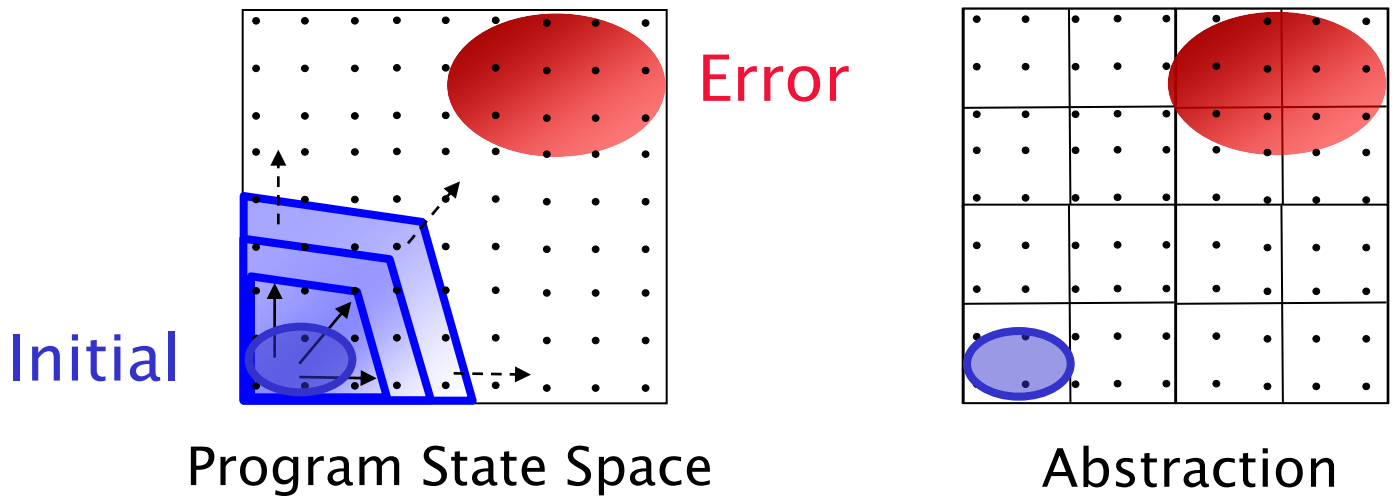
Ranjit Jhala
UCSD

Rupak Majumdar
MPI

Ken McMillan
MSR

Thomas Henzinger
IST

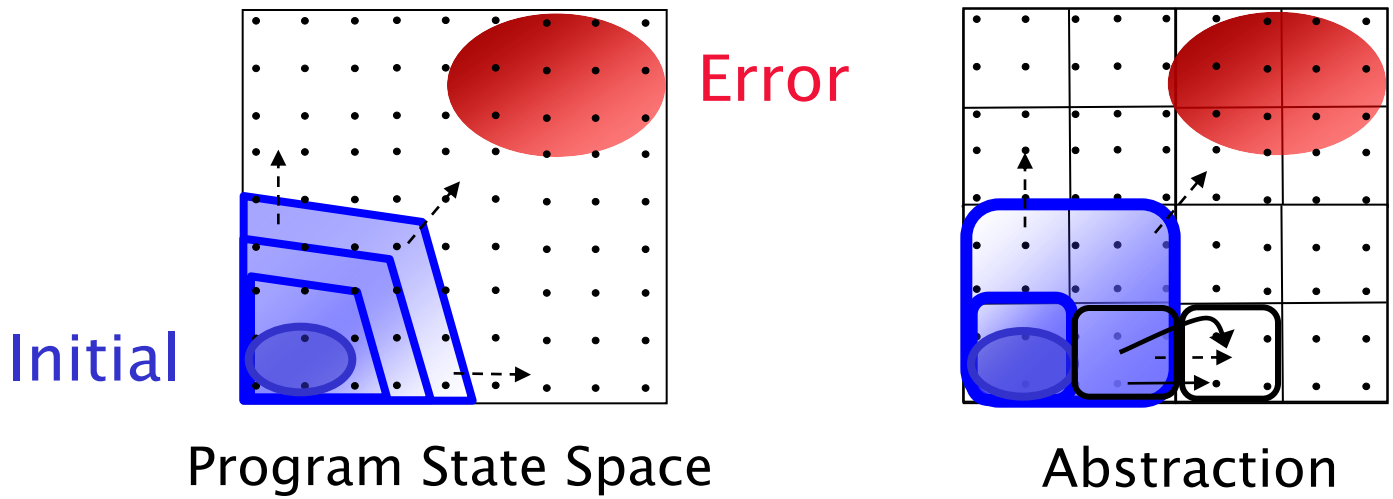
Predicate Abstraction: A crash course



- Abstraction: ***Predicates*** on program state
  - Signs: $x > 0$
  - Aliasing: $\&x \neq \&y$

- States satisfying the same predicates are equivalent
  - Merged into single abstract state

(Predicate) Abstraction: A crash course



Q1*: Which predicates* are required to verify a property ?

The Predicate Abstraction Domain

- Fixed set of predicates Pred
- The meaning of each predicate $p_i \in$ Pred is a closed first order formula f_i
- The relational domain is
  $<P(P(Pred)), \emptyset, P(Pred), \cup, \cap>$
  - Join is set union
  - State space explosion
- Special case of canonic abstraction

A Simple Example

Predicates: p1 = x > 0 p2 = y ≥ 0

```
int x, y;
x = 1;
y = 2 ;
while (*) do {
    x = x + y ;
  }
assert x > 0;
```

```
bool p1, p2;
p1 = true ;
p2 = true ;
while (*) do {
    p1 = (p1&&p2 ? 1 : *)
  }
assert p1 ;
```

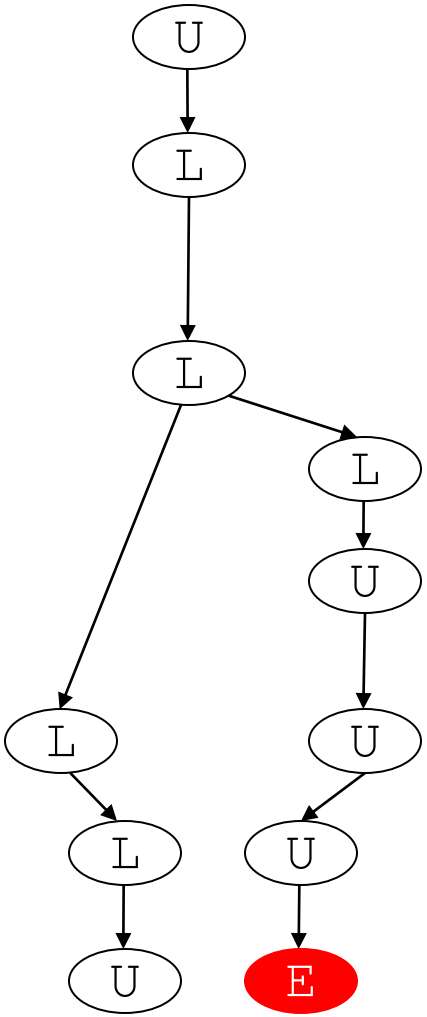
SLAM Example

Does this code obey the locking rule?

```
do {
   KeAcquireSpinLock();

   nPacketsOld = nPackets;

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

KeReleaseSpinLock();
```

SLAM Example

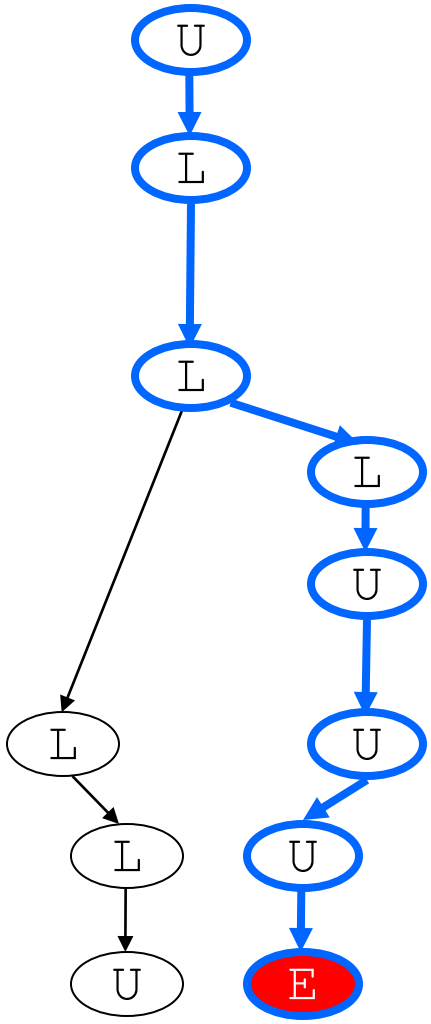
Model checking boolean program (bebop)




```
do {
   KeAcquireSpinLock();

   if(*){

      KeReleaseSpinLock();

   }
} while (*);

KeReleaseSpinLock();
```

SLAM Example

Is error path feasibl
in C program?
(newton)

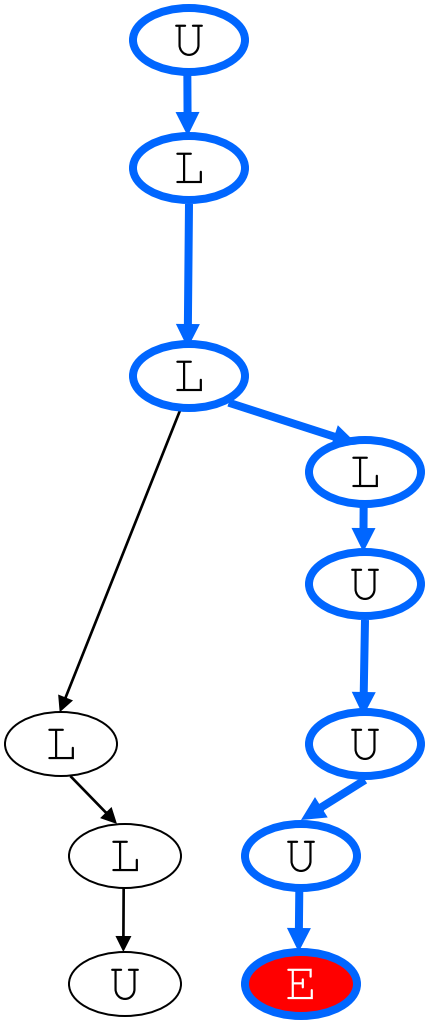



```
do {
   KeAcquireSpinLock();

   nPacketsOld = nPackets;

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

KeReleaseSpinLock();
```

SLAM Example

b : (nPacketsOld == nPackets)

Add new predicate
boolean program
(c2bp)

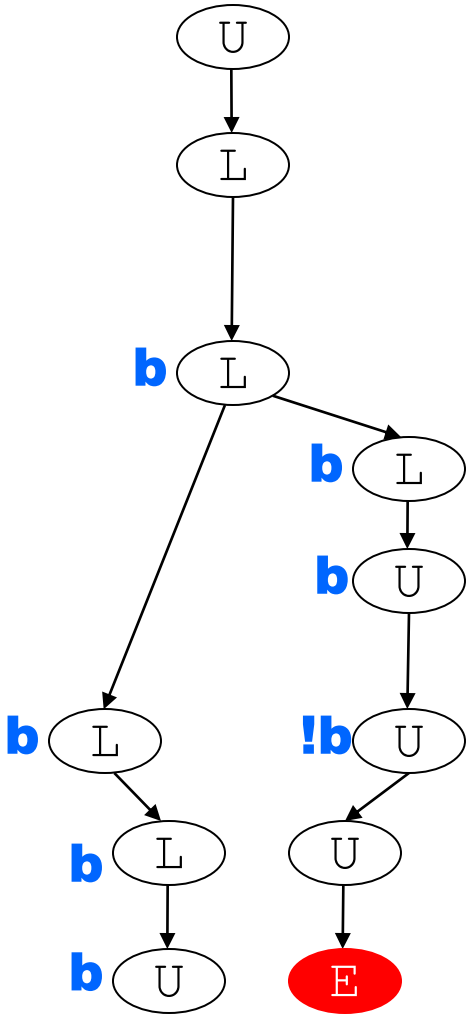



```
do {
   KeAcquireSpinLock();

   nPacketsOld = nPackets; b = true;

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

KeReleaseSpinLock();
```

SLAM Example

b : (nPacketsOld == nPackets)

Model checking refined boolean program (bebop)




```
do {
   KeAcquireSpinLock();

   b = true;

   if(*){

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

KeReleaseSpinLock();
```

Scalability vs. Verification

- Few predicates tracked
  - *e.g.* type of variables
- Imprecision hinders Verification
  - Spurious counterexamples
- Many predicates tracked
  - *e.g.* values of variables
- State explosion
  - Analysis drowned in detail

Example

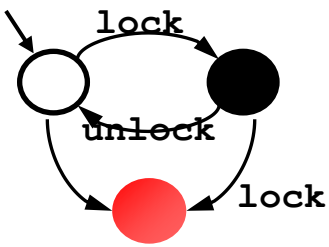
scalability

Only track *lock*

```
while(*){
1: if (p1) lock();
   if (p1) unlock()
                 …
2: if (p2) lock();
   if (p2) unlock()
                 …
n: if (pn) lock();
   if (pn) unlock()
}
```

T ● *F*

T ●



Bogus Counterexample

- Must *correlate* branches

Predicate p_1 makes trace *abstractly infeasible*

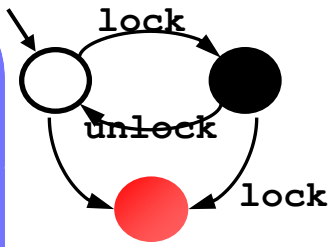
p_i required for verification

Example

scalability

Only track *lock*

```
while(*){
1: if (p1) lock();
   if (p1) unlock()
                 …
2: if (p2) lock();
   if (p2) unlock()
                 …
n: if (pn) lock();
   if (pn) unlock()
}
```



verification

Track *lock*, p_i s

Bogus Counterexample

- Must *correlate* branches

State Explosion

- $> 2^n$ distinct states
- intractable

How can we get scalable verification ?

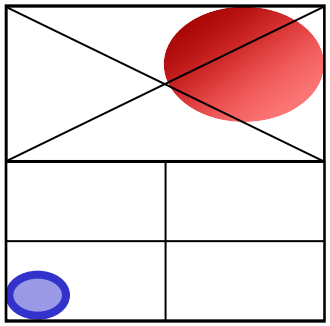
By Localizing Precision

p_1

p_2

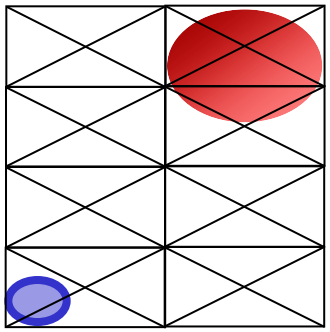
p_n

```
while (*) {
1: if (p1) lock();
   if (p1) unlock();
           …
2: if (p2) lock();
   if (p2) unlock();
           …
n: if (pn) lock();
   if (pn) unlock();
}
```



Preds. Used locally

Ex: 2 * n states

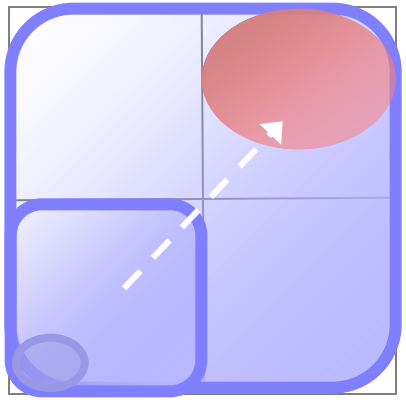


Preds. used globally

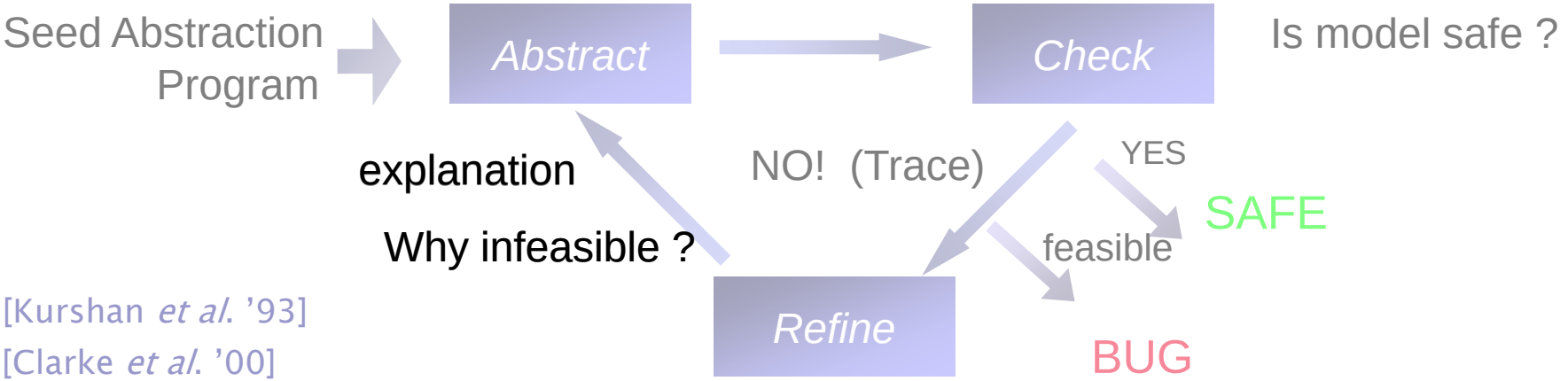
Ex: 2^n states

Q2: *Where* are the predicates required ?

Counterexample Guided Refinement

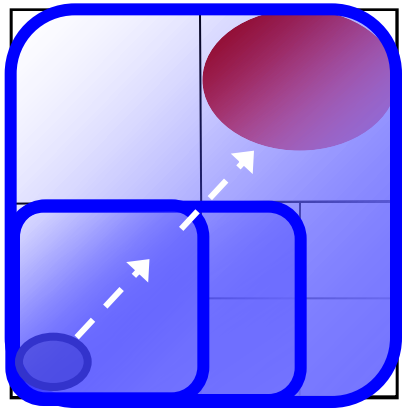


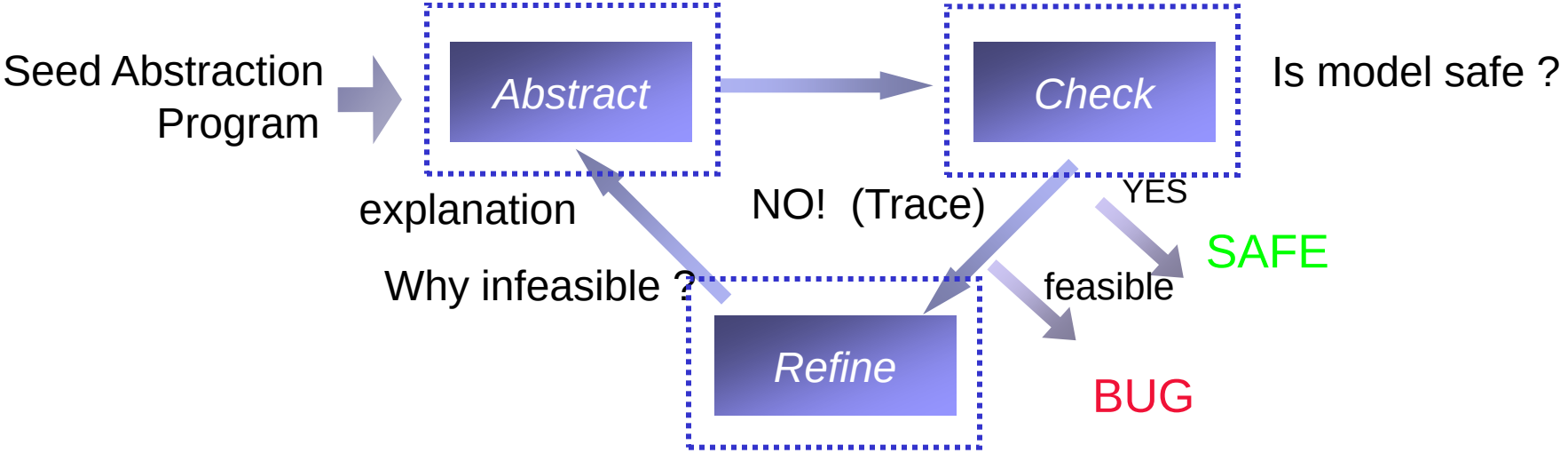
1. *What predicates* remove trace ?
   - Make it abstractly infeasible
2. *Where* are predicates needed ?



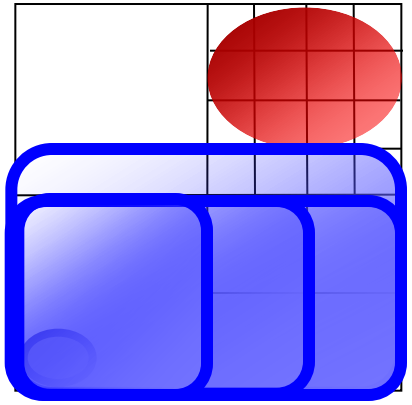
[Kurshan *et al.* '93]
[Clarke *et al.* '00]
[Ball, Rajamani '01]

Counterexample Guided Refinement

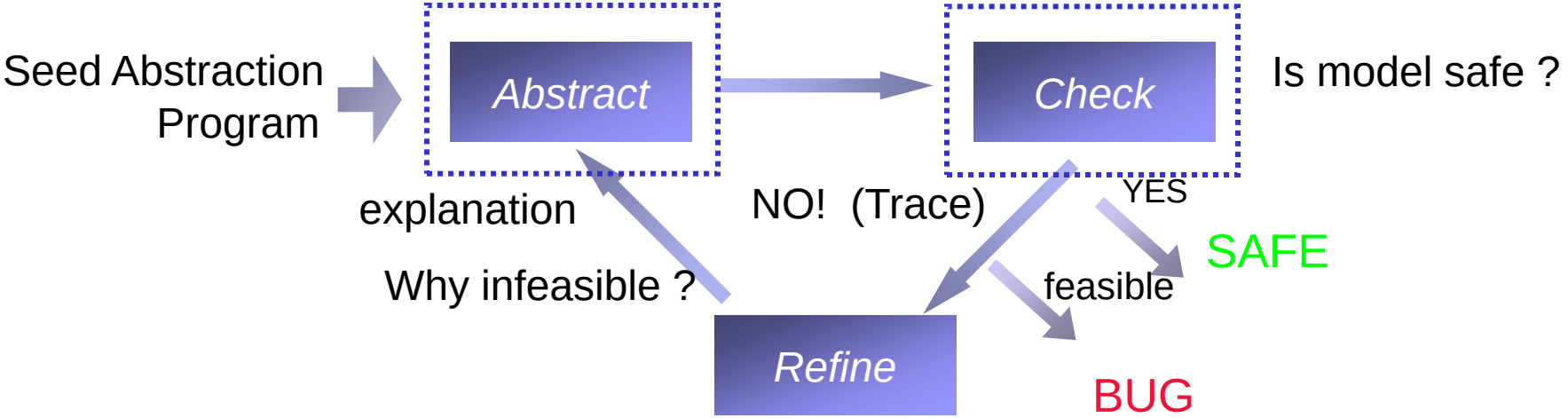




Counterexample Guided Refinement

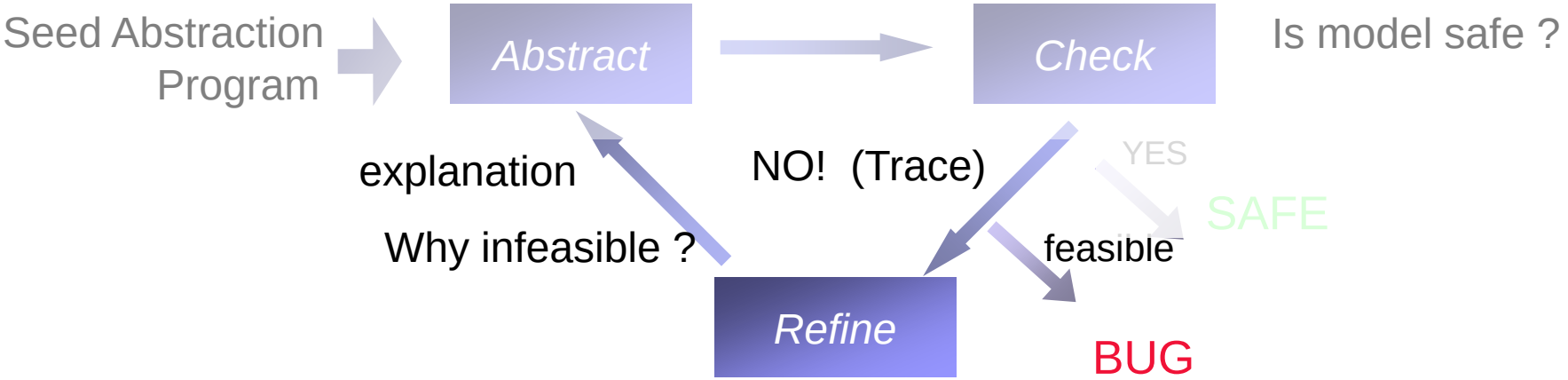


safe



This Talk: Counterexample Analysis

1. *What predicates* remove trace ?
   - Make it abstractly infeasible
2. *Where* are predicates needed ?



Plan

1. Motivation

2. Refinement using Traces
   - Simple
   - Procedure calls

3. Results

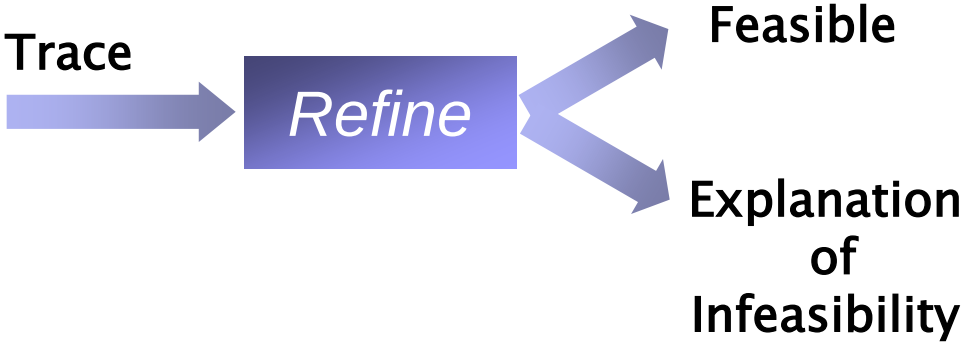
Trace Formulas

- A single abstract trace represents infinite number of traces
  - Different loop iterations
  - Different concrete values
- Solution
  - Only considers concrete traces with the same number of executions
  - Use formulas to represent sets of states

Representing States as *Formulas*

$[F]$ states satisfying F $\{s \mid s \models F\}$	F FO formula over prog. vars
$[F_1] \cap [F_2]$	$F_1 \wedge F_2$
$[F_1] \cup [F_2]$	$F_1 \vee F_2$
$\overline{[F]}$	$\neg F$
$[F_1] \subseteq [F_2]$	F_1 implies F_2 i.e. $F_1 \wedge \neg F_2$ unsatisfiable

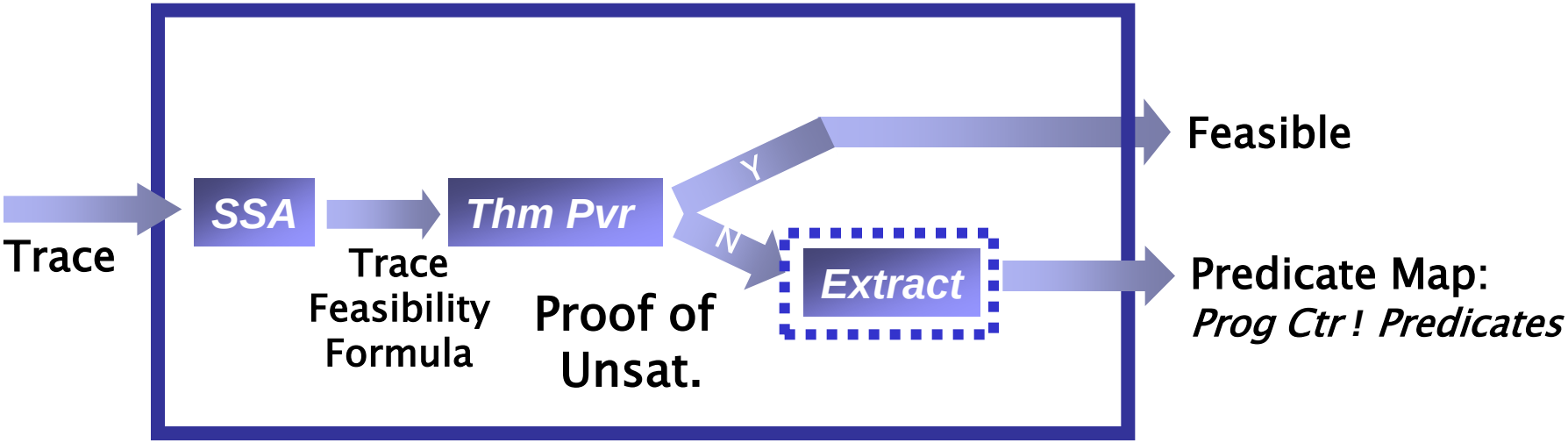
Counterexample Analysis



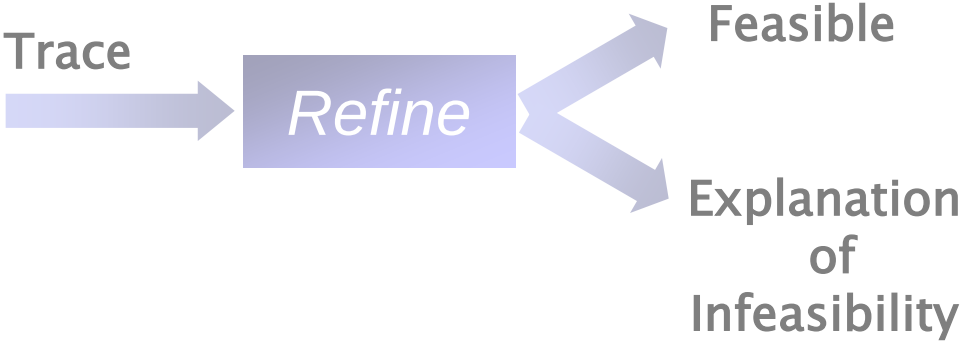
Q0: Is trace feasible ?

Q1: What predicates remove trace ?

Q2: *Where* are preds required ?



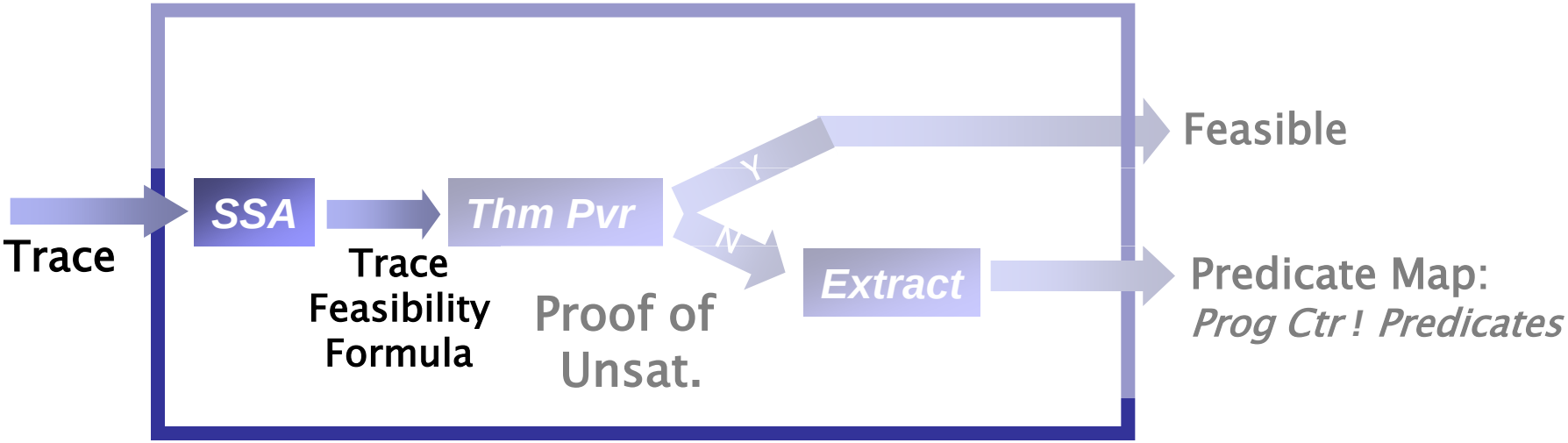
Counterexample Analysis



Q0: Is trace feasible ?

Q1: What predicates remove trace ?

Q2: Where are preds required ?



Traces

```
pc1: x = ctr;
pc2: ctr = ctr + 1;
pc3: y = ctr;
pc4: if (x = i-1){
pc5:   if (y != i){
        ERROR: }
     }
```

```
pc1: x = ctr
pc2: ctr = ctr + 1
pc3: y = ctr
pc4: assume(x = i-1)
pc5: assume(y ≠ i)
```

$y = x + 1$

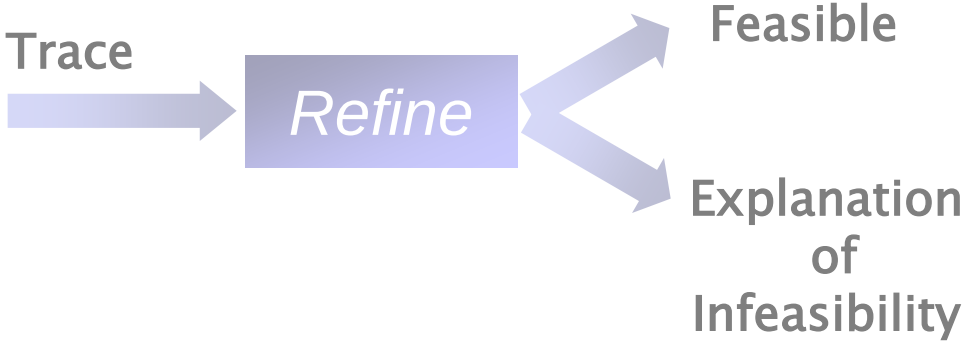
Trace Feasibility Formulas

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***

Compact Verification Conditions [Flanagan,Saxe '00]

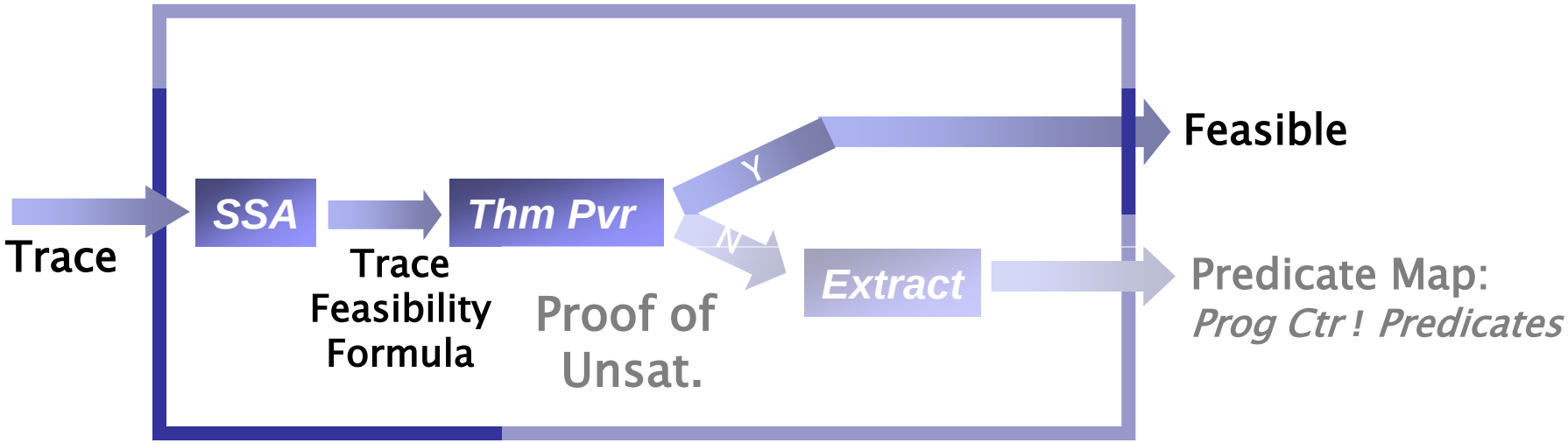
Counterexample Analysis



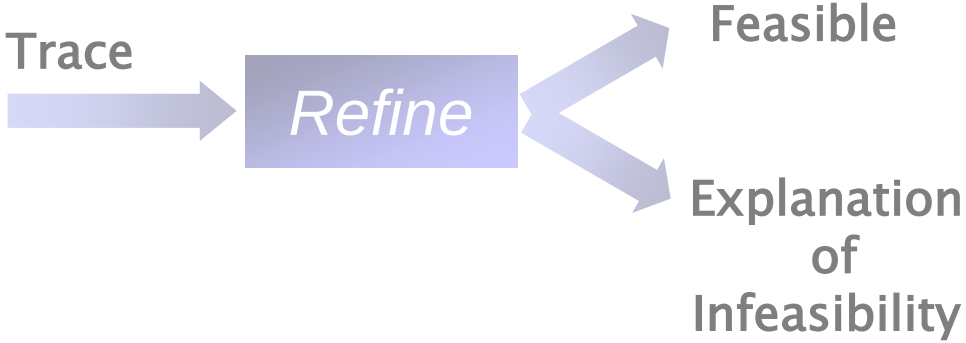
Q0: Is trace feasible ?

Q1: What predicates remove trace ?

Q2: Where are preds required ?



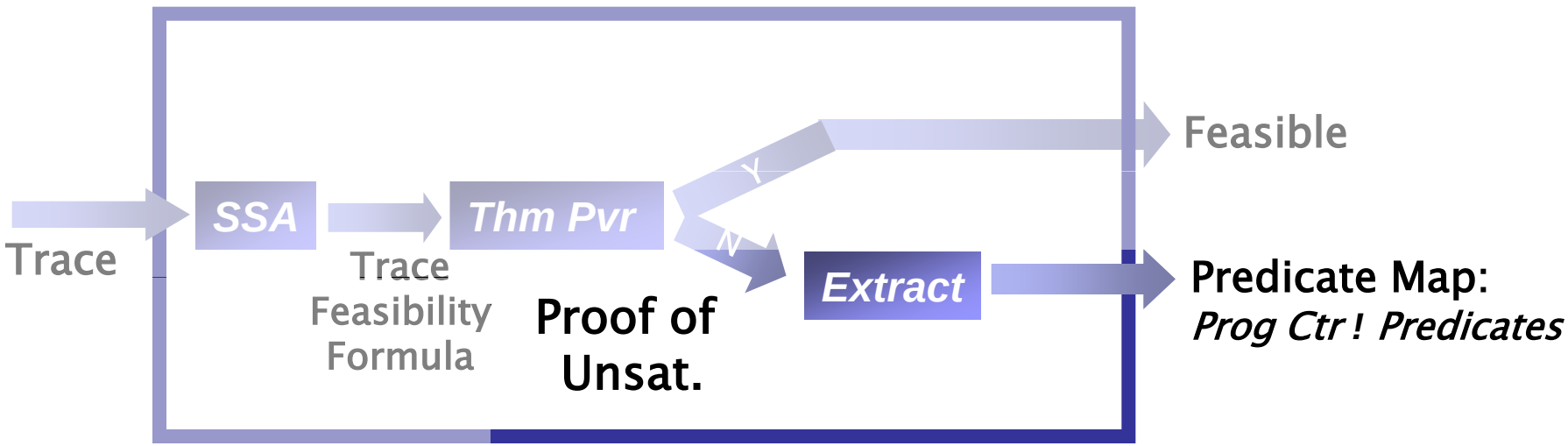
Counterexample Analysis



Q0: Is trace feasible ?

Q1: What predicates remove trace ?

Q2: Where are preds required ?



Proof of Unsatisfiability

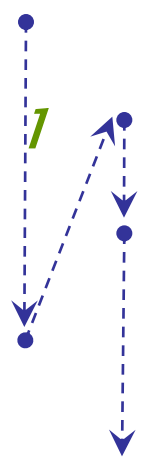
$x_1 = ctr_0$

$\wedge ctr_1 = ctr_0 + 1$

$\wedge y_1 = ctr_1$

$\wedge x_1 = i_0 - 1$

$\wedge y_1 \neq i_0$



Trace Formula

$$\dfrac{\dfrac{\dfrac{\dfrac{x_1 = ctr_0 \quad x_1 = i_0 - 1}{ctr_0 = i_0 - 1} \quad ctr_1 = ctr_0 + 1}{ctr_1 = i_0} \quad y_1 = ctr_1}{y_1 = i_0} \quad y_1 \neq i_0}{;}$$

Proof of Unsatisfiability

PROBLEM

Proof uses entire *history* of execution

- Information flows up and down

No *localized* or *state* information !

The Present State…

Trace

```
pc1: x = ctr
pc2: ctr = ctr + 1
pc3: y = ctr
pc4: assume(x = i-1)
pc5: assume(y ≠ i)
```

… is all the information the executing program has ***here***

State…

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

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

What Predicate is needed ?

Trace

```
pc1: x = ctr
pc2: ctr = ctr + 1
pc3: y = ctr
pc4: assume(x = i-1)
pc5: assume(y ≠ i)
```

Trace Formula (TF)

$$
\begin{aligned}
& x_1 = ctr_0 \\
\wedge\ & ctr_1 = ctr_0 + 1 \\
\wedge\ & y_1 = ctr_1 \\
\wedge\ & x_1 = i_0 - 1 \\
\wedge\ & y_1 \neq i_0
\end{aligned}
$$

State…

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

Predicate …

… implied by TF *prefix*

What Predicate is needed ?

Trace

```
pc1: x = ctr
pc2: ctr = ctr + 1
pc3: y = ctr
pc4: assume(x = i-1)
pc5: assume(y ≠ i)
```

Trace Formula (TF)

$$
\begin{aligned}
& x_1 = ctr_0 \\
\wedge\ & ctr_1 = ctr_0 + 1 \\
\wedge\ & y_1 = ctr_1 \\
\wedge\ & x_1 = i_0 - 1 \\
\wedge\ & y_1 \neq i_0
\end{aligned}
$$

State…

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

Predicate …

… implied by TF *prefix*

… on *common* variables

What Predicate is needed ?

Trace

```
pc1: x = ctr
pc2: ctr = ctr + 1
pc3: y = ctr
pc4: assume(x = i-1)
pc5: assume(y ≠ i)
```

Trace Formula (TF)

$$
\begin{aligned}
& x_1 = ctr_0 \\
\wedge\ & ctr_1 = ctr_0 + 1 \\
\wedge\ & y_1 = ctr_1 \\
\wedge\ & x_1 = i_0 - 1 \\
\wedge\ & y_1 \neq i_0
\end{aligned}
$$

State…

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

Predicate …

… implied by TF *prefix*

… on *common* variables

… & TF *suffix* is *unsatisfiable*

What Predicate is needed ?

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)`

Trace Formula (TF)

$$x_1 = ctr_0$$
$$\wedge \quad ctr_1 = ctr_0 + 1$$
$$\wedge \quad y_1 = ctr_1$$
$$\wedge \quad x_1 = i_0 - 1$$
$$\wedge \quad y_1 \neq i_0$$

State…

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

Predicate …

… implied by TF *prefix*

… on *common* variables

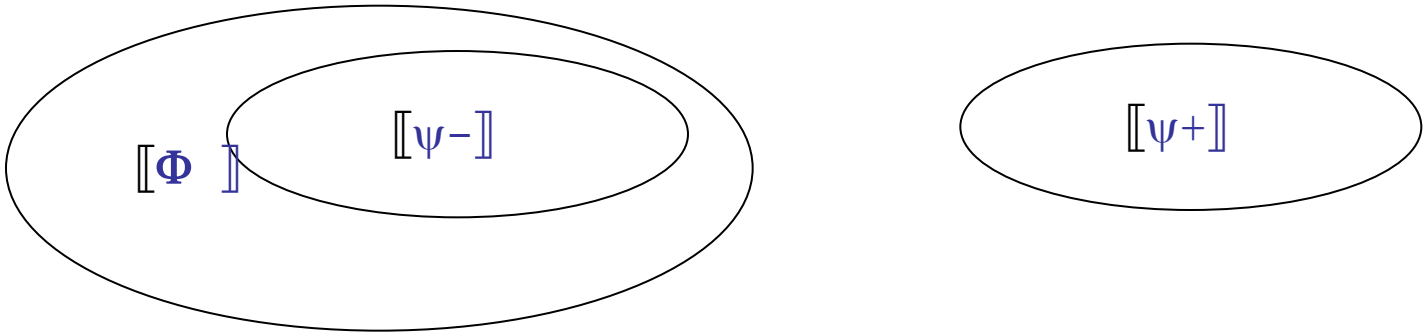
… & TF *suffix* is *unsatisfiable*

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***

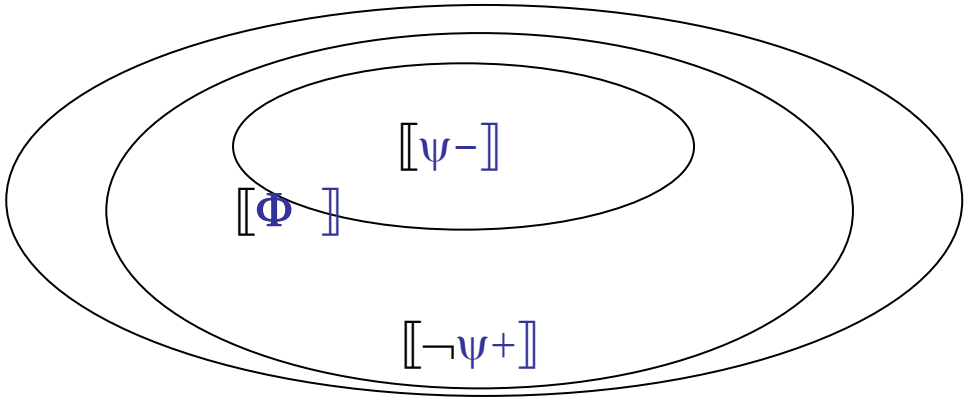


Craig's Interpolation Theorem (take 2)

Given formulas ψ^- , $\neg\psi^+$ s.t. ψ^- ***implies*** $\neg\psi^+$

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

1. ψ^- ***implies*** Φ ***implies*** $\neg\psi^+$
2. Φ has symbols ***common*** to ψ^-, $\neg\psi^+$



Examples of Craig's Interpolation

- $\psi^- = b \wedge (\neg b \vee c)$
  $\psi^+ = \neg c$
- $\psi^- = x_1 = ctr_0 \wedge ctr_1 = ctr_0 + 1 \wedge y_1 = ctr_1$
  $\psi^+ = x_1 = i_0 - 1 \wedge y_1 \neq i_0$
  - $y_1 = x_1 + 1$

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 only 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]

Interpolant = Predicate !

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

Interpolate $\Rightarrow$ Φ

Require:

1. Predicate *implied* by trace *prefix*
2. Predicate on *common* variables
   common = *current* value
3. Predicate & *suffix* yields a *contradiction*

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`	$\wedge \quad ctr_1 = ctr_0 + 1$
pc_3: `y = ctr`	$\wedge \quad y_1 = ctr_1$
pc_4: `assume(x = i-1)`	$\wedge \quad x_1 = i_0 - 1$
pc_5: `assume(y ≠ i)`	$\wedge \quad y_1 \neq i_0$

ψ^-

ψ^+

Interpolate $\Rightarrow$ Φ

$y_1 = x_1 + 1$

Require:

1. Predicate *implied* by trace *prefix*
2. Predicate on *common* variables
3. Predicate & *suffix* yields a *contradiction*

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`	$\wedge\ ctr_1 = ctr_0 + 1$	ψ^-
pc_3: `y = ctr`	$\wedge\ y_1 = ctr_1$	
pc_4: `assume(x = i-1)`	$\wedge\ x_1 = i_0 - 1$	ψ^+
pc_5: `assume(y ≠ i)`	$\wedge\ y_1 \neq i_0$	

Interpolate → Φ

$y_1 = x_1 + 1$

Predicate at pc_4:
$y = x+1$

Require:

1. Predicate *implied* by trace *prefix*
2. Predicate on *common* variables
3. Predicate & *suffix* yields a *contradiction*

Interpolant:

1. ψ^- *implies* Φ
2. Φ has symbols *common* to ψ^-, ψ^+
3. Φ Æ ψ^+ 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`	$\wedge \quad ctr_1 = ctr_0 + 1$
pc_3: `y = ctr`	$\wedge \quad y_1 = ctr_1$
pc_4: `assume(x = i-1)`	$\wedge \quad x_1 = i_0 - 1$
pc_5: `assume(y ≠ i)`	$\wedge \quad y_1 \neq i_0$

ψ^-

Interpolate → $x_1 = ctr_0$

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: $x = ctr - 1$

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

ψ^-

ψ^+

Interpolate → $x_1 = ctr_1 - 1$

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

Building Predicate Maps

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

ψ^-

ψ^+

Interpolate $\rightarrow$ $y_1 = i_0$

Predicate Map

pc_2: $x = ctr$

pc_3: $x = ctr-1$

pc_4: $y = x+1$

pc_5: $y = i$

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

Building Predicate Maps

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

Predicate Map

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

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

Plan

1. Motivation

2. Refinement using Traces
   - Simple
   - Procedure calls

3. Results

Traces with Procedure Calls

Trace **Trace Formula**

pc_1: $x_1 = 3$
pc_2: assume $(x_1>0)$
pc_3: $x_3 = f_1(x_1)$
pc_4: $y_1 = y_1$
pc_5: $y_3 = f_2(y_2)$
pc_6: $z_2 = z_1+1$
pc_7: $z_3 = 2*z_2$
pc_8: return z_3
pc_9: return y_3
pc_{10}: $x_4 = x_3+1$
pc_{11}: $x_5 = f_3(x_4)$
pc_{12}: assume $(w_1<5)$
pc_{13}: return w_1
pc_{14}: assume $x_4>5$
pc_{15}: assume $(x_1=x_3+2)$

$x_3 = f_1(x_1)$
$y_1 = y_1$
$y_3 = f_2(y_2)$
$z_2 = z_1+1$
$z_3 = 2*z_2$
return z_3
return y_3
$x_5 = f_3(x_4)$
assume $(w_1<5)$
return w_1

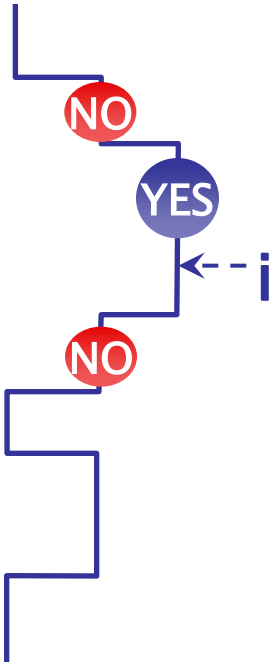
i

i

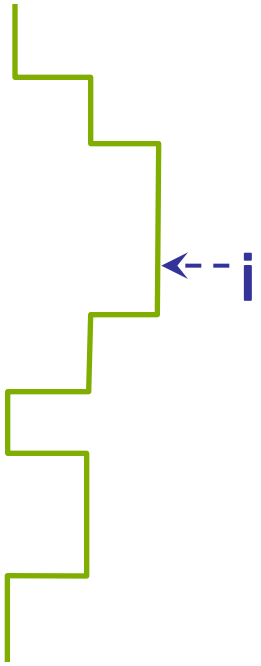
Find predicate needed at point i

Interprocedural Analysis

Trace



Trace Formula



Find predicate needed at point i

Require at each point i:

Well-scoped predicates

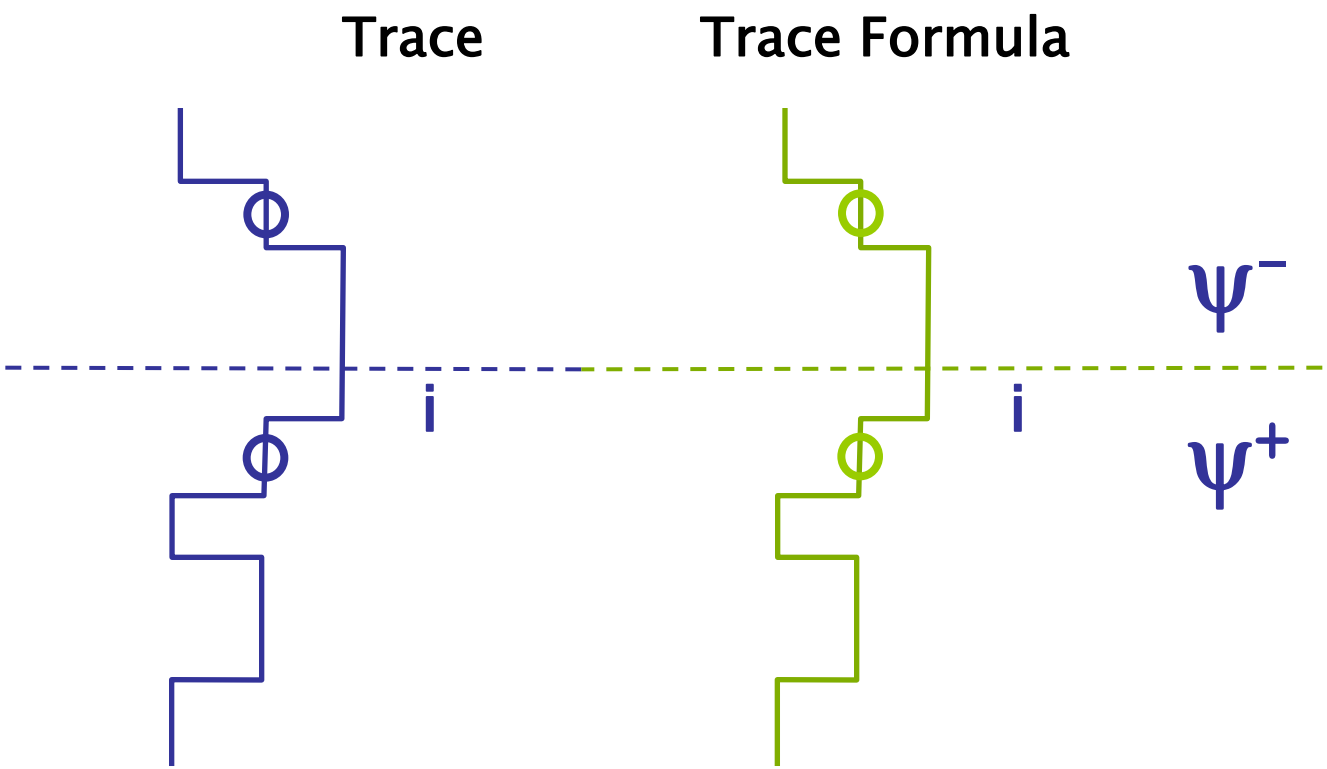
YES: Variables ***visible*** at i

NO: Caller's local variables

Procedure Summaries [Reps,Horwitz,Sagiv '95]

Polymorphic Predicate Abstraction [Ball,Millstein,Rajamani '02]

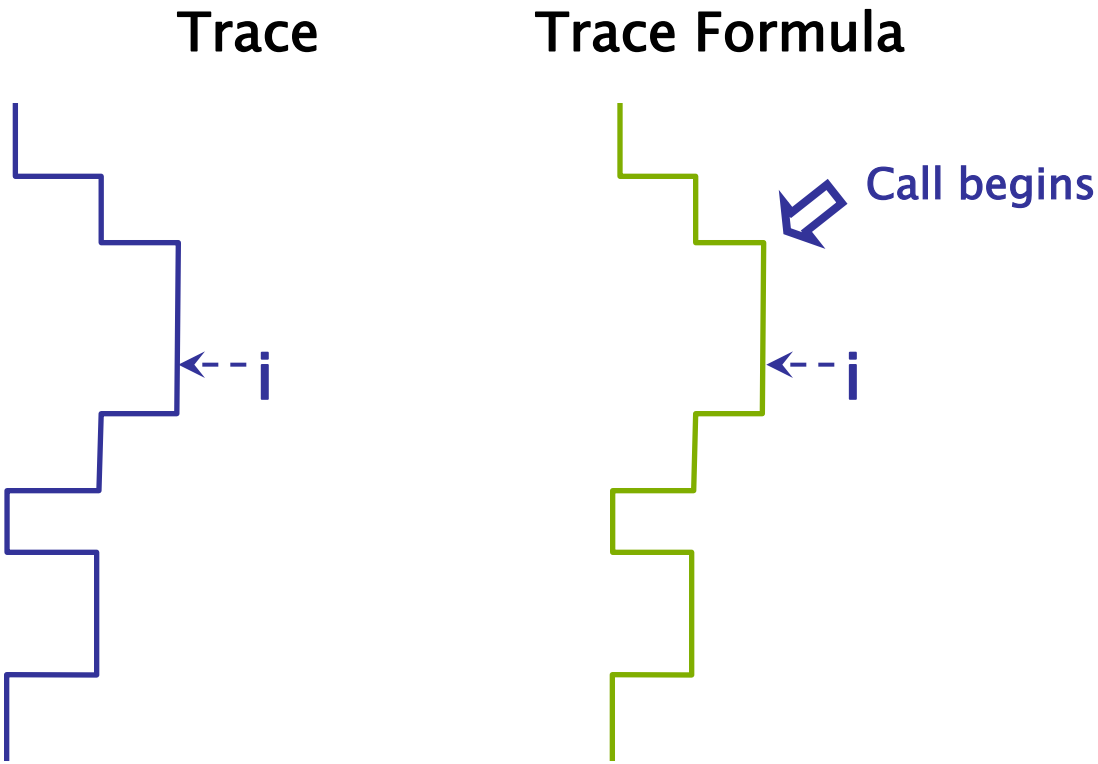
Problems with Cutting



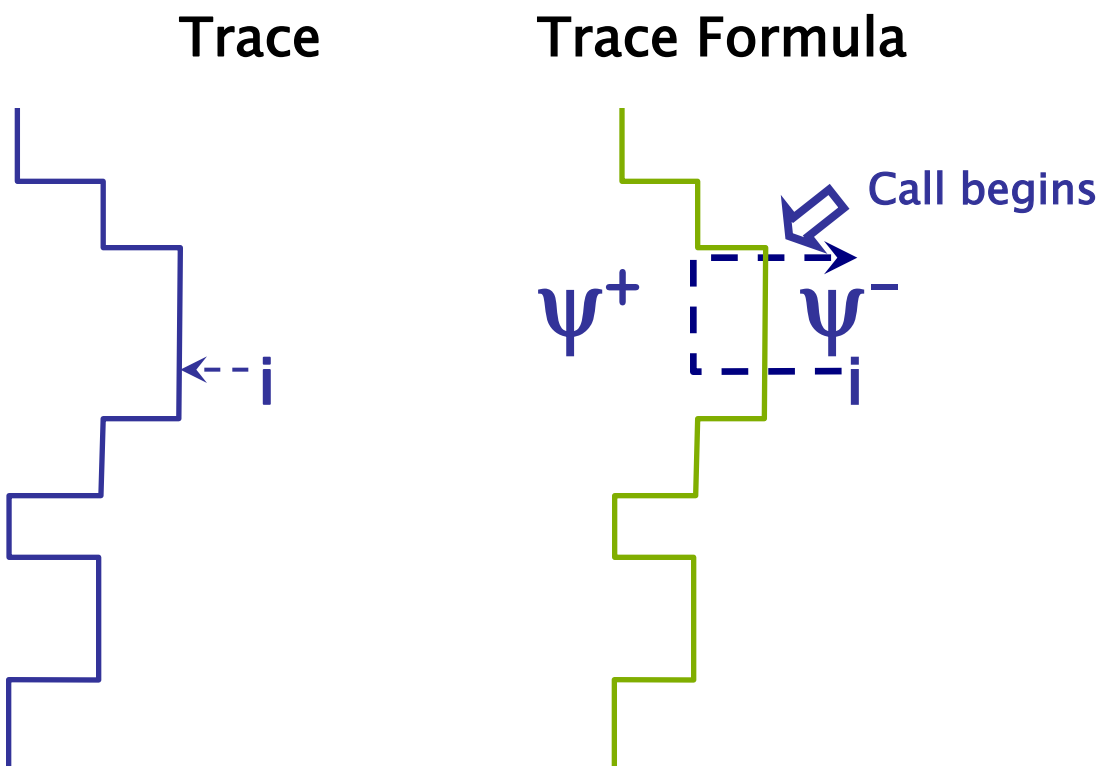
Caller variables common to ψ^- and ψ^+

- Unsuitable interpolant: not well-scoped

Interprocedural Cuts

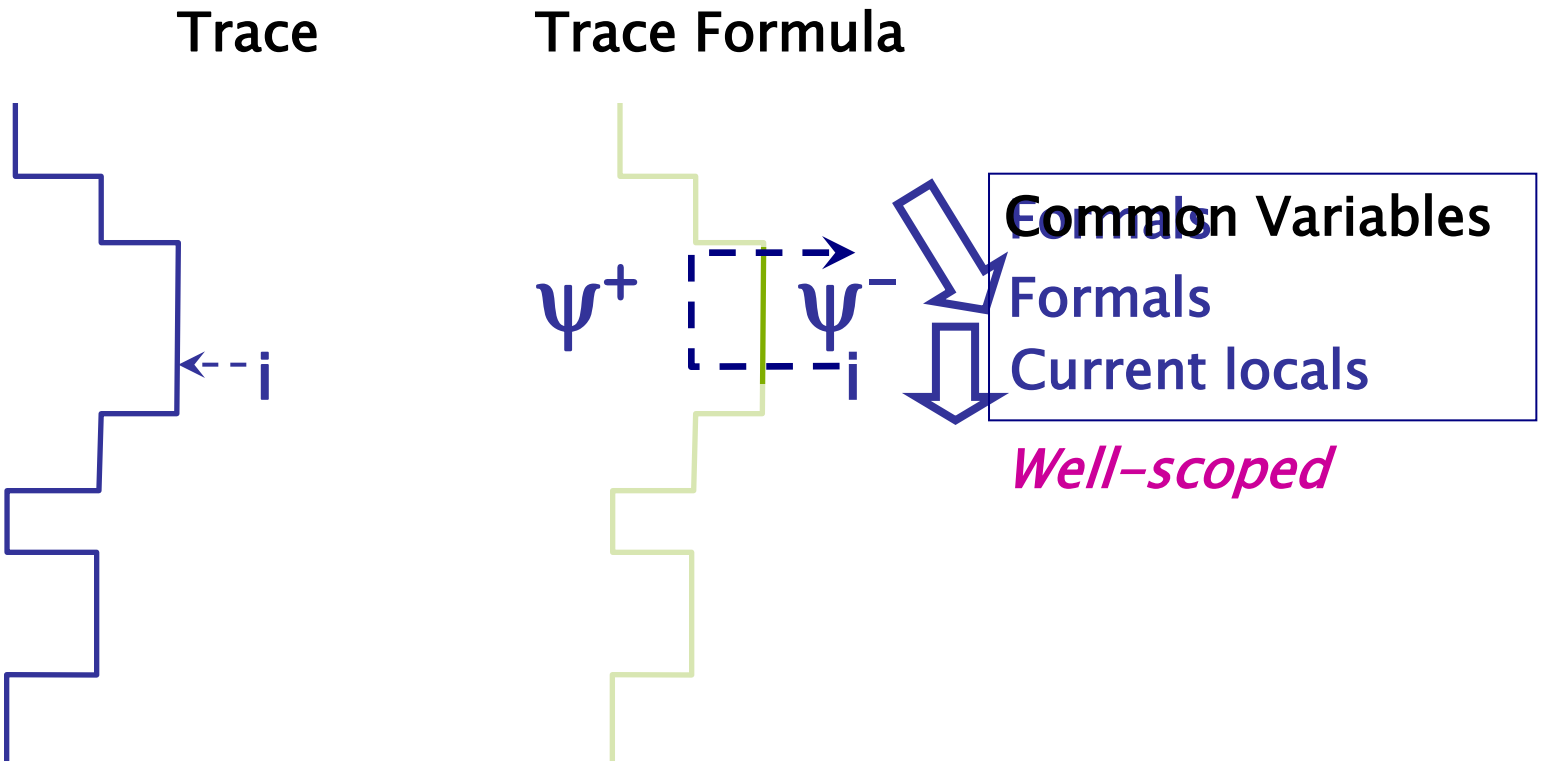


Interprocedural Cuts



Predicate at pc_i = Interpolant from cut i

Common Variables



Predicate at pc_i = Interpolant from i-cut

Plan

1. Motivation

2. Refinement using Traces
   - Simple
   - Procedure calls

3. Results

Implementation

- Algorithms implemented in BLAST
  - Verifier for C programs, Lazy Abstraction [POPL '02]
- FOCI : Interpolating decision procedure
- Examples:
  - Windows Device Drivers (DDK)
  - IRP Specification: 22 state FSM
  - Current: Security properties of Linux programs

Windows DDK

IRP

22 state

Results

Program	*LOC**	*Previous Time*	*New Time*	*Predicates*	
				Total	*Average*
kbfiltr	12k	1m12s	3m48s	72	6.5
floppy	17k	7m10s	25m20s	240	7.7
diskperf	14k	5m36s	13m32s	140	10
cdaudio	18k	20m18s	23m51s	256	7.8
parport	61k	*DNF*	*74m58s*	753	8.1
parclass	138k	*DNF*	*77m40s*	382	7.2

** Pre-processed*

Windows DDK

IRP

22 state

Localizing works…

Program	*LOC**	*Previous Time*	*New Time*	*Predicates*	
				Total	*Average*
kbfiltr	12k	1m12s	3m48s	72	*6.5*
floppy	17k	7m10s	25m20s	240	*7.7*
diskperf	14k	5m36s	13m32s	140	*10*
cdaudio	18k	20m18s	23m51s	256	*7.8*
parport	61k	DNF	74m58s	753	*8.1*
parclass	138k	DNF	77m40s	382	*7.2*

** Pre-processed*

Conclusion

- Scalability *and* Precision by *localizing*
- Craig Interpolation
  - Interprocedural cuts give well-scoped predicates

- Some Current and Future Work:
  - Multithreaded Programs
    - Project local info of thread to predicates over globals
  - Hierarchical trace analysis

Limitations of CEGAR

- Limited to powerset/relational abstract domains
- Interpolant computations
- Interactions with widening
- Starting on the right foot
- Unnecessary refinement steps
- Long and infinite number of refinement steps
- Long traces

Unnecessary Refinements

```
x = 0
while (x < 10^6) do
    x = x + 1
assert x < 100
```

Unsuccessful Refinement Set

```
x = malloc();
y = x ;
while (…)
      t = malloc();
      t->next = x
      x = t;
…
while (x !=y) do
      assert x != null;
      x = x->next
```

Long Traces

```
Example ( ) {
1:c = 0;
2:for(i=1;i<1000;i++)
3:   c = c + f(i);

4:if (a>0) {
5:   if (x==0) {
ERR:  ;
     }
  }
}
```

- Assume f always terminates
- ERR is reachable
  - a and x are unconstrained
- Any feasible path to error must unroll the loop 1000 times AND find feasible paths through f
- Any other path must be dismissed as a false positive

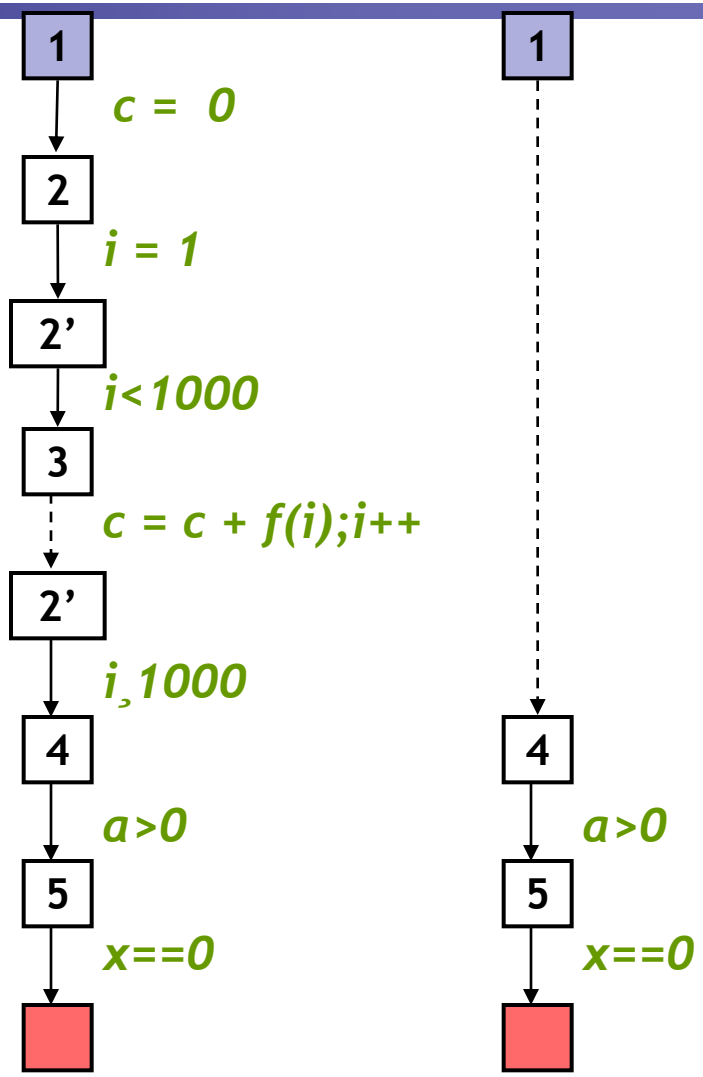
Long Traces

```
Example ( ) {
1:c = 0;
2:for(i=1;i<1000;i++)
3:   c = c + f(i);

4:if (a>0) {
5:   if (x==0) {
ERR:  ;
     }
  }
}
```

- Intuitively, the for loop is irrelevant
- ERR reachable as long as there exists some path from 2 to 4 that does not modify a or x
- Can we use static analysis to precisely report a statement is reachable *without* finding a feasible path?

Long Traces

```
Example ( ) {
1:c = 0;
2:for(i=1;i<1000;i++)
3:   c = c + f(i);

4:if (a>0) {
5:   if (x==0) {
ERR:  ;
      }
  }
}
```



Path Slice (PLDI'05)

The path slice of a program path π is a subsequence of the edges of π such that if the sequence of operations along the subsequence is:

1. infeasible, then π is infeasible, and
2. feasible, then the last location of π is reachable (but not necessarily along π)