

Counter Example Guided Refinement CEGAR

Mooly Sagiv

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

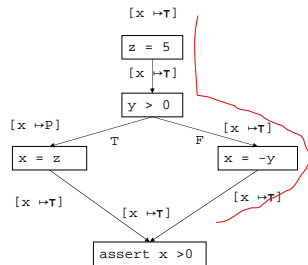
- Precision vs. Scalability
- Sometimes precision improves scalability
- Specialize the abstraction for the desired property

Counter Example Guided Refinement (CEGAR)

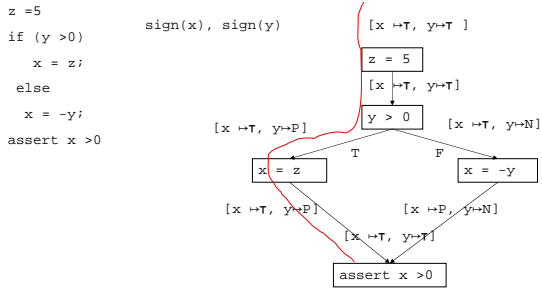
- Run the analysis with a simple abstract domain
- When the analysis verifies the property declare done
- If the analysis reports an error employs a theorem prover to identify if the error is feasible
 - If the error is feasible generate a concrete trace
 - If the error is spurious refine the abstract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

A Simple Example

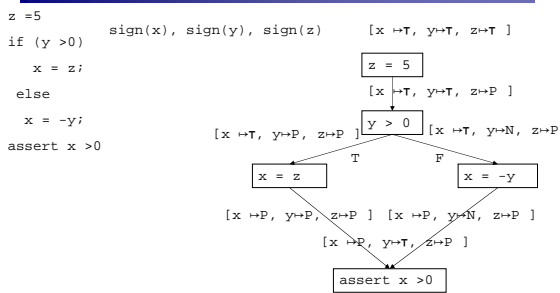
```
z = 5
sign(x)
if (y > 0)
  x = z;
else
  x = -y;
assert x > 0
```



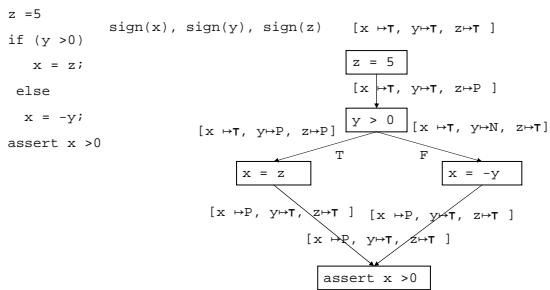
A Simple Example



A Simple Example



Simple Example (local abstractions)



Plan

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

Abstractions from Proofs



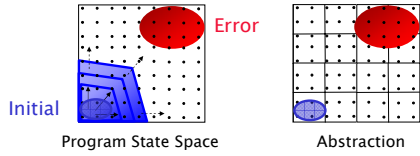
cadence

Thomas A. Henzinger
Ranjit Jhala
[UC Berkeley]
Rupak Majumdar
[UC Los Angeles]
Kenneth L. McMillan
[Cadence Berkeley Labs]

Scalable Program Verification

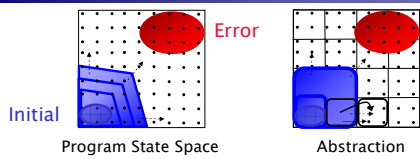
- *Little theorems about big programs*
 - Partial Specifications
 - Device drivers use kernel API correctly
 - Applications use root privileges correctly
 - Behavioral, path-sensitive properties

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

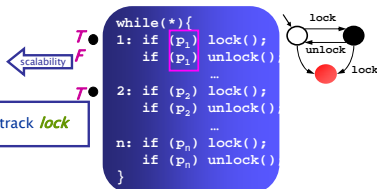
- Fixed set of predicates Pred
- The relational domain is $\langle \mathcal{P}(\text{Pred}), \emptyset, \text{Pred}, \cup, \cap \rangle$
 - Join is set union
 - State space explosion
- Special case of canonic abstraction

Scalability vs. Verification



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

Example

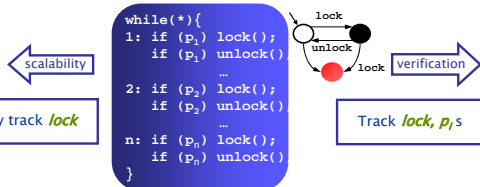


Bogus Counterexample
- Must *correlate* branches

Predicate p_i makes trace *abstractly infeasible*

p_i required for verification

Example



Bogus Counterexample
- Must *correlate* branches

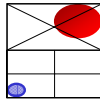
State Explosion
- $> 2^n$ distinct states
- Intractable

How can we get scalable verification ?

By Localizing Precision

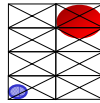
```

while (*) {
P1 { 1: if (p1) lock();
      if (p1) unlock();
      ...
P2 { 2: if (p2) lock();
      if (p2) unlock();
      ...
Pn { n: if (pn) lock();
      if (pn) unlock();
      }
}
    
```



Preds. Used locally

Ex: $2 \times n$ states

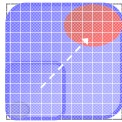


Preds. used globally

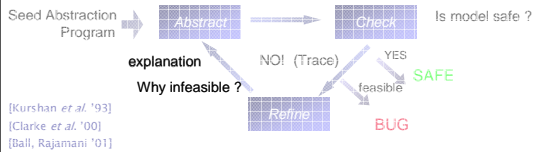
Ex: 2^n states

Q2: *Where* are the predicates required ?

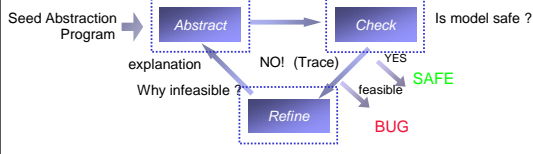
Counterexample Guided Refinement



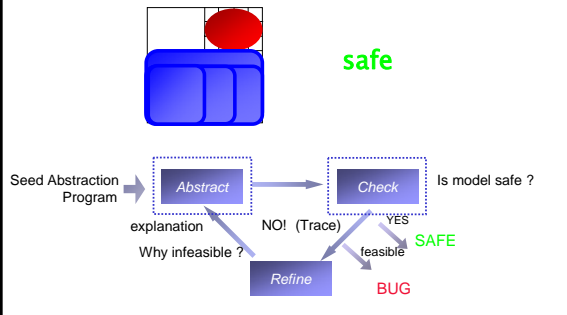
1. *What predicates* remove trace ?
 - Make it abstractly infeasible
2. *Where* are predicates needed ?



Counterexample Guided Refinement

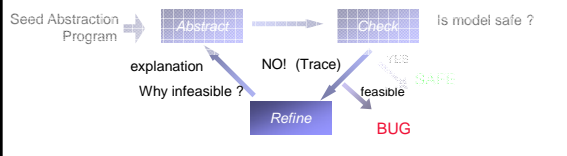


Counterexample Guided Refinement



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

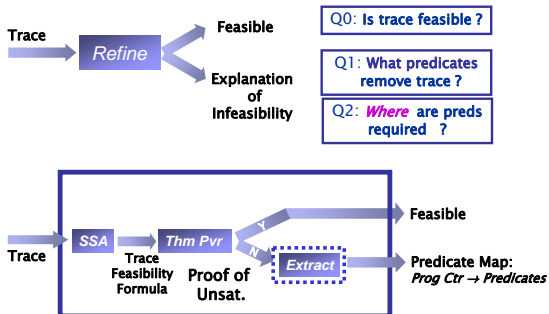
Trace Formulas

- 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

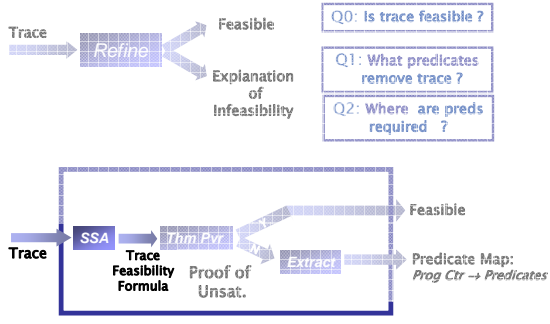
Representing States as *Formulas*

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

Counterexample Analysis



Counterexample Analysis



Traces

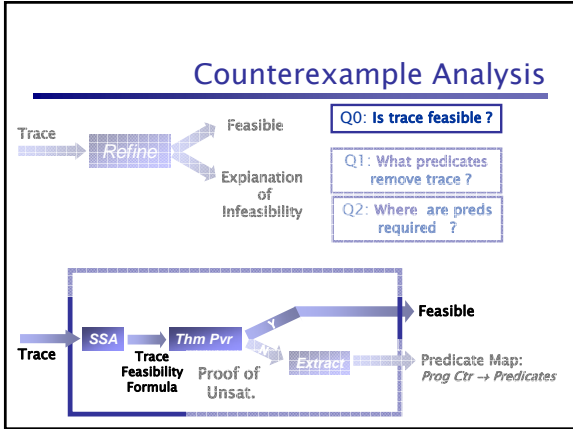
<pre> pc1: x = ctr; pc2: ctr = ctr + 1; pc3: y = ctr; pc4: if (x = i-1){ pc5: if (y != i){ ERROR; } </pre>	<pre> pc1: x = ctr pc2: ctr = ctr + 1 pc3: y = ctr pc4: assume(x = i-1) pc5: assume(y != i) </pre>	$\leftarrow y = x + 1$
------------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------	------------------------

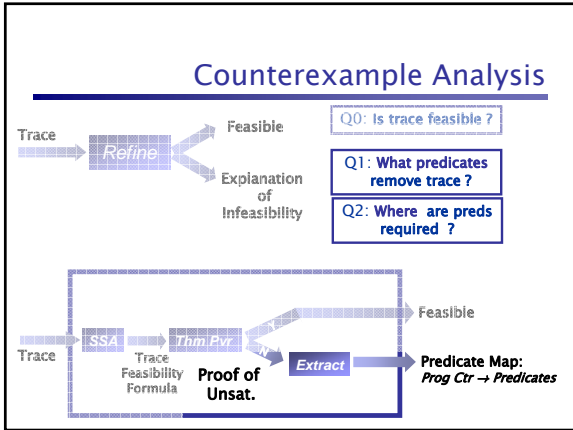
Trace Feasibility Formulas

<pre> pc1: x = ctr pc2: ctr = ctr+1 pc3: y = ctr pc4: assume(x=i-1) pc5: assume(y!=i) </pre>	<pre> pc1: x1 = ctr0 pc2: ctr1 = ctr0+1 pc3: y1 = ctr1 pc4: assume(x1=i0-1) pc5: assume(y1!=i0) </pre>	<pre> x1 = ctr0 ^ ctr1 = ctr0 + 1 ^ y1 = ctr1 ^ x1 = i0 - 1 ^ y1 != i0 </pre>
Trace	SSA Trace	Trace Feasibility Formula

Theorem: Trace is *Feasible* \Leftrightarrow TFF is *Satisfiable*

Compact Verification Conditions [Flanagan,Saxe '00]





Proof of Unsatisfiability

Trace Formula

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

Proof of Unsatisfiability

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

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)

```

x1 = ctr0
∧ ctr1 = ctr0 + 1
∧ y1 = ctr1
∧ x1 = i0 - 1
∧ y1 ≠ i0
    
```

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)

```

x1 = ctr0
∧ ctr1 = ctr0 + 1
∧ y1 = ctr1
∧ x1 = i0 - 1
∧ y1 ≠ i0
    
```

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	Trace Formula (TF)
$pc_1: x = ctr$	$x_1 = ctr_0$
$pc_2: ctr = ctr + 1$	$\wedge ctr_1 = ctr_0 + 1$
$pc_3: y = ctr$	$\wedge y_1 = ctr_1$
$pc_4: assume(x = i-1)$	$\wedge x_1 = i_0 - 1$
$pc_5: assume(y \neq i)$	$\wedge y_1 \neq i_0$
State...	Predicate ...
1. ... after executing trace <i>prefix</i>	... implied by TF <i>prefix</i>
2. ... has <i>present values</i> of variables	... on <i>common</i> variables
3. ... makes trace <i>suffix</i> infeasible	... & TF <i>suffix</i> is <i>unsatisfiable</i>

What Predicate is needed ?

Trace	Trace Formula (TF)
$pc_1: x = ctr$	$x_1 = ctr_0$
$pc_2: ctr = ctr + 1$	$\wedge ctr_1 = ctr_0 + 1$
$pc_3: y = ctr$	$\wedge y_1 = ctr_1$
$pc_4: assume(x = i-1)$	$\wedge x_1 = i_0 - 1$
$pc_5: assume(y \neq i)$	$\wedge y_1 \neq i_0$
State...	Predicate ...
1. ... after executing trace <i>prefix</i>	... implied by TF <i>prefix</i>
2. ... knows <i>present values</i> of variables	... on <i>common</i> variables
3. ... makes trace <i>suffix</i> infeasible	... & TF <i>suffix</i> is <i>unsatisfiable</i>

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*

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 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
$pc_1: x = ctr$	$x_i = ctr_0$
$pc_2: ctr = ctr + 1$	$\wedge ctr_i = ctr_0 + i$
$pc_3: y = ctr$	$\wedge y_i = ctr_i$
$pc_4: \text{assume}(x = i-1)$	$\wedge x_i = i_0 - 1$
$pc_5: \text{assume}(y \neq i)$	$\wedge y_i \neq i_0$

Ψ^- (above pc_2)
 Ψ^+ (below pc_4)
 Φ (to the right of pc_3)
 Interpolate \rightarrow

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
$pc_1: x = ctr$	$x_1 = ctr_0$
$pc_2: ctr = ctr + 1$	$\wedge ctr_1 = ctr_0 + 1$
$pc_3: y = ctr$	$\wedge y_1 = ctr_1$
$pc_4: \text{assume}(x = i-1)$	$\wedge x_1 = i_0 - 1$
$pc_5: \text{assume}(y \neq i)$	$\wedge y_1 \neq i_0$

Interpolate Ψ^- Ψ^+ 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
$pc_1: x = ctr$	$x_1 = ctr_0$
$pc_2: ctr = ctr + 1$	$\wedge ctr_1 = ctr_0 + 1$
$pc_3: y = ctr$	$\wedge y_1 = ctr_1$
$pc_4: \text{assume}(x = i-1)$	$\wedge x_1 = i_0 - 1$
$pc_5: \text{assume}(y \neq i)$	$\wedge y_1 \neq i_0$

Predicate at pc_4 :
 $y = x + 1$

Interpolate Ψ^- Ψ^+ 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*

Building Predicate Maps

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

Predicate Map
 $pc_2: x = ctr$

Interpolate Ψ^- Ψ^+ $x_1 = ctr_0$

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

Building Predicate Maps

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

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

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

Ψ^- Interpolate $y_1 = i_0$
 Ψ^+

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

Building Predicate Maps

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

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

Plan

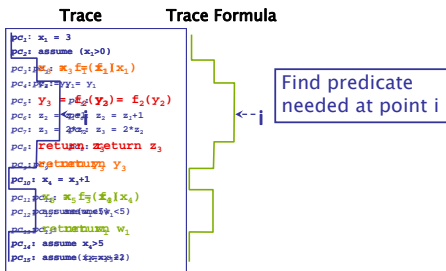
1. Motivation

2. Refinement using Traces

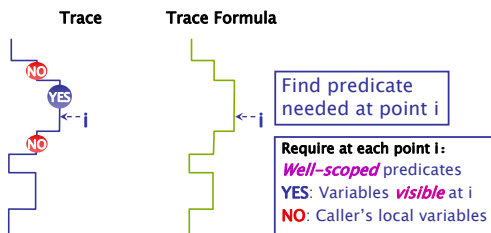
- Simple
- Procedure calls

3. Results

Traces with Procedure Calls

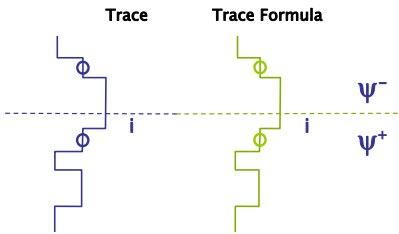


Interprocedural Analysis



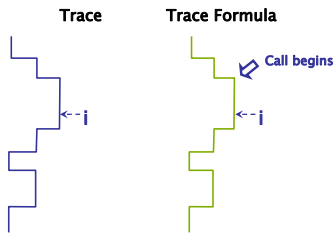
Procedure Summaries [Reps,Horwitz,Sagiv '95]
 Polymorphic Predicate Abstraction [Ball,Millstein,Rajamani '02]

Problems with Cutting

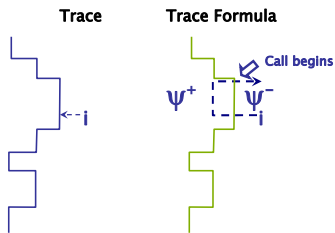


• *Caller variables* common to ψ^- and ψ^+
 • Unsuitable interpolant: not well-scoped

Interprocedural Cuts

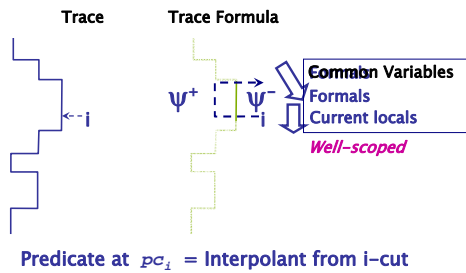


Interprocedural Cuts



Predicate at pC_i = Interpolant from cut i

Common Variables



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

BLAST

Berkeley Lazy Abstraction
Software * Tool

www.eecs.berkeley.edu/~blast/

SLAM

- A Microsoft tool for checking safety of device drivers
- Inspired BLAST

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 < 106) 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

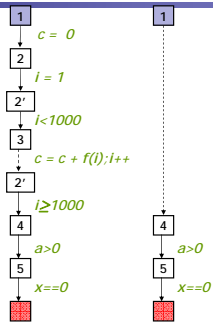
Long Traces

```
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)
