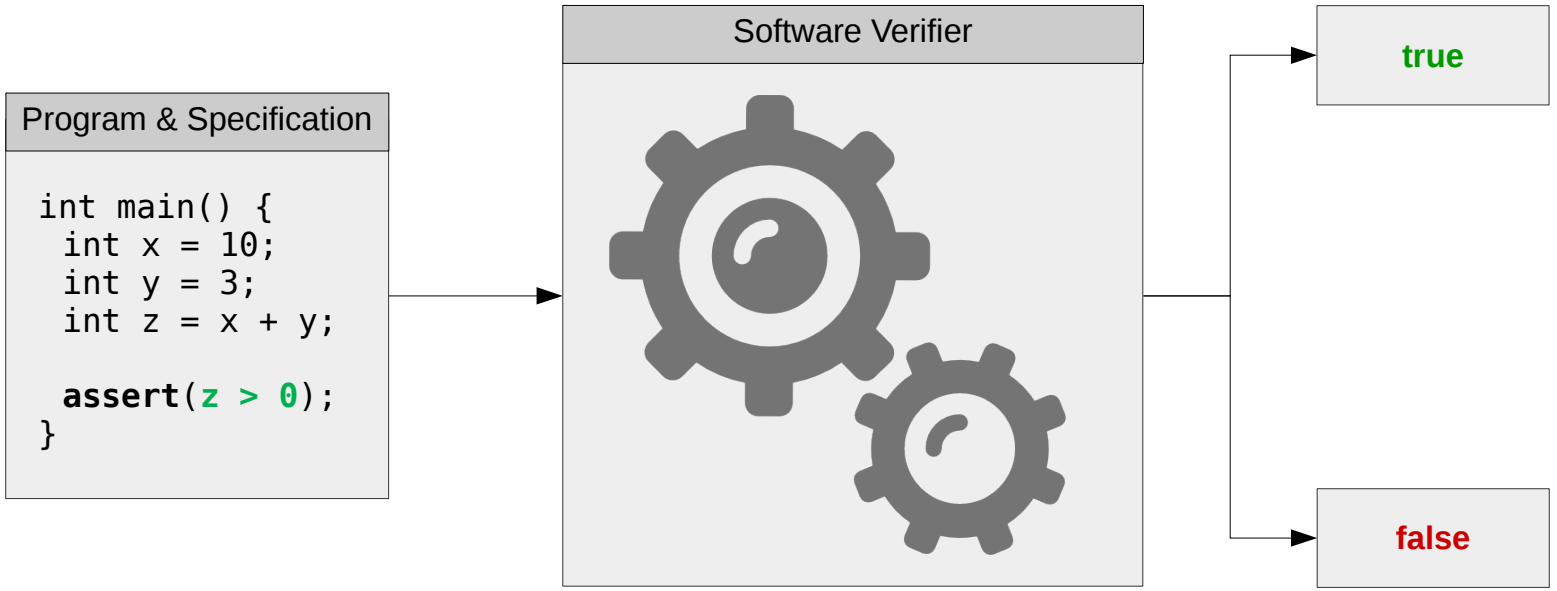
Effective Approaches to Abstraction Refinement for Automatic Software Verification

Stefan Löwe

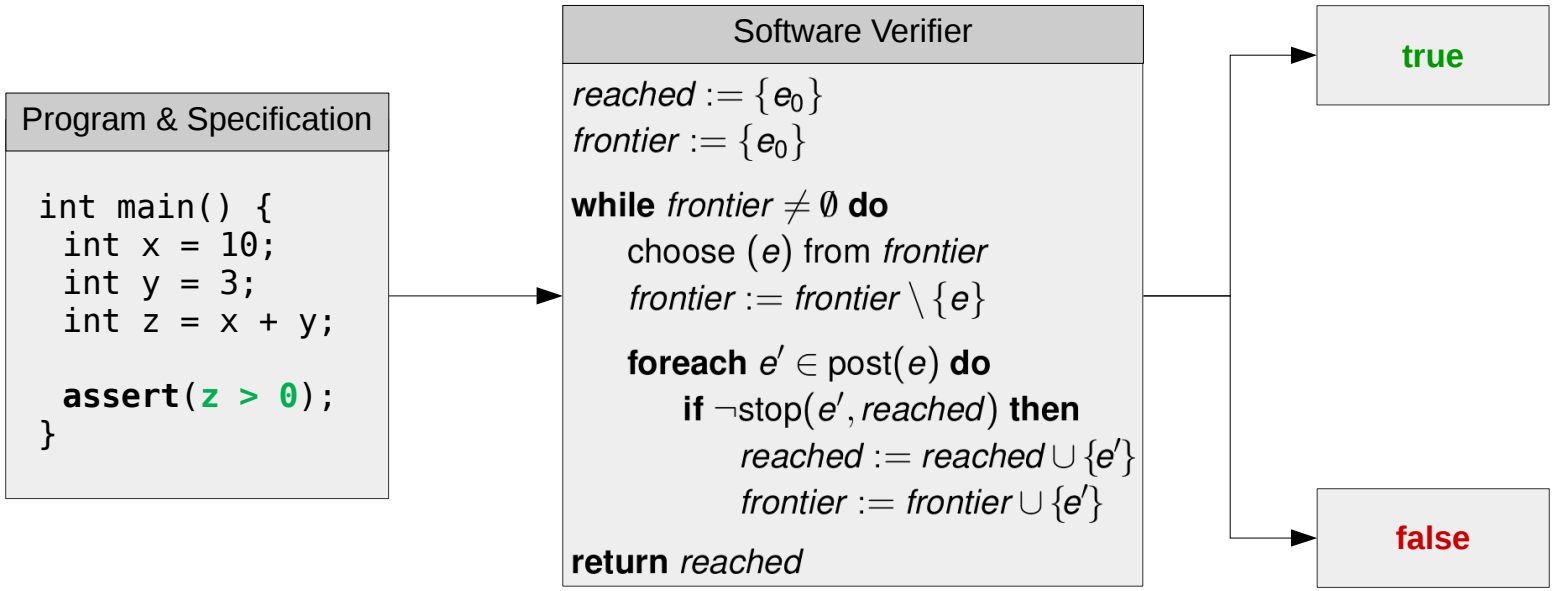
Faculty of Computer Science and Mathematics
—
University of Passau

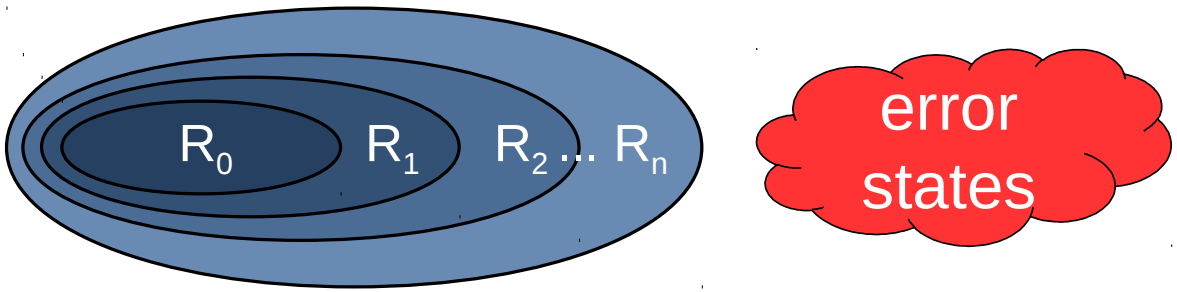
March 24, 2017

Automatic Software Verification



Automatic Software Verification





Motivation

Statement

We want software verification

Statement

We want software verification that is automatic

Statement

We want software verification that is automatic, as well as effective

Software is Full of Devastating, Critical Bugs


“Heartbleed” allows sensitive data theft


“Shellshock” allows arbitrary code execution


“goto-fail” eases man-in-the-middle attack


“Ghost” allows arbitrary code execution


“Stagefright” allows arbitrary code execution

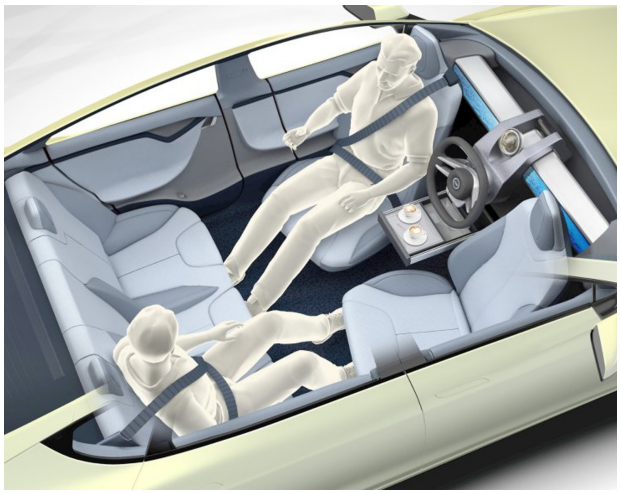

New vulnerabilities of your personal computers on a daily basis

Some even may have Catastrophic Consequences



Some even may have Catastrophic Consequences




?

Motivation

We want software verification

Agreed

Statement

We want software verification that is automatic

Statement

We want software verification that is automatic, as well as effective

Doing it Manually is Impossible

Motivation

We want software verification

Agreed

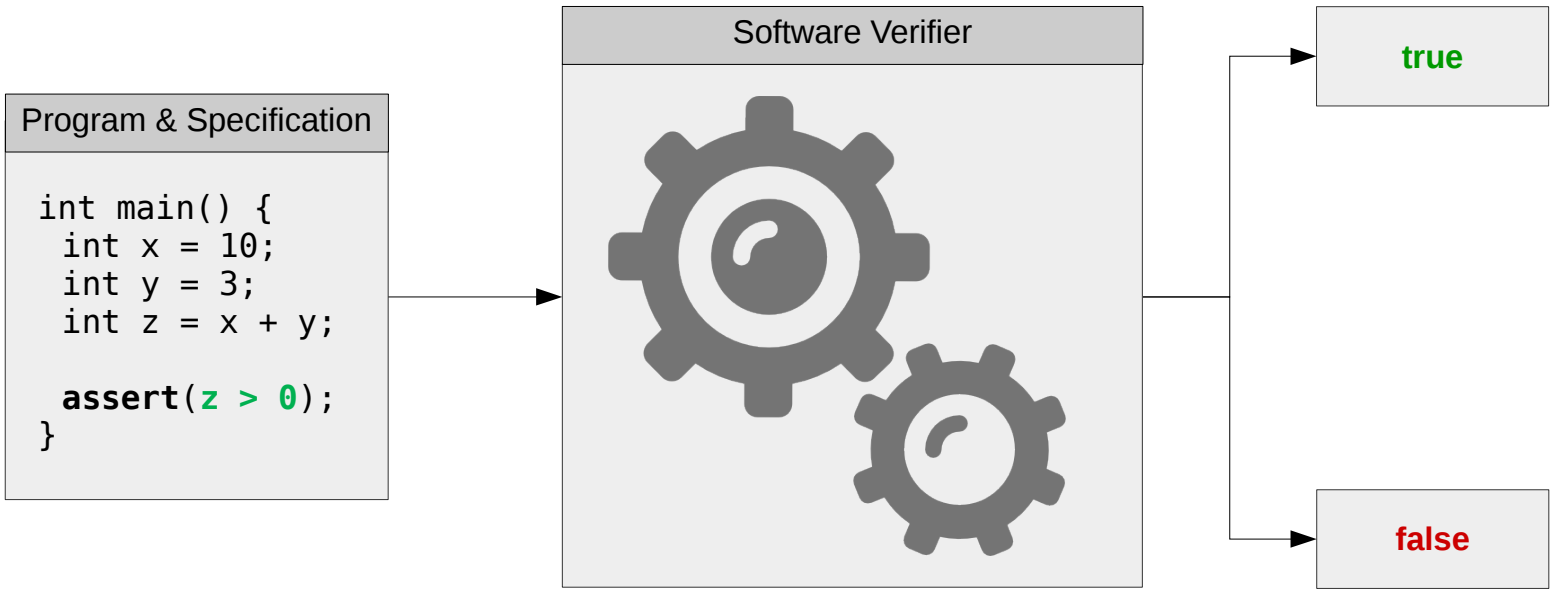
We want software verification that is automatic

Agreed

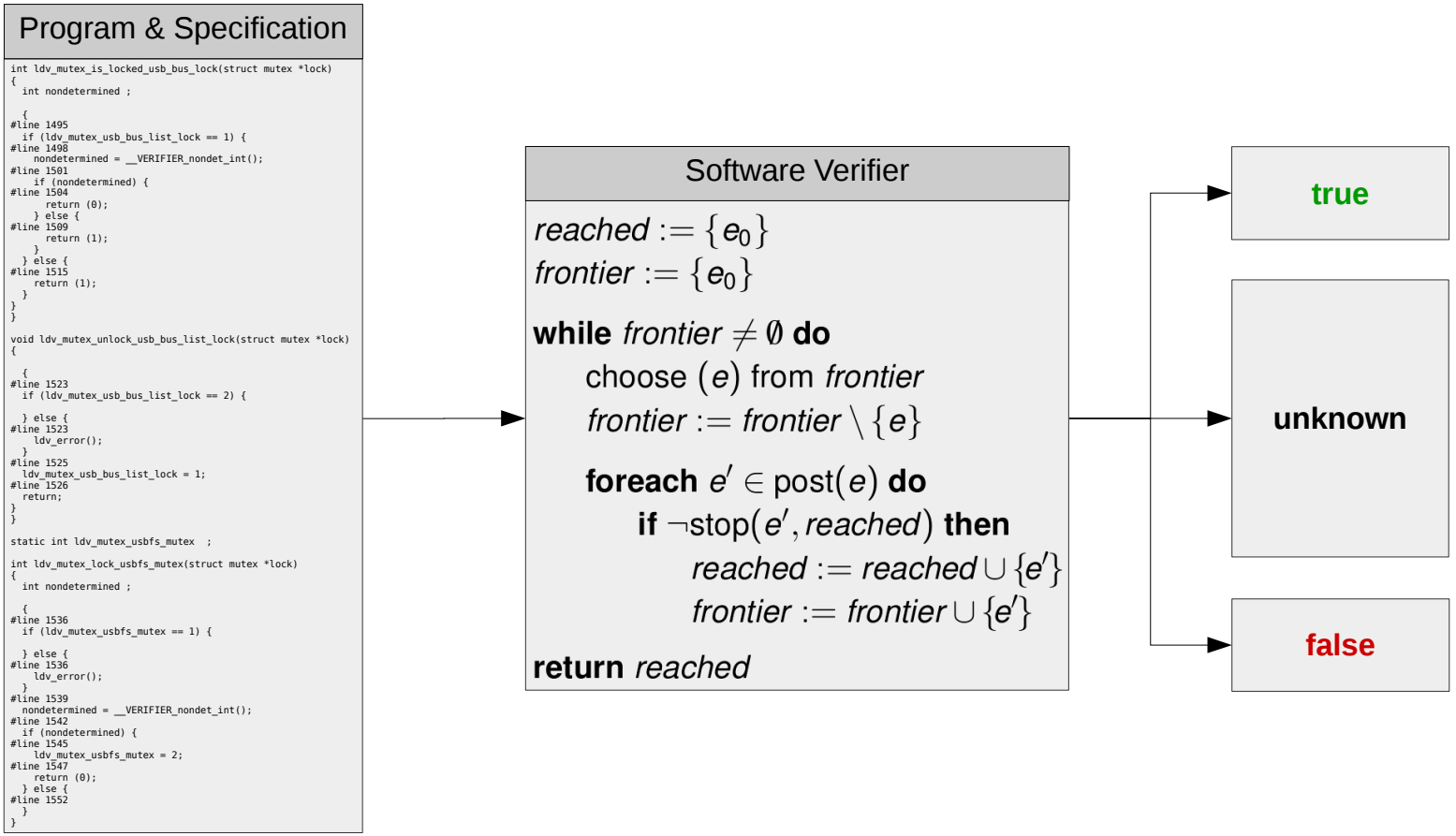
Statement

We want software verification that is automatic, as well as effective

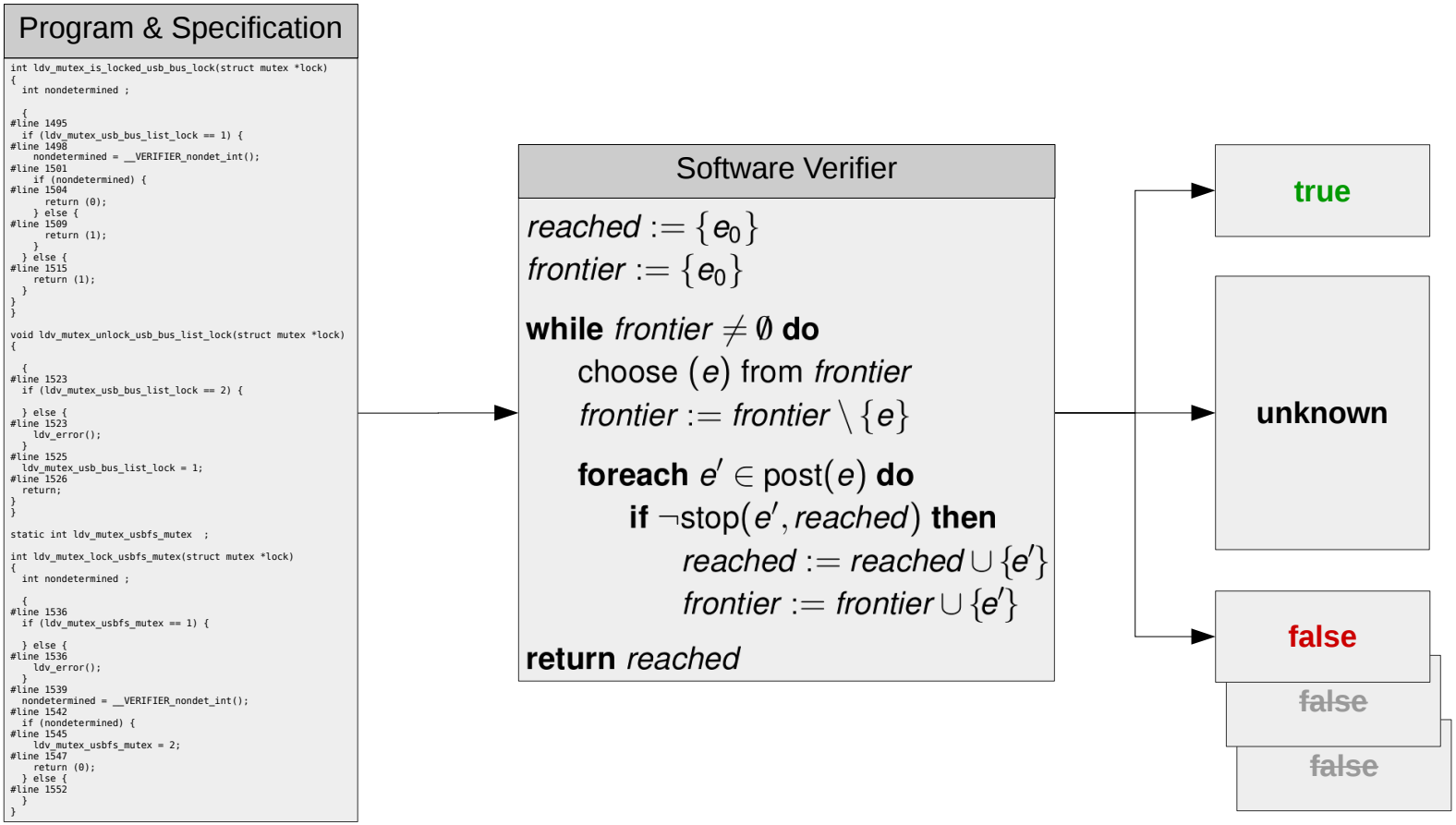
Automatic Software Verification



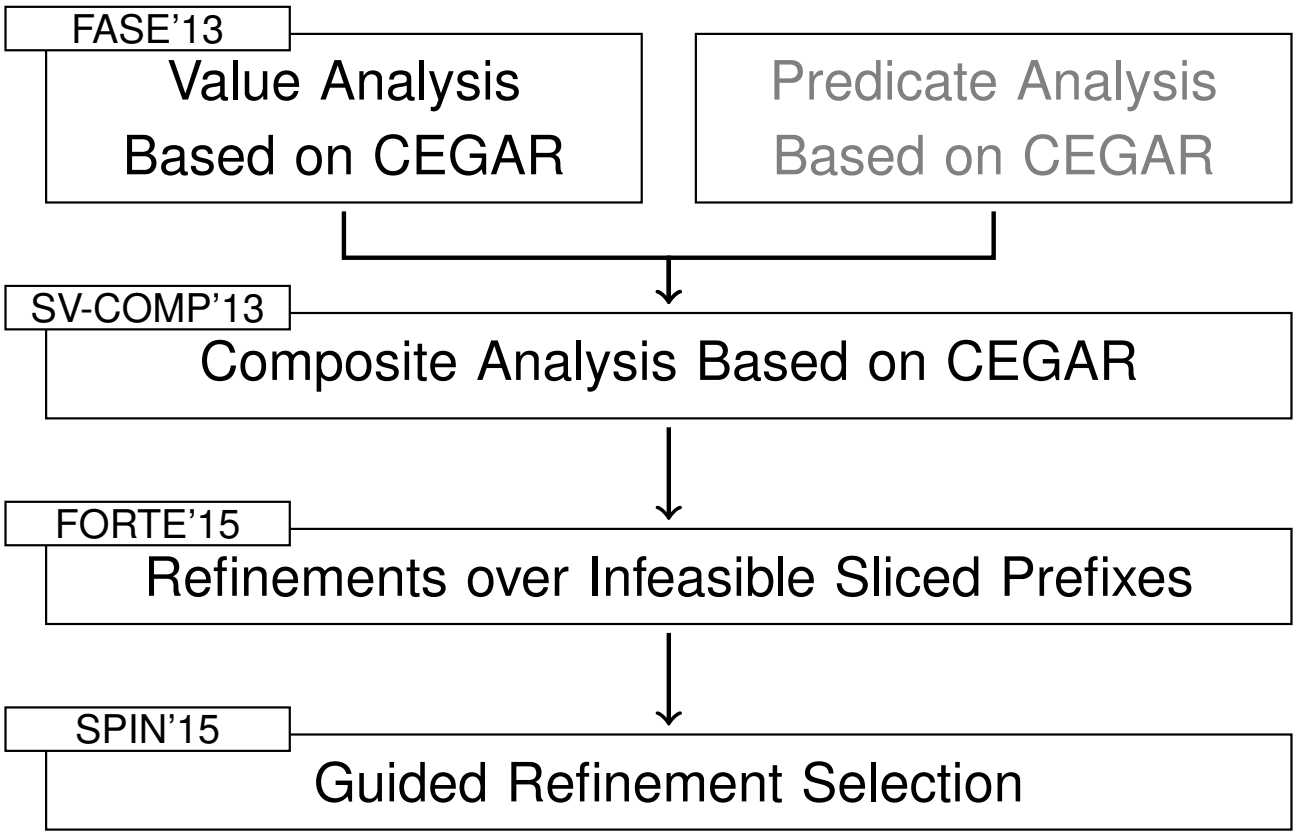
Automatic Software Verification for Real-World Programs



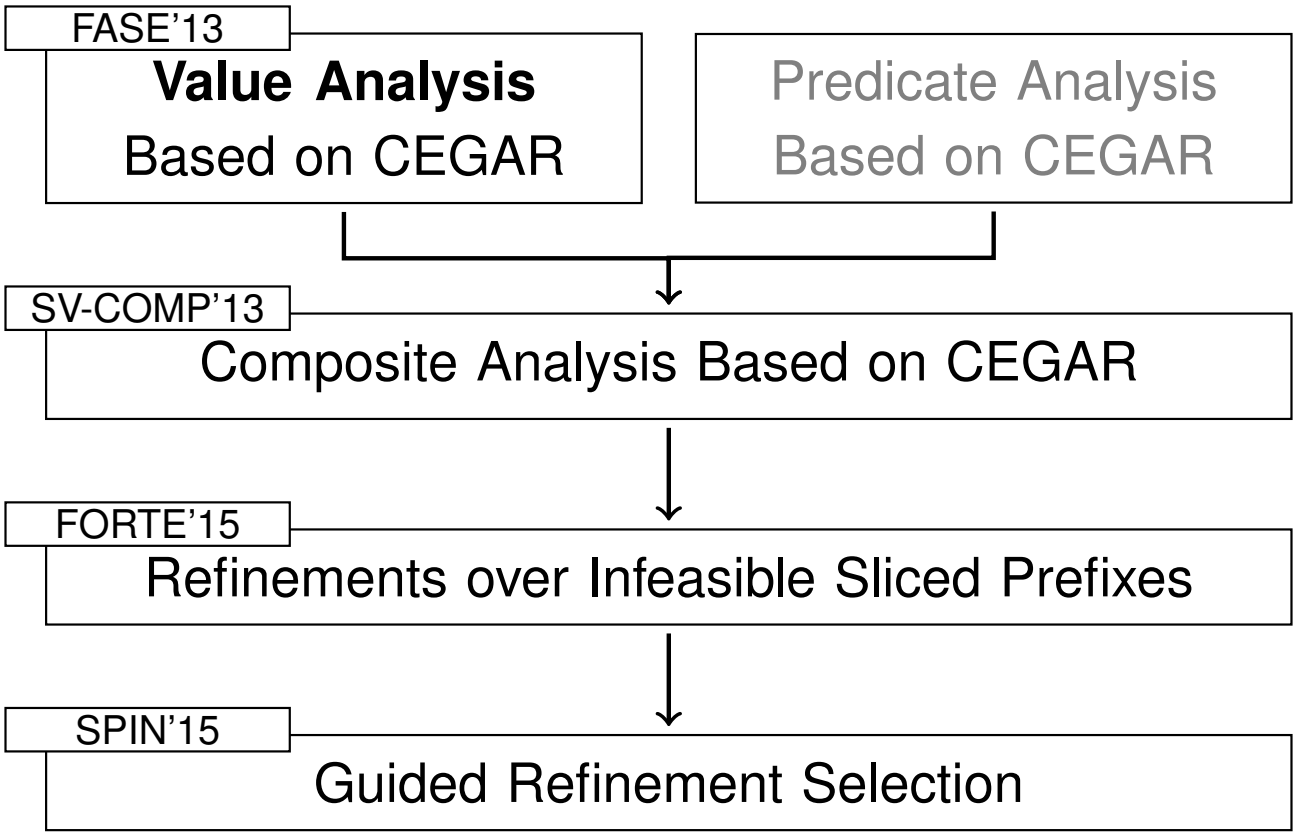
Automatic Software Verification for Real-World Programs



Contributions



Outline


FASE'13
Value Analysis
Based on CEGAR
Predicate Analysis
Based on CEGAR
SV-COMP'13
Composite Analysis Based on CEGAR
FORTE'15
Refinements over Infeasible Sliced Prefixes
SPIN'15
Guided Refinement Selection

Design Decisions and Characteristics

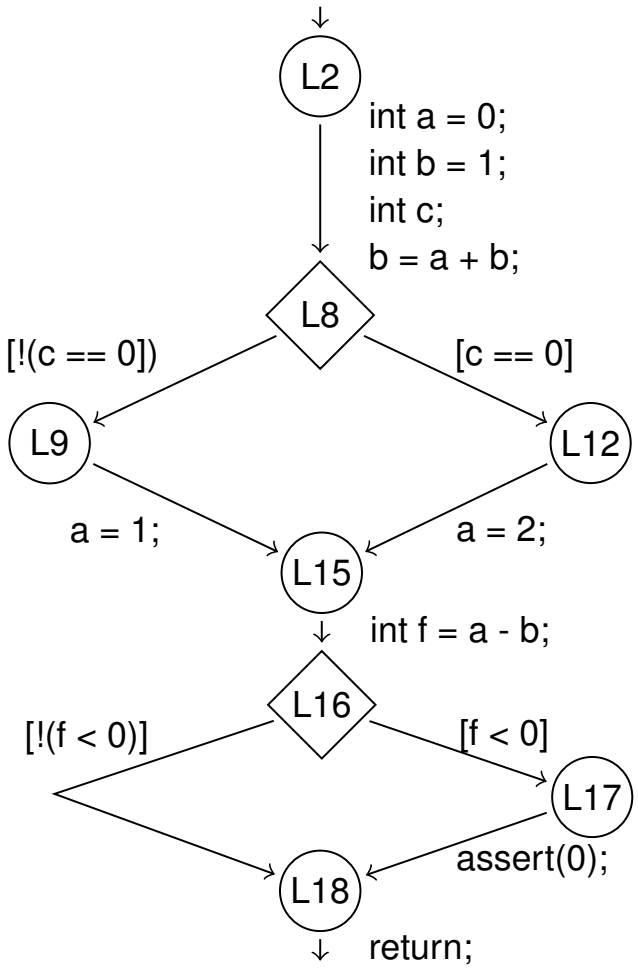
- Contrasts the predominant symbolic analyses
- Does not rely on SAT or SMT solvers
- Tracks concrete assignments of program variables
- Successor computations are simple arithmetic evaluations

Design Decisions and Characteristics

- Contrasts the predominant symbolic analyses
- Does not rely on SAT or SMT solvers
- Tracks concrete assignments of program variables
- Successor computations are simple arithmetic evaluations

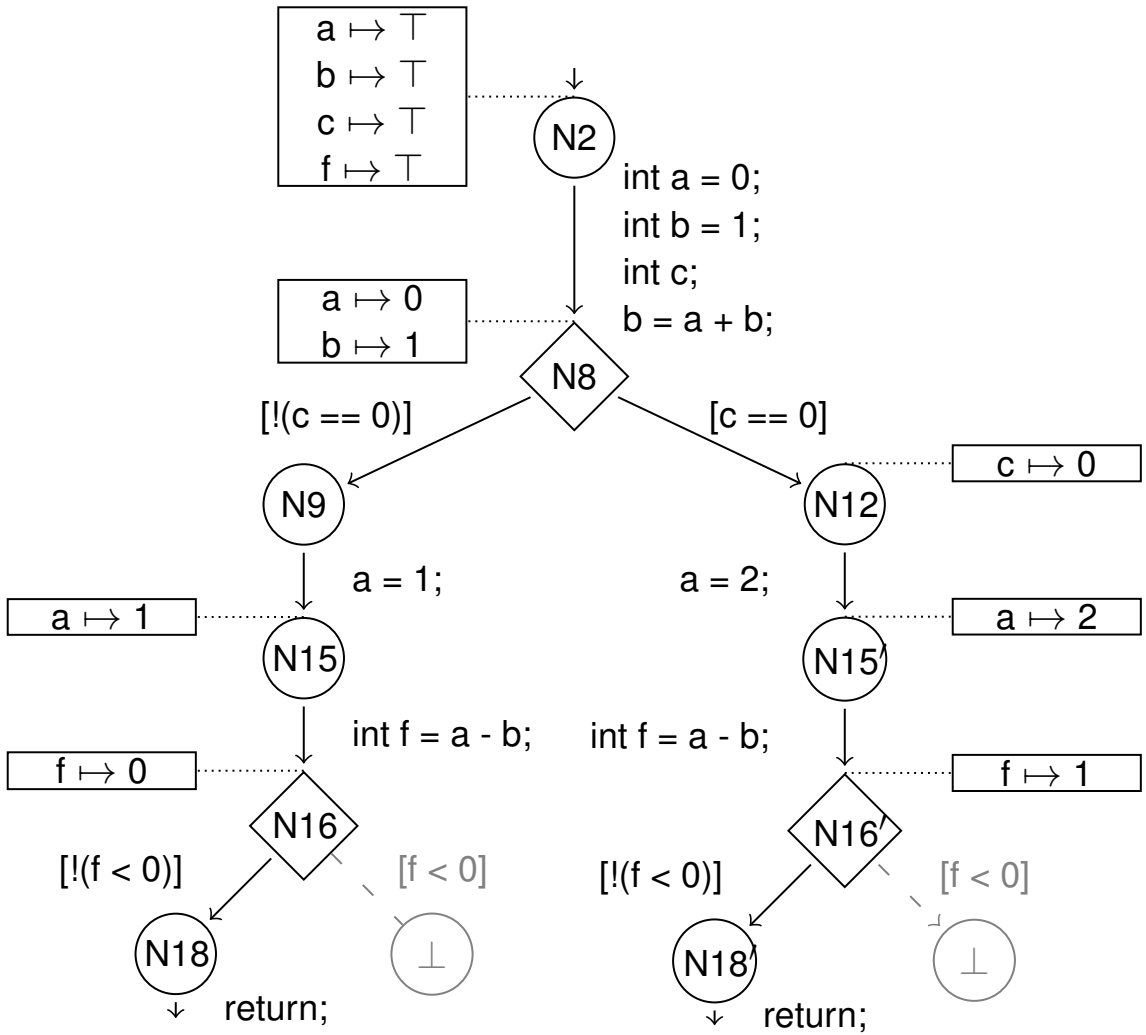
Example Program and its Control-Flow Automaton

```
 1 #include <assert.h>
 2 int main() {
 3   int a = 0;
 4   int b = 1;
 5   int c;
 6   b = a + b;
 7
 8   if (c) {
 9     a = 1;
10   }
11   else {
12     a = 2;
13   }
14
15   int f = a - b;
16   if (f < 0) {
17     assert(0);
18   }
19 }
```

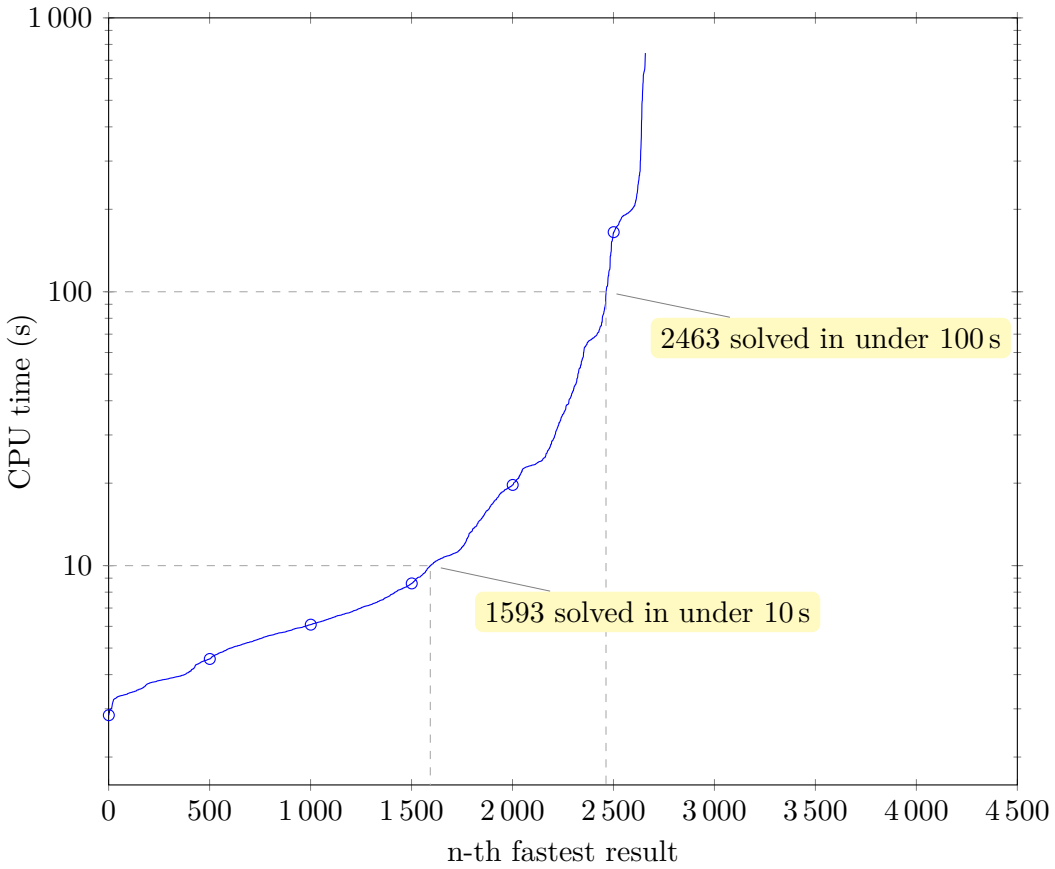


Example Program and its Abstract Reachability Graph

```
 1 #include <assert.h>
 2 int main() {
 3   int a = 0;
 4   int b = 1;
 5   int c;
 6   b = a + b;
 7
 8   if (c) {
 9     a = 1;
10   }
11   else {
12     a = 2;
13   }
14
15   int f = a - b;
16   if (f < 0) {
17     assert(0);
18   }
19 }
```

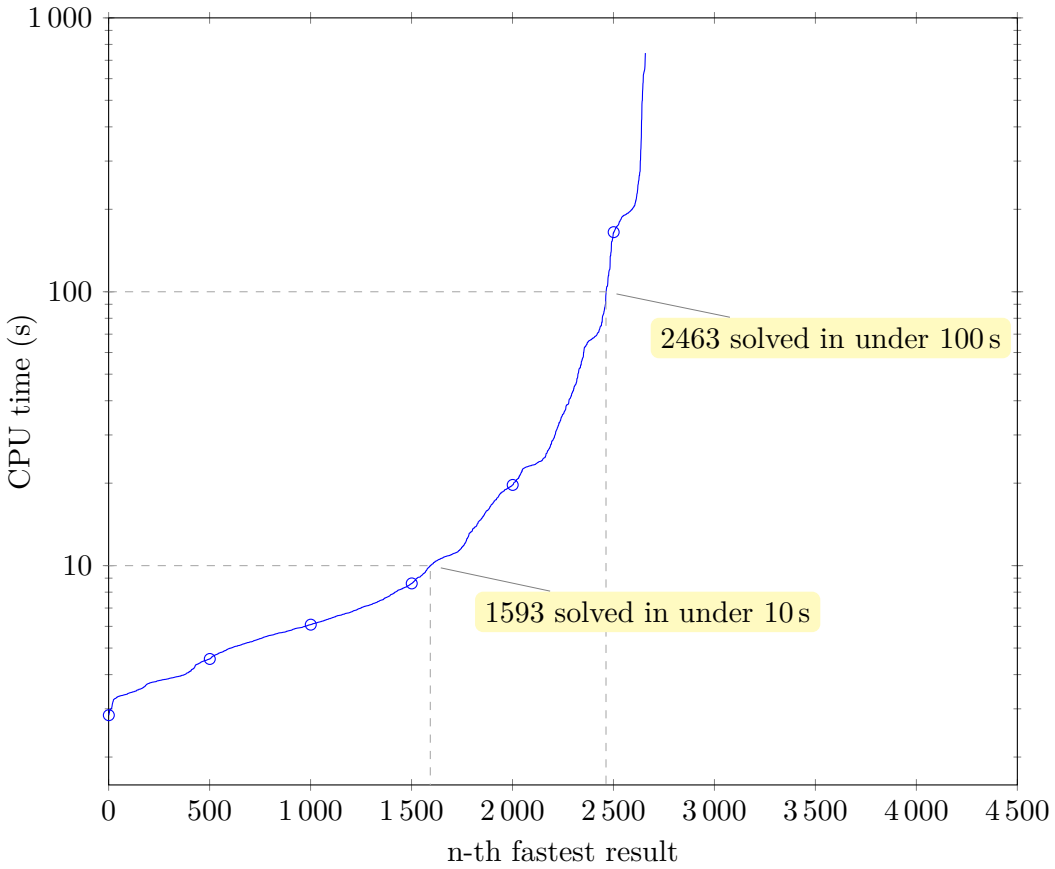


Value Analysis in Action



- Evaluated on over 4 200 verification tasks from SV-COMP'16
- Value Analysis solves almost two thirds
- Under SV-COMP'16 rules, complete evaluation takes 440 hours, i.e, over 18 days
- More than 90 % of the CPU time is wasted for unsolved verification tasks

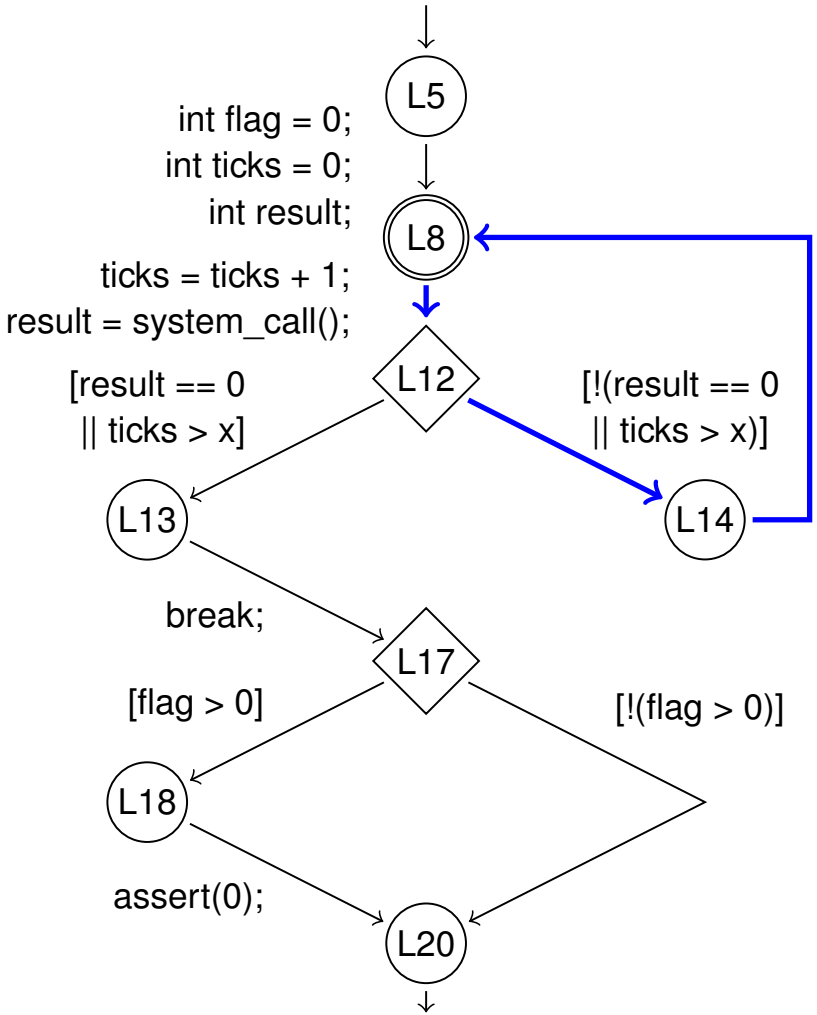
Value Analysis in Action



- Evaluated on over 4 200 verification tasks from SV-COMP'16
- Value Analysis solves almost two thirds
- Under SV-COMP'16 rules, complete evaluation takes 440 hours, i.e, over 18 days
- More than 90 % of the CPU time is wasted for unsolved verification tasks

State-Space Explosion

```
1  #include <assert.h>
2  extern int system_call();
3
4  int main(int x) {
5    int flag = 0, ticks = 0;
6    int result;
7
8    while(1) {
9      ticks  = ticks + 1;
10     result = system_call();
11
12     if(result == 0 || ticks > x) {
13       break;
14     }
15   }
16
17   if(flag > 0) {
18     assert(0);
19   }
20 }
```



State-Space Explosion

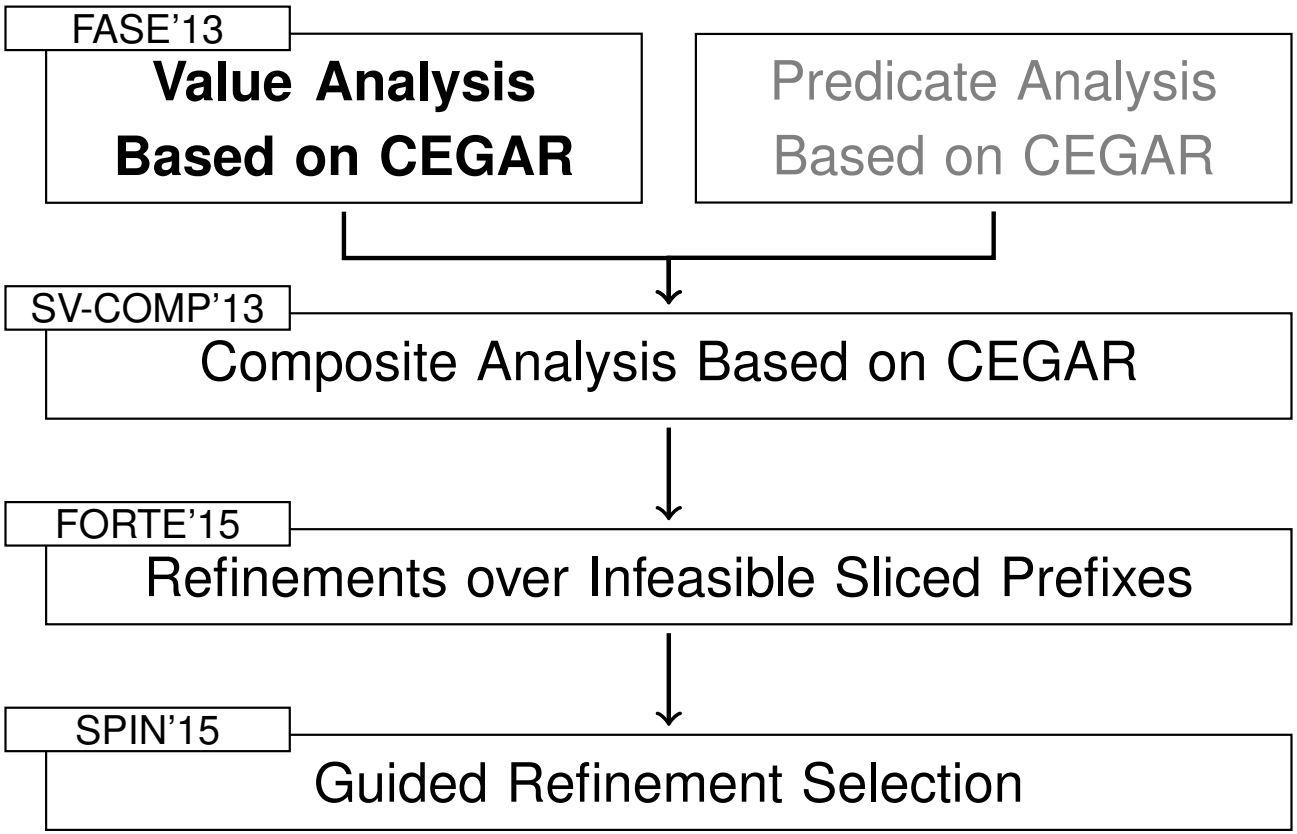
Problem

The plain Value Analysis suffers from state-space explosion

Proposition

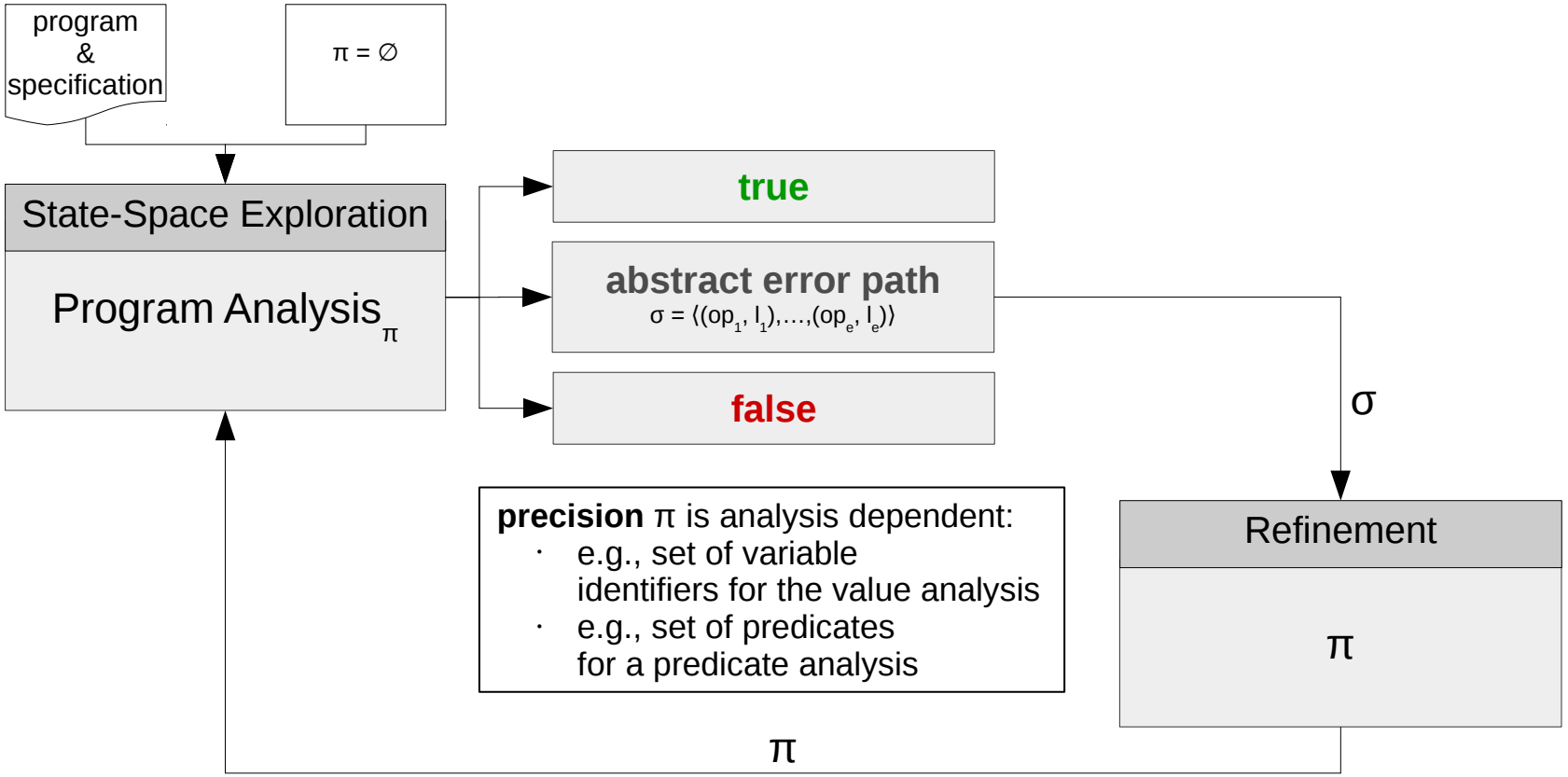
Abstraction techniques could reduce the size of the state space

Outline


FASE'13
Value Analysis Based on CEGAR
Predicate Analysis Based on CEGAR
SV-COMP'13
Composite Analysis Based on CEGAR
FORTE'15
Refinements over Infeasible Sliced Prefixes
SPIN'15
Guided Refinement Selection

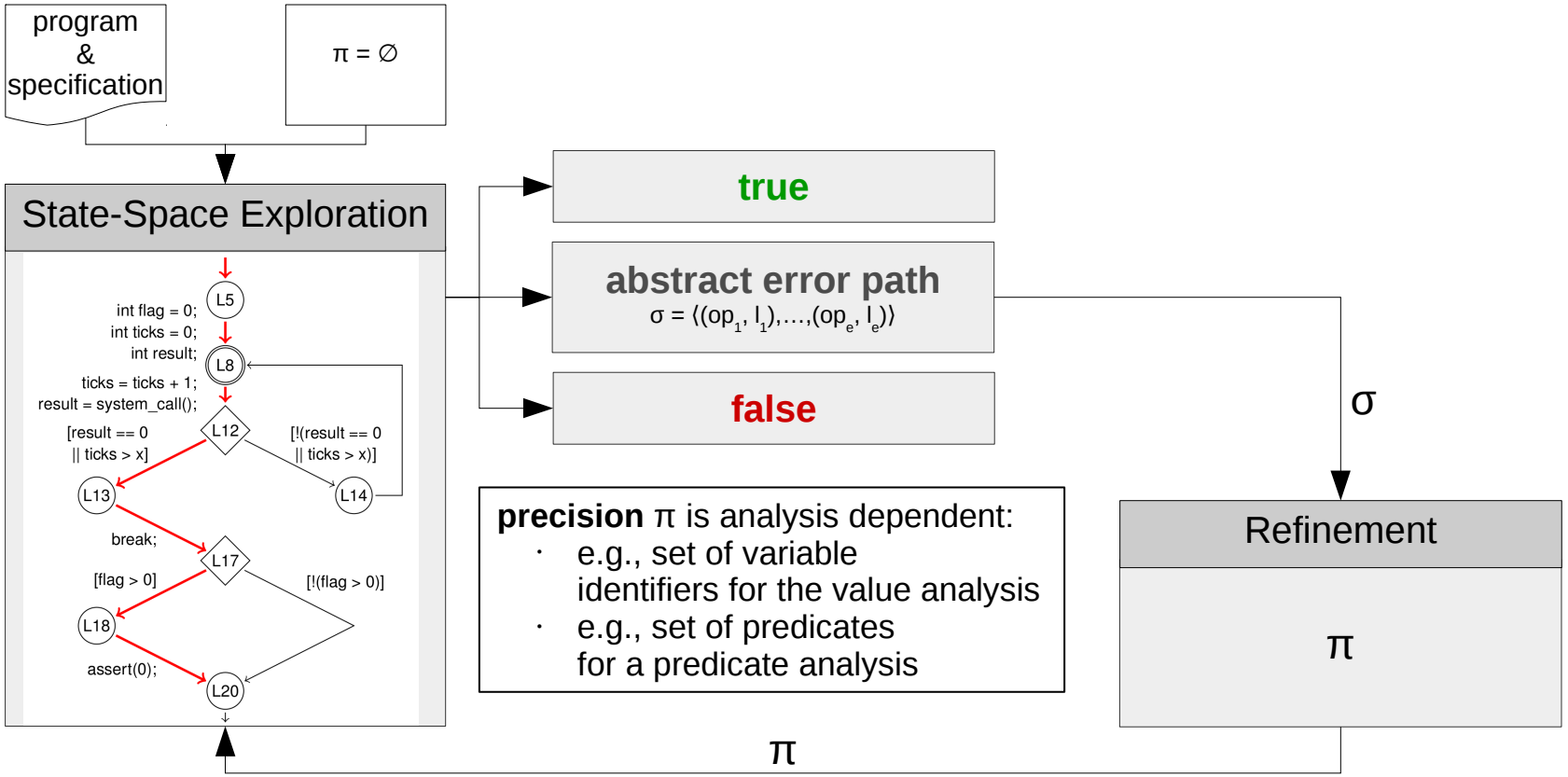
Counterexample-Guided Abstraction Refinement — CEGAR

CAV'00, Clarke, Grumberg, Jha, Lu, Veith



Counterexample-Guided Abstraction Refinement — CEGAR

CAV'00, Clarke, Grumberg, Jha, Lu, Veith



Value Interpolation — Concept

Definition

A constraint sequence γ is a sequence of program operations, formally, $\gamma = \langle op_1, \ldots, op_n \rangle$, where $op_1, \ldots, op_n$ are program operations

Examples

The sequence $\gamma_1 = \langle [\text{flag} = 0], [\text{ticks} = 0] \rangle$ is a constraint sequence, the constraint sequence $\gamma_2 = \langle [\text{flag} = 0], [\text{flag} > 0] \rangle$ is contradicting

Value Interpolation — Concept

FASE'13, Beyer, Löwe

inspired by Interpolation (JSL'57, Craig) and Abstractions from Proofs (POPL'04, Henzinger et al)

Theorem

For a pair of constraint sequences γ^- and γ^+, such that $\gamma^- \wedge \gamma^+$ is contradicting, an interpolant Γ is a constraint sequence that fulfills the following three requirements:

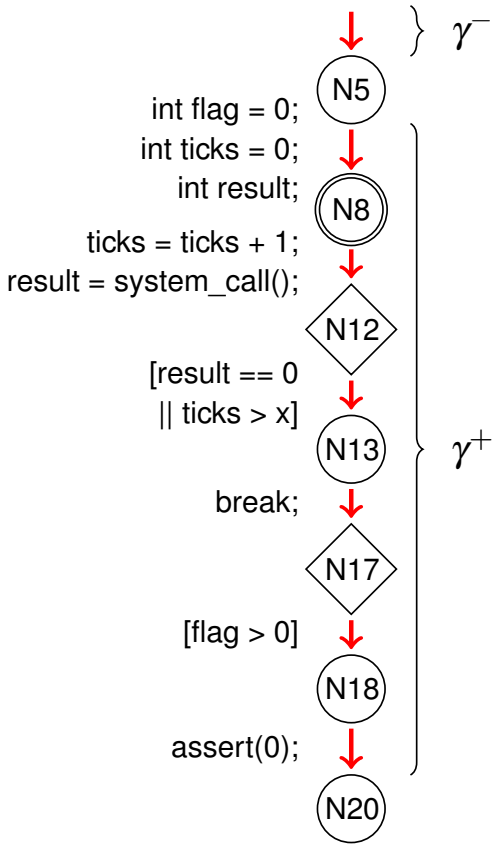
1. *the implication $\gamma^- \implies \Gamma$ holds,*
2. *the conjunction $\Gamma \wedge \gamma^+$ is contradicting, and*
3. *Γ contains in its constraints only program variables that occur in the constraints of both γ^- and γ^+*

Value Interpolation — Algorithm

Algorithm 1: Interpolate(γ^-, γ^+)

Input : two constraint sequences γ^- and γ^+, such that $\gamma^- \wedge \gamma^+$ is contradicting
Output : an interpolant Γ for γ^- and γ^+
Variables: an abstract variable assignment v

$v := \widehat{\mathrm{SP}}_{\gamma^-}(\top)$;
foreach $x \in \mathrm{def}(v)$ **do**
    if $\widehat{\mathrm{SP}}_{\gamma^+}(v_{|\mathrm{def}(v)\setminus\{x\}}) = \bot$ **then**
        // x is irrelevant and must not be in the interpolant
        $v := v_{|\mathrm{def}(v)\setminus\{x\}}$;
// start assembling the interpolating constraint sequence
$\Gamma := \langle\rangle$;
foreach $x \in \mathrm{def}(v)$ **do**
    // append an assume constraint for x
    $\Gamma := \Gamma \wedge \langle [x = v(x)] \rangle$;
return Γ

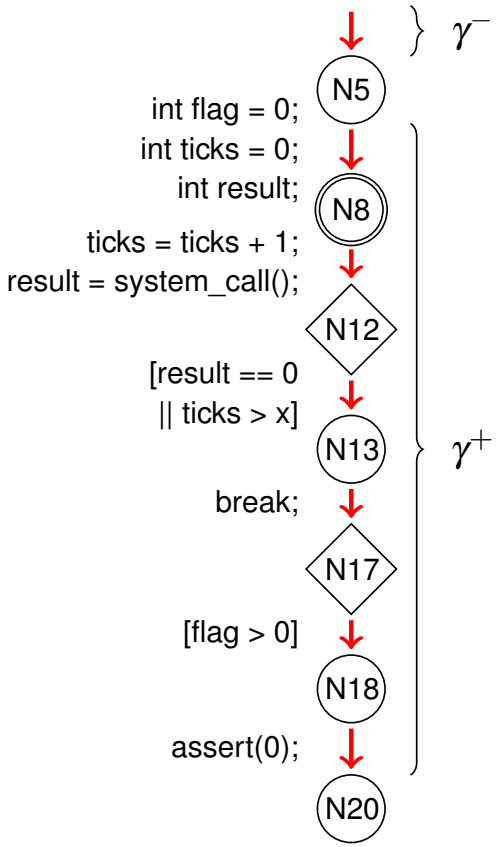


Value Interpolation — Algorithm

Algorithm 1: Interpolate(γ^-, γ^+)

Input : two constraint sequences γ^- and γ^+, such that $\gamma^- \wedge \gamma^+$ is contradicting
Output : an interpolant Γ for γ^- and γ^+
Variables: an abstract variable assignment v

$v := \widehat{\mathrm{SP}}_{\gamma^-}(\top)$; // $\rightarrow v = \top$
foreach $x \in \mathrm{def}(v)$ **do**
  if $\widehat{\mathrm{SP}}_{\gamma^+}(v_{|\mathrm{def}(v)\setminus\{x\}}) = \bot$ **then**
    // x is irrelevant and must not be in the interpolant
    $v := v_{|\mathrm{def}(v)\setminus\{x\}}$;
// start assembling the interpolating constraint sequence
$\Gamma := \langle\rangle$;
foreach $x \in \mathrm{def}(v)$ **do**
  // append an assume constraint for x
  $\Gamma := \Gamma \wedge \langle [x = v(x)] \rangle$;
return Γ



Value Interpolation — Algorithm

Algorithm 1: Interpolate(γ^-, γ^+)

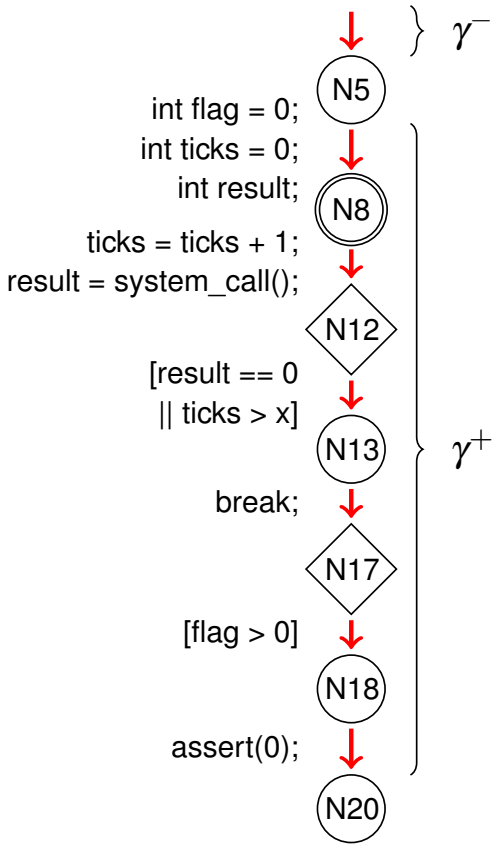
Input : two constraint sequences γ^- and γ^+, such that $\gamma^- \wedge \gamma^+$ is contradicting
Output : an interpolant Γ for γ^- and γ^+
Variables: an abstract variable assignment v

$v := \widehat{\mathrm{SP}}_{\gamma^-}(\top)$; // $\rightarrow v = \top$
foreach $x \in \mathrm{def}(v)$ **do**
  if $\widehat{\mathrm{SP}}_{\gamma^+}(v_{|\mathrm{def}(v)\setminus\{x\}}) = \bot$ **then**
    // x is irrelevant and must not be in the interpolant
    $v := v_{|\mathrm{def}(v)\setminus\{x\}}$;

// start assembling the interpolating constraint sequence
$\Gamma := \langle\rangle$;
foreach $x \in \mathrm{def}(v)$ **do**
  // append an assume constraint for x
  $\Gamma := \Gamma \wedge \langle [x = v(x)] \rangle$;

return Γ // $\rightarrow \Gamma = \langle\rangle$

$\Gamma_{\mathrm{N5}} = \langle\rangle$



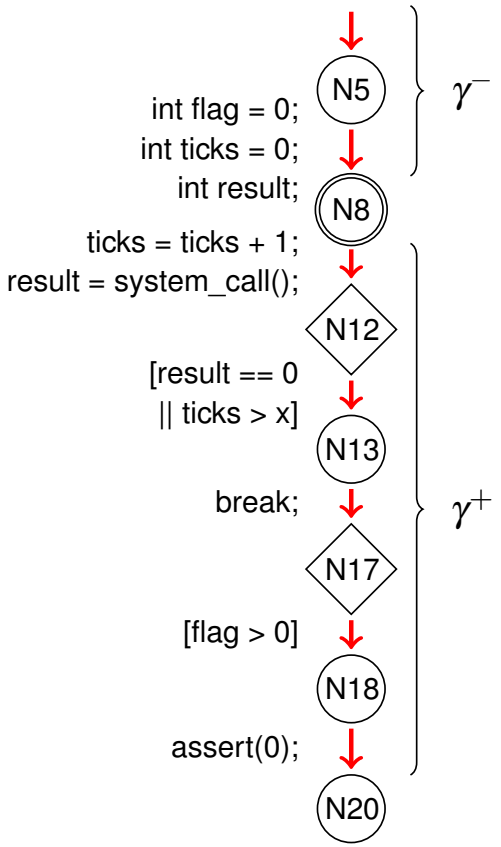
Value Interpolation — Algorithm

Algorithm 1: Interpolate(γ^-, γ^+)

Input : two constraint sequences γ^- and γ^+, such that $\gamma^- \wedge \gamma^+$ is contradicting
Output : an interpolant Γ for γ^- and γ^+
Variables: an abstract variable assignment v

$v := \widehat{\mathrm{SP}}_{\gamma^-}(\top)$; // $\rightarrow v = \{\mathrm{flag} \mapsto 0; \mathrm{ticks} \mapsto 0\}$
foreach $x \in \mathrm{def}(v)$ **do**
  if $\widehat{\mathrm{SP}}_{\gamma^+}(v_{|\mathrm{def}(v)\setminus\{x\}}) = \bot$ **then**
    // x is irrelevant and must not be in the interpolant
    $v := v_{|\mathrm{def}(v)\setminus\{x\}}$;
// start assembling the interpolating constraint sequence
$\Gamma := \langle\rangle$;
foreach $x \in \mathrm{def}(v)$ **do**
  // append an assume constraint for x
  $\Gamma := \Gamma \wedge \langle [x = v(x)] \rangle$;
return Γ

$\Gamma_{\mathrm{N5}} = \langle\rangle$



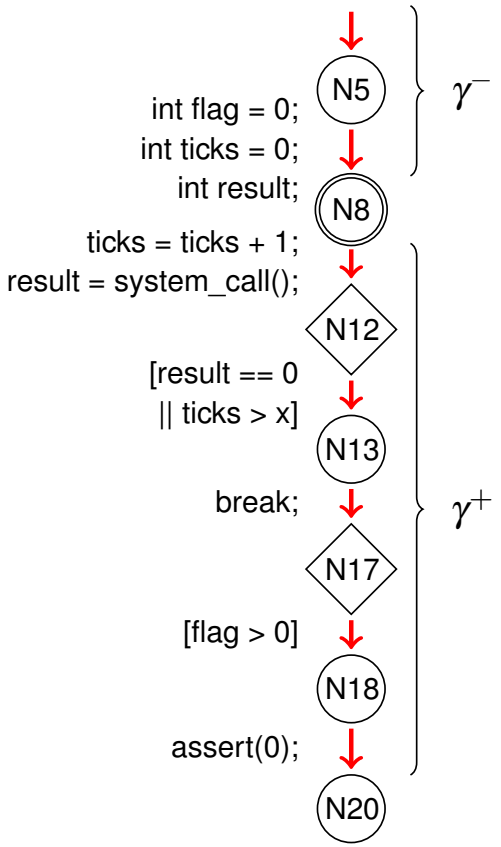
Value Interpolation — Algorithm

Algorithm 1: Interpolate(γ^-, γ^+)

Input : two constraint sequences γ^- and γ^+, such that $\gamma^- \wedge \gamma^+$ is contradicting
Output : an interpolant Γ for γ^- and γ^+
Variables: an abstract variable assignment v

$v := \widehat{\mathrm{SP}}_{\gamma^-}(\top)$; // $\rightarrow v = \{\mathrm{flag} \mapsto 0; \mathrm{ticks} \mapsto 0\}$
foreach $x \in \mathrm{def}(v)$ **do**
  if $\widehat{\mathrm{SP}}_{\gamma^+}(v_{|\mathrm{def}(v)\setminus\{x\}}) = \bot$ **then**
    // x is irrelevant and must not be in the interpolant
    $v := v_{|\mathrm{def}(v)\setminus\{x\}}$;
// start assembling the interpolating constraint sequence
$\Gamma := \langle\rangle$;
foreach $x \in \mathrm{def}(v)$ **do**
  // append an assume constraint for x
  $\Gamma := \Gamma \wedge \langle [x = v(x)] \rangle$;
return Γ

$\Gamma_{N5} = \langle\rangle$



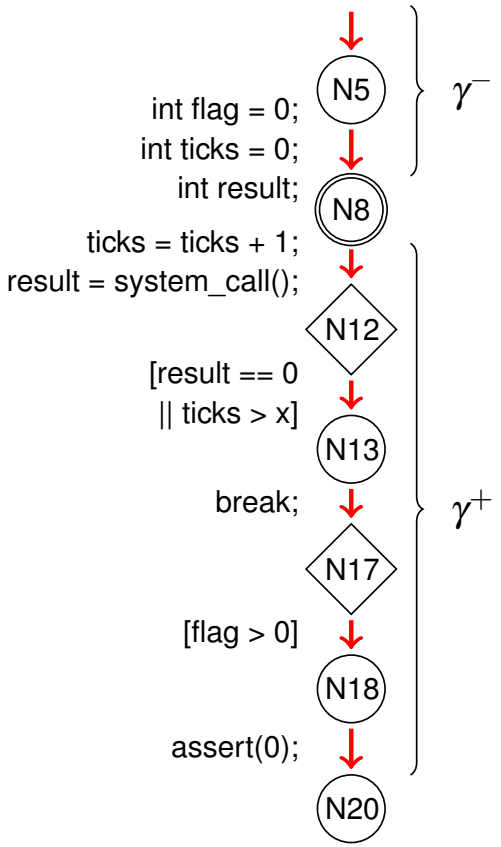
Value Interpolation — Algorithm

Algorithm 1: Interpolate(γ^-, γ^+)

Input : two constraint sequences γ^- and γ^+, such that $\gamma^- \wedge \gamma^+$ is contradicting
Output : an interpolant Γ for γ^- and γ^+
Variables: an abstract variable assignment v

$v := \widehat{\mathrm{SP}}_{\gamma^-}(\top)$; // $\rightarrow v = \{\mathrm{flag} \mapsto 0; \mathrm{ticks} \mapsto 0\}$
foreach $x \in \mathrm{def}(v)$ **do**
  if $\widehat{\mathrm{SP}}_{\gamma^+}(v_{|\mathrm{def}(v)\setminus\{x\}}) = \bot$ **then**
    // ticks is irrelevant and must not be in the interpolant
    $v := v_{|\mathrm{def}(v)\setminus\{x\}}$;
// start assembling the interpolating constraint sequence
$\Gamma := \langle\rangle$;
foreach $x \in \mathrm{def}(v)$ **do**
  // append an assume constraint for x
  $\Gamma := \Gamma \wedge \langle [x = v(x)] \rangle$;
return Γ

$\Gamma_{N5} = \langle\rangle$



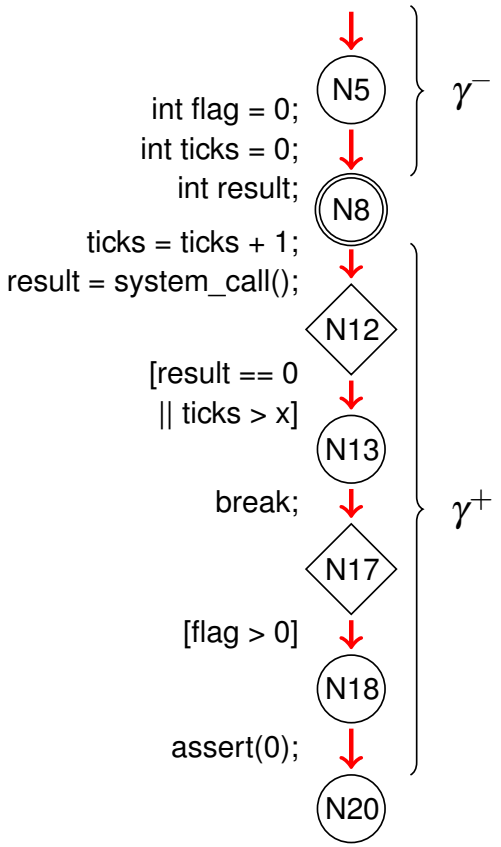
Value Interpolation — Algorithm

Algorithm 1: Interpolate(γ^-, γ^+)

Input : two constraint sequences γ^- and γ^+, such that $\gamma^- \wedge \gamma^+$ is contradicting
Output : an interpolant Γ for γ^- and γ^+
Variables: an abstract variable assignment v

$v := \widehat{\mathrm{SP}}_{\gamma^-}(\top)$; // $\rightarrow v = \{\mathrm{flag} \mapsto 0; \mathrm{ticks} \mapsto 0\}$
foreach $x \in \mathrm{def}(v)$ **do**
  if $\widehat{\mathrm{SP}}_{\gamma^+}(v_{|\mathrm{def}(v)\setminus\{x\}}) = \bot$ **then**
    // ticks is irrelevant and must not be in the interpolant
    $v := v_{|\mathrm{def}(v)\setminus\{x\}}$;
// start assembling the interpolating constraint sequence
$\Gamma := \langle\rangle$;
foreach $x \in \mathrm{def}(v)$ **do**
  // append an assume constraint for x
  $\Gamma := \Gamma \wedge \langle [x = v(x)] \rangle$;
return Γ

$\Gamma_{\mathrm{N5}} = \langle\rangle$



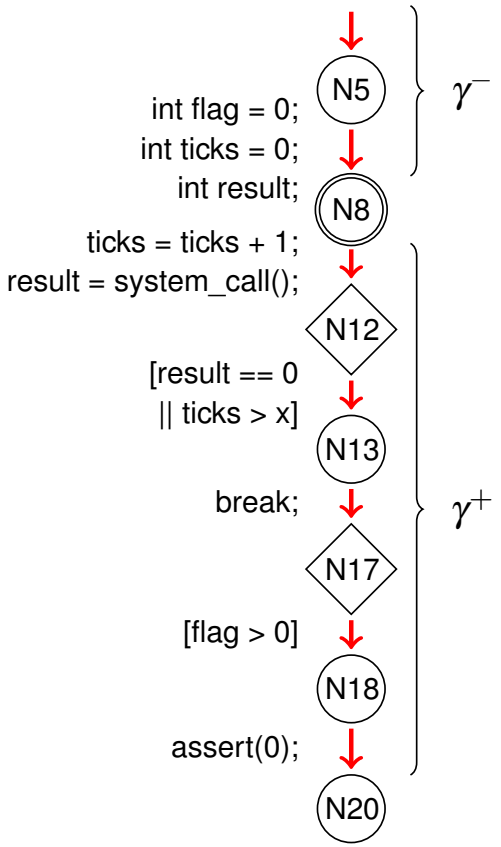
Value Interpolation — Algorithm

Algorithm 1: Interpolate(γ^-, γ^+)

Input : two constraint sequences γ^- and γ^+, such that $\gamma^- \wedge \gamma^+$ is contradicting
Output : an interpolant Γ for γ^- and γ^+
Variables: an abstract variable assignment v

$v := \widehat{\mathrm{SP}}_{\gamma^-}(\top)$; // $\rightarrow v = \{\mathrm{flag} \mapsto 0; \mathrm{ticks} \mapsto 0\}$
foreach $x \in \mathrm{def}(v)$ **do**
  if $\widehat{\mathrm{SP}}_{\gamma^+}(v_{|\mathrm{def}(v)\setminus\{x\}}) = \bot$ **then**
    // x is irrelevant and must not be in the interpolant
    $v := v_{|\mathrm{def}(v)\setminus\{x\}}$;
// start assembling the interpolating constraint sequence
$\Gamma := \langle\rangle$;
foreach $x \in \mathrm{def}(v)$ **do**
  // append an assume constraint for x
  $\Gamma := \Gamma \wedge \langle [x = v(x)] \rangle$;
return Γ

$\Gamma_{\mathrm{N5}} = \langle\rangle$



Value Interpolation — Algorithm

Algorithm 1: Interpolate(γ^-, γ^+)

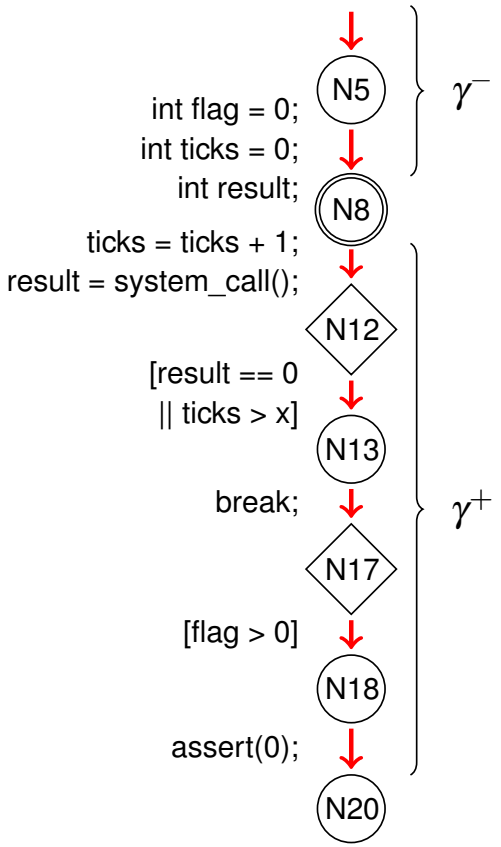
Input : two constraint sequences γ^- and γ^+, such that $\gamma^- \wedge \gamma^+$ is contradicting
Output : an interpolant Γ for γ^- and γ^+
Variables: an abstract variable assignment v

$v := \widehat{\mathrm{SP}}_{\gamma^-}(\top)$; // $\rightarrow v = \{\mathrm{flag} \mapsto 0; \mathrm{ticks} \mapsto 0\}$
foreach $x \in \mathrm{def}(v)$ **do**
  if $\widehat{\mathrm{SP}}_{\gamma^+}(v_{|\mathrm{def}(v)\setminus\{x\}}) = \bot$ **then**
    // x is irrelevant and must not be in the interpolant
    $v := v_{|\mathrm{def}(v)\setminus\{x\}}$;

// start assembling the interpolating constraint sequence
$\Gamma := \langle\rangle$;
foreach $x \in \mathrm{def}(v)$ **do**
  // append an assume constraint for x
  $\Gamma := \Gamma \wedge \langle [x = v(x)] \rangle$;

return Γ

$\Gamma_{\mathrm{N5}} = \langle\rangle$



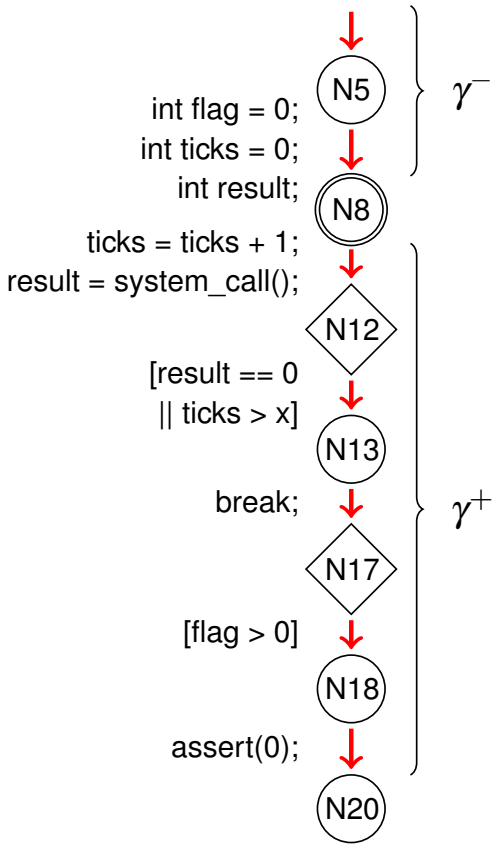
Value Interpolation — Algorithm

Algorithm 1: Interpolate(γ^-, γ^+)

Input : two constraint sequences γ^- and γ^+, such that $\gamma^- \wedge \gamma^+$ is contradicting
Output : an interpolant Γ for γ^- and γ^+
Variables: an abstract variable assignment v

$v := \widehat{\mathrm{SP}}_{\gamma^-}(\top)$; // $\rightarrow v = \{\mathrm{flag} \mapsto 0; \mathrm{ticks} \mapsto 0\}$
foreach $x \in \mathrm{def}(v)$ **do**
  if $\widehat{\mathrm{SP}}_{\gamma^+}(v_{|\mathrm{def}(v)\setminus\{x\}}) = \bot$ **then**
    // x is irrelevant and must not be in the interpolant
    $v := v_{|\mathrm{def}(v)\setminus\{x\}}$;
// start assembling the interpolating constraint sequence
$\Gamma := \langle\rangle$;
foreach $x \in \mathrm{def}(v)$ **do**
  // append an assume constraint for flag
  $\Gamma := \Gamma \wedge \langle [x = v(x)] \rangle$;
return Γ

$\Gamma_{\mathrm{N5}} = \langle\rangle$



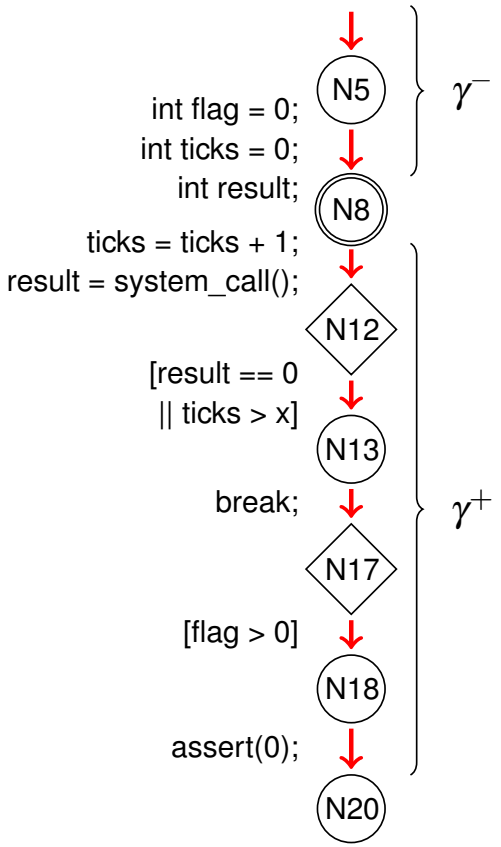
Value Interpolation — Algorithm

Algorithm 1: Interpolate(γ^-, γ^+)

Input : two constraint sequences γ^- and γ^+, such that $\gamma^- \wedge \gamma^+$ is contradicting
Output : an interpolant Γ for γ^- and γ^+
Variables: an abstract variable assignment v

$v := \widehat{\mathrm{SP}}_{\gamma^-}(\top)$; // $\rightarrow v = \{\mathrm{flag} \mapsto 0; \mathrm{ticks} \mapsto 0\}$
foreach $x \in \mathrm{def}(v)$ **do**
  if $\widehat{\mathrm{SP}}_{\gamma^+}(v_{|\mathrm{def}(v)\setminus\{x\}}) = \bot$ **then**
    // x is irrelevant and must not be in the interpolant
    $v := v_{|\mathrm{def}(v)\setminus\{x\}}$;
// start assembling the interpolating constraint sequence
$\Gamma := \langle\rangle$;
foreach $x \in \mathrm{def}(v)$ **do**
  // append an assume constraint for x
  $\Gamma := \Gamma \wedge \langle[x = v(x)]\rangle$;
return Γ // $\rightarrow \Gamma = \langle[\mathrm{flag} = 0]\rangle$

$\Gamma_{\mathrm{N5}} = \langle\rangle$, $\Gamma_{\mathrm{N8}} = \langle[\mathrm{flag} = 0]\rangle$

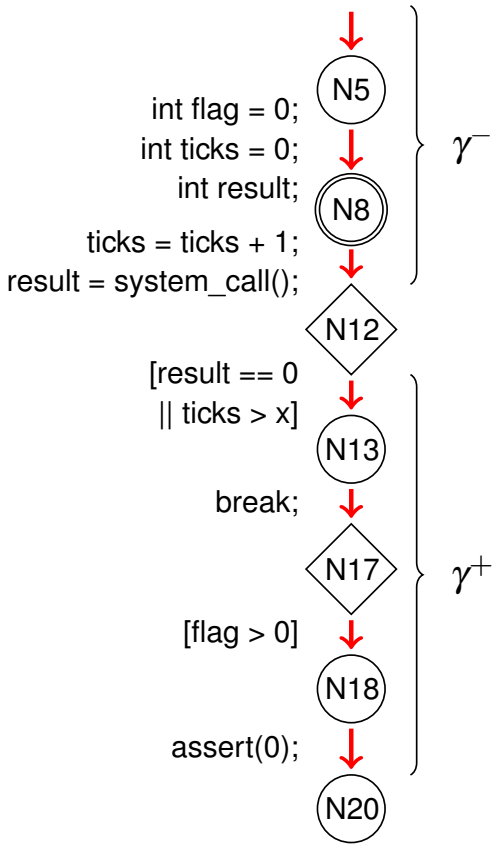


Value Interpolation — Algorithm

Algorithm 1: Interpolate(γ^-, γ^+)

Input : two constraint sequences γ^- and γ^+, such that $\gamma^- \wedge \gamma^+$ is contradicting
Output : an interpolant Γ for γ^- and γ^+
Variables: an abstract variable assignment v

$v := \widehat{\mathrm{SP}}_{\gamma^-}(\top)$; // $\rightarrow v = \{\mathrm{flag} \mapsto 0; \mathrm{ticks} \mapsto 1\}$
foreach $x \in \mathrm{def}(v)$ **do**
  if $\widehat{\mathrm{SP}}_{\gamma^+}(v_{|\mathrm{def}(v)\setminus\{x\}}) = \bot$ **then**
    // x is irrelevant and must not be in the interpolant
    $v := v_{|\mathrm{def}(v)\setminus\{x\}}$;
// start assembling the interpolating constraint sequence
$\Gamma := \langle\rangle$;
foreach $x \in \mathrm{def}(v)$ **do**
  // append an assume constraint for x
  $\Gamma := \Gamma \wedge \langle[x = v(x)]\rangle$;
return Γ



$\Gamma_{N5} = \langle\rangle$, $\Gamma_{N8} = \langle[\mathrm{flag} = 0]\rangle$

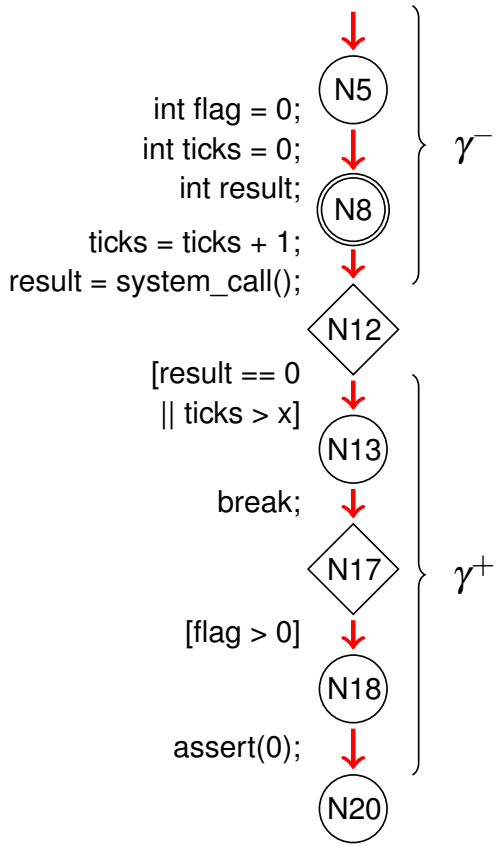
Value Interpolation — Algorithm

Algorithm 1: Interpolate(γ^-, γ^+)

Input : two constraint sequences γ^- and γ^+, such that $\gamma^- \wedge \gamma^+$ is contradicting
Output : an interpolant Γ for γ^- and γ^+
Variables: an abstract variable assignment v

$v := \widehat{\mathrm{SP}}_{\gamma^-}(\top)$; // $\rightarrow v = \{\mathrm{flag} \mapsto 0; \mathrm{ticks} \mapsto 1\}$
foreach $x \in \mathrm{def}(v)$ **do**
  if $\widehat{\mathrm{SP}}_{\gamma^+}(v_{|\mathrm{def}(v)\setminus\{x\}}) = \bot$ **then**
    // x is irrelevant and must not be in the interpolant
    $v := v_{|\mathrm{def}(v)\setminus\{x\}}$;
// start assembling the interpolating constraint sequence
$\Gamma := \langle\rangle$;
foreach $x \in \mathrm{def}(v)$ **do**
  // append an assume constraint for x
  $\Gamma := \Gamma \wedge \langle [x = v(x)] \rangle$;
return Γ

$\Gamma_{N5} = \langle\rangle$, $\Gamma_{N8} = \langle [\mathrm{flag} = 0] \rangle$



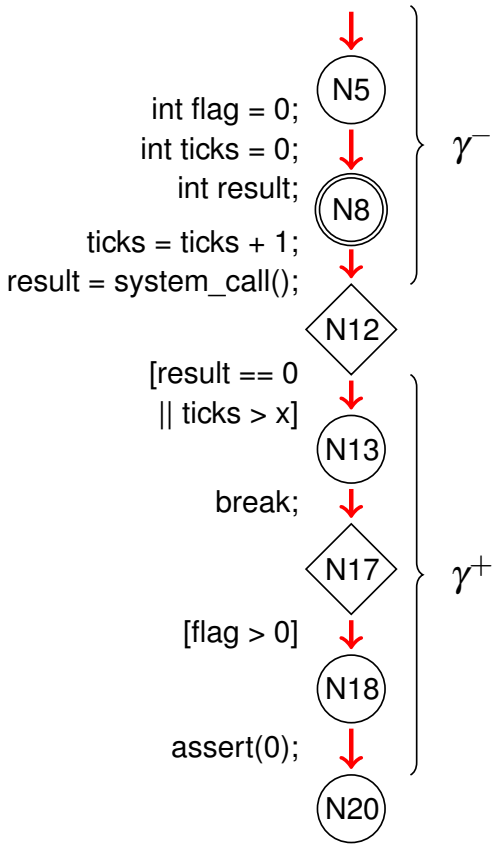
Value Interpolation — Algorithm

Algorithm 1: Interpolate(γ^-, γ^+)

Input : two constraint sequences γ^- and γ^+, such that $\gamma^- \wedge \gamma^+$ is contradicting
Output : an interpolant Γ for γ^- and γ^+
Variables: an abstract variable assignment v

$v := \widehat{\mathrm{SP}}_{\gamma^-}(\top)$; // $\rightarrow v = \{\mathrm{flag} \mapsto 0; \mathrm{ticks} \mapsto 1\}$
foreach $x \in \mathrm{def}(v)$ **do**
  if $\widehat{\mathrm{SP}}_{\gamma^+}(v_{|\mathrm{def}(v)\setminus\{x\}}) = \bot$ **then**
    // ticks is irrelevant and must not be in the interpolant
    $v := v_{|\mathrm{def}(v)\setminus\{x\}}$;
// start assembling the interpolating constraint sequence
$\Gamma := \langle\rangle$;
foreach $x \in \mathrm{def}(v)$ **do**
  // append an assume constraint for x
  $\Gamma := \Gamma \wedge \langle [x = v(x)] \rangle$;
return Γ

$\Gamma_{N5} = \langle\rangle$, $\Gamma_{N8} = \langle [\mathrm{flag} = 0] \rangle$



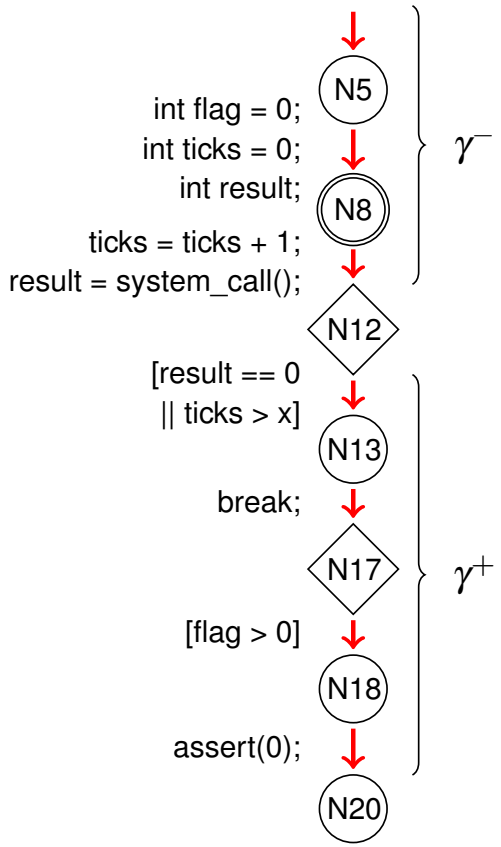
Value Interpolation — Algorithm

Algorithm 1: Interpolate(γ^-, γ^+)

Input : two constraint sequences γ^- and γ^+, such that $\gamma^- \wedge \gamma^+$ is contradicting
Output : an interpolant Γ for γ^- and γ^+
Variables: an abstract variable assignment v

$v := \widehat{\mathrm{SP}}_{\gamma^-}(\top)$; // $\rightarrow v = \{\mathrm{flag} \mapsto 0; \mathrm{ticks} \mapsto 1\}$
foreach $x \in \mathrm{def}(v)$ **do**
  if $\widehat{\mathrm{SP}}_{\gamma^+}(v_{|\mathrm{def}(v)\setminus\{x\}}) = \bot$ **then**
    // ticks is irrelevant and must not be in the interpolant
    $v := v_{|\mathrm{def}(v)\setminus\{x\}}$;
// start assembling the interpolating constraint sequence
$\Gamma := \langle\rangle$;
foreach $x \in \mathrm{def}(v)$ **do**
  // append an assume constraint for x
  $\Gamma := \Gamma \wedge \langle [x = v(x)] \rangle$;
return Γ

$\Gamma_{N5} = \langle\rangle$, $\Gamma_{N8} = \langle [\mathrm{flag} = 0] \rangle$



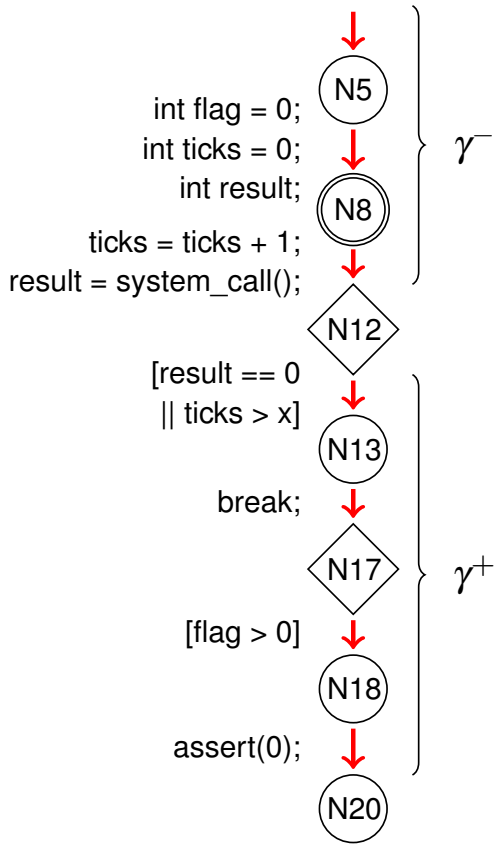
Value Interpolation — Algorithm

Algorithm 1: Interpolate(γ^-, γ^+)

Input : two constraint sequences γ^- and γ^+, such that $\gamma^- \wedge \gamma^+$ is contradicting
Output : an interpolant Γ for γ^- and γ^+
Variables: an abstract variable assignment v

$v := \widehat{\mathrm{SP}}_{\gamma^-}(\top)$; // $\rightarrow v = \{\mathrm{flag} \mapsto 0; \mathrm{ticks} \mapsto 1\}$
foreach $x \in \mathrm{def}(v)$ **do**
  if $\widehat{\mathrm{SP}}_{\gamma^+}(v_{|\mathrm{def}(v)\setminus\{x\}}) = \bot$ **then**
    // x is irrelevant and must not be in the interpolant
    $v := v_{|\mathrm{def}(v)\setminus\{x\}}$;
// start assembling the interpolating constraint sequence
$\Gamma := \langle\rangle$;
foreach $x \in \mathrm{def}(v)$ **do**
  // append an assume constraint for x
  $\Gamma := \Gamma \wedge \langle[x = v(x)]\rangle$;
return Γ

$\Gamma_{N5} = \langle\rangle$, $\Gamma_{N8} = \langle[\mathrm{flag} = 0]\rangle$



Value Interpolation — Algorithm

Algorithm 1: Interpolate(γ^-, γ^+)

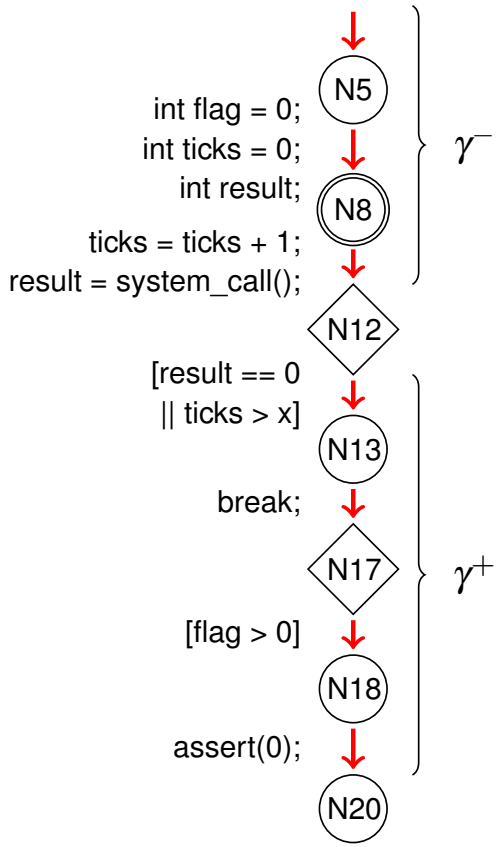
Input : two constraint sequences γ^- and γ^+, such that $\gamma^- \wedge \gamma^+$ is contradicting
Output : an interpolant Γ for γ^- and γ^+
Variables: an abstract variable assignment v

$v := \widehat{\mathrm{SP}}_{\gamma^-}(\top)$; // $\rightarrow v = \{\mathrm{flag} \mapsto 0; \mathrm{ticks} \mapsto 1\}$
foreach $x \in \mathrm{def}(v)$ **do**
  if $\widehat{\mathrm{SP}}_{\gamma^+}(v_{|\mathrm{def}(v)\setminus\{x\}}) = \bot$ **then**
    // x is irrelevant and must not be in the interpolant
    $v := v_{|\mathrm{def}(v)\setminus\{x\}}$;

// start assembling the interpolating constraint sequence
$\Gamma := \langle\rangle$;
foreach $x \in \mathrm{def}(v)$ **do**
  // append an assume constraint for x
  $\Gamma := \Gamma \wedge \langle [x = v(x)] \rangle$;

return Γ

$\Gamma_{N5} = \langle\rangle$, $\Gamma_{N8} = \langle [\mathrm{flag} = 0] \rangle$



Value Interpolation — Algorithm

Algorithm 1: Interpolate(γ^-, γ^+)

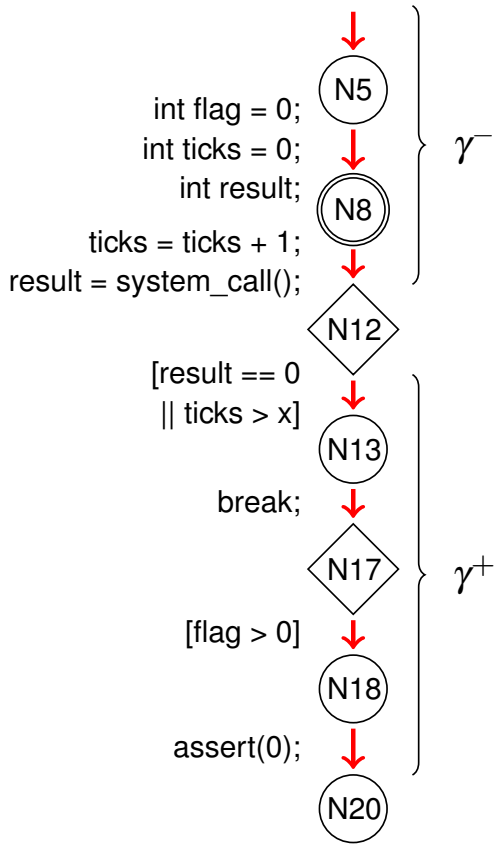
Input : two constraint sequences γ^- and γ^+, such that $\gamma^- \wedge \gamma^+$ is contradicting
Output : an interpolant Γ for γ^- and γ^+
Variables: an abstract variable assignment v

$v := \widehat{\mathrm{SP}}_{\gamma^-}(\top)$; // $\rightarrow v = \{\mathrm{flag} \mapsto 0; \mathrm{ticks} \mapsto 1\}$
foreach $x \in \mathrm{def}(v)$ **do**
  if $\widehat{\mathrm{SP}}_{\gamma^+}(v_{|\mathrm{def}(v)\setminus\{x\}}) = \bot$ **then**
    // x is irrelevant and must not be in the interpolant
    $v := v_{|\mathrm{def}(v)\setminus\{x\}}$;

// start assembling the interpolating constraint sequence
$\Gamma := \langle\rangle$;
foreach $x \in \mathrm{def}(v)$ **do**
  // append an assume constraint for flag
  $\Gamma := \Gamma \wedge \langle [x = v(x)] \rangle$;

return Γ

$\Gamma_{N5} = \langle\rangle$, $\Gamma_{N8} = \langle [\mathrm{flag} = 0] \rangle$



Value Interpolation — Algorithm

Algorithm 1: Interpolate(γ^-, γ^+)

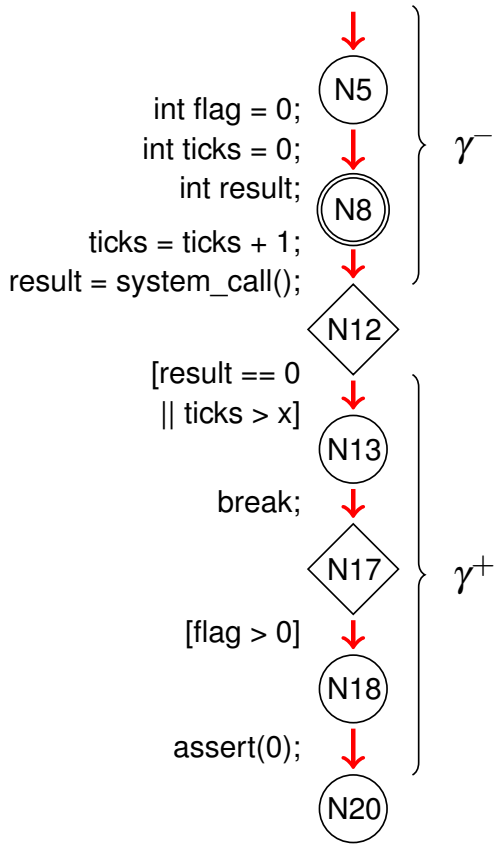
Input : two constraint sequences γ^- and γ^+, such that $\gamma^- \wedge \gamma^+$ is contradicting
Output : an interpolant Γ for γ^- and γ^+
Variables: an abstract variable assignment v

$v := \widehat{\mathrm{SP}}_{\gamma^-}(\top)$; // $\rightarrow v = \{\mathrm{flag} \mapsto 0; \mathrm{ticks} \mapsto 1\}$
foreach $x \in \mathrm{def}(v)$ **do**
  if $\widehat{\mathrm{SP}}_{\gamma^+}(v_{|\mathrm{def}(v)\setminus\{x\}}) = \bot$ **then**
    // x is irrelevant and must not be in the interpolant
    $v := v_{|\mathrm{def}(v)\setminus\{x\}}$;

// start assembling the interpolating constraint sequence
$\Gamma := \langle\rangle$;
foreach $x \in \mathrm{def}(v)$ **do**
  // append an assume constraint for x
  $\Gamma := \Gamma \wedge \langle[x = v(x)]\rangle$;

return Γ // $\rightarrow \Gamma = \langle[\mathrm{flag} = 0]\rangle$

$\Gamma_{N5} = \langle\rangle$, $\Gamma_{N8} = \langle[\mathrm{flag} = 0]\rangle$, $\Gamma_{N12} = \langle[\mathrm{flag} = 0]\rangle$



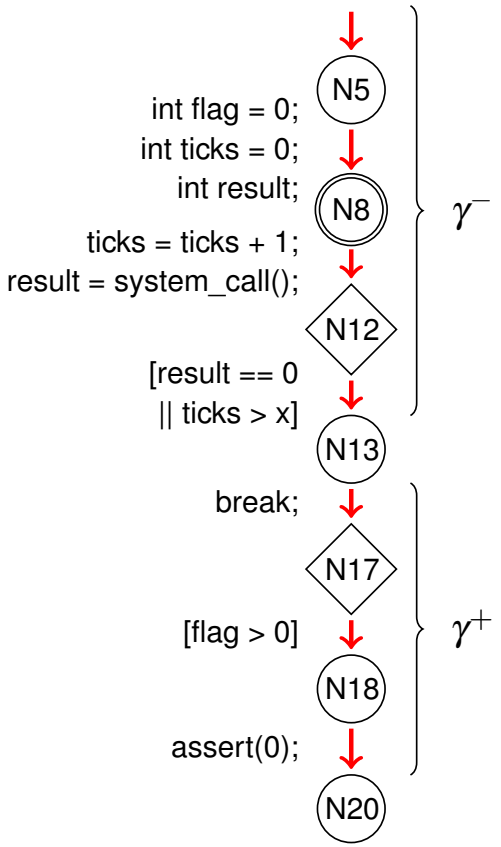
Value Interpolation — Algorithm

Algorithm 1: Interpolate(γ^-, γ^+)

Input : two constraint sequences γ^- and γ^+, such that $\gamma^- \wedge \gamma^+$ is contradicting
Output : an interpolant Γ for γ^- and γ^+
Variables: an abstract variable assignment v

$v := \widehat{\mathrm{SP}}_{\gamma^-}(\top)$;
foreach $x \in \mathrm{def}(v)$ **do**
  if $\widehat{\mathrm{SP}}_{\gamma^+}(v_{|\mathrm{def}(v)\setminus\{x\}}) = \bot$ **then**
    // x is irrelevant and must not be in the interpolant
    $v := v_{|\mathrm{def}(v)\setminus\{x\}}$;
// start assembling the interpolating constraint sequence
$\Gamma := \langle\rangle$;
foreach $x \in \mathrm{def}(v)$ **do**
  // append an assume constraint for x
  $\Gamma := \Gamma \wedge \langle [x = v(x)] \rangle$;
return Γ

$\Gamma_{N5} = \langle\rangle$, $\Gamma_{N8} = \langle [\mathrm{flag} = 0] \rangle$, $\Gamma_{N12} = \langle [\mathrm{flag} = 0] \rangle$, $\Gamma_{N13} = \langle [\mathrm{flag} = 0] \rangle$



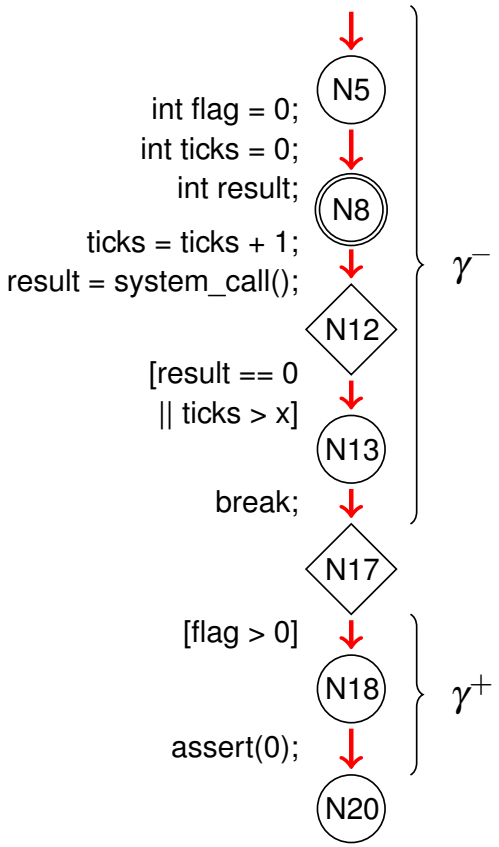
Value Interpolation — Algorithm

Algorithm 1: Interpolate(γ^-, γ^+)

Input : two constraint sequences γ^- and γ^+, such that $\gamma^- \wedge \gamma^+$ is contradicting
Output : an interpolant Γ for γ^- and γ^+
Variables: an abstract variable assignment v

$v := \widehat{\mathrm{SP}}_{\gamma^-}(\top)$;
foreach $x \in \mathrm{def}(v)$ **do**
  if $\widehat{\mathrm{SP}}_{\gamma^+}(v_{|\mathrm{def}(v)\setminus\{x\}}) = \bot$ **then**
    // x is irrelevant and must not be in the interpolant
    $v := v_{|\mathrm{def}(v)\setminus\{x\}}$;
// start assembling the interpolating constraint sequence
$\Gamma := \langle\rangle$;
foreach $x \in \mathrm{def}(v)$ **do**
  // append an assume constraint for x
  $\Gamma := \Gamma \wedge \langle [x = v(x)] \rangle$;
return Γ

$\Gamma_{N5} = \langle\rangle$, $\Gamma_{N8} = \langle [\mathrm{flag} = 0] \rangle$, $\Gamma_{N12} = \langle [\mathrm{flag} = 0] \rangle$, $\Gamma_{N13} = \langle [\mathrm{flag} = 0] \rangle$, $\Gamma_{N17} = \langle [\mathrm{flag} = 0] \rangle$



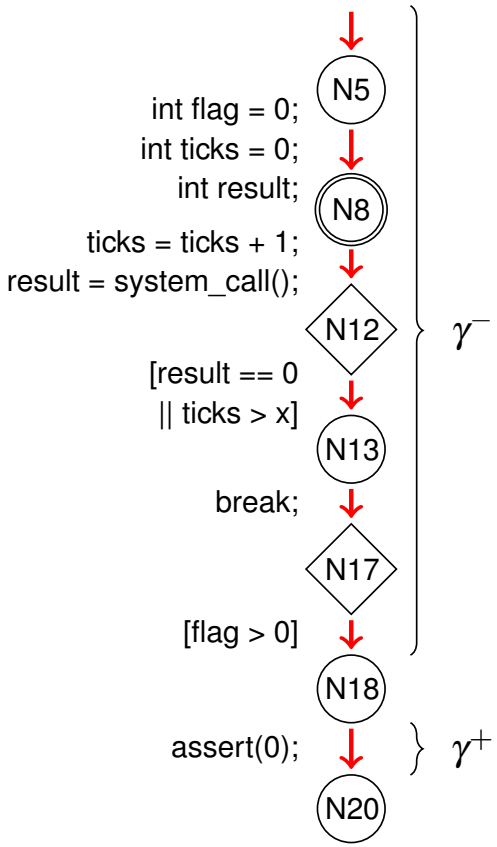
Value Interpolation — Algorithm

Algorithm 1: Interpolate(γ^-, γ^+)

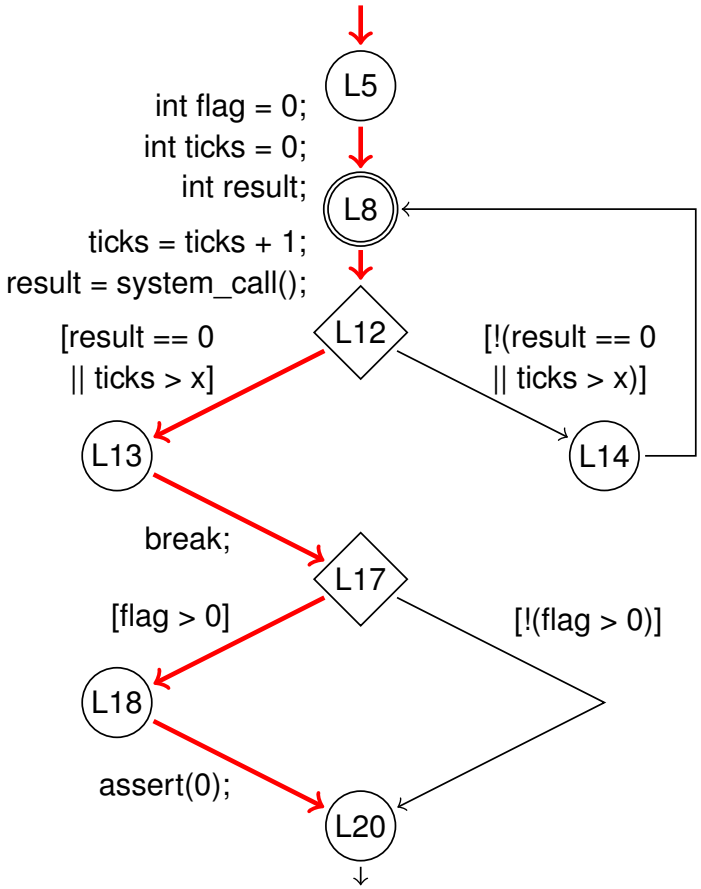
Input : two constraint sequences γ^- and γ^+, such that $\gamma^- \wedge \gamma^+$ is contradicting
Output : an interpolant Γ for γ^- and γ^+
Variables: an abstract variable assignment v

$v := \widehat{\mathrm{SP}}_{\gamma^-}(\top)$;
foreach $x \in \mathrm{def}(v)$ **do**
  if $\widehat{\mathrm{SP}}_{\gamma^+}(v_{|\mathrm{def}(v)\setminus\{x\}}) = \bot$ **then**
    // x is irrelevant and must not be in the interpolant
    $v := v_{|\mathrm{def}(v)\setminus\{x\}}$;
// start assembling the interpolating constraint sequence
$\Gamma := \langle\rangle$;
foreach $x \in \mathrm{def}(v)$ **do**
  // append an assume constraint for x
  $\Gamma := \Gamma \wedge \langle [x = v(x)] \rangle$;
return Γ

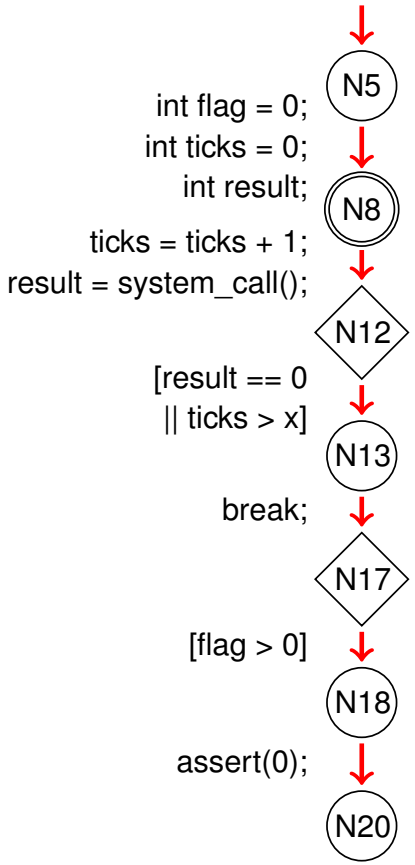
$\Gamma_{N5} = \langle\rangle$, $\Gamma_{N8} = \langle [\mathrm{flag} = 0] \rangle$, $\Gamma_{N12} = \langle [\mathrm{flag} = 0] \rangle$, $\Gamma_{N13} = \langle [\mathrm{flag} = 0] \rangle$, $\Gamma_{N17} = \langle [\mathrm{flag} = 0] \rangle$, $\Gamma_{N18} = \langle\rangle$, $\Gamma_{N20} = \langle\rangle$



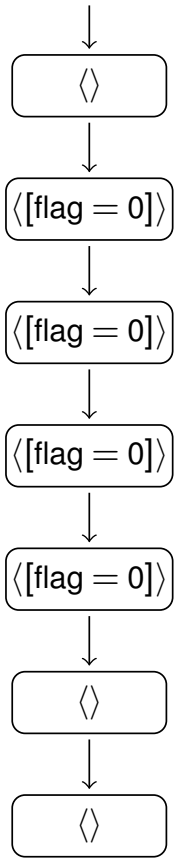
Value Refinement Completed



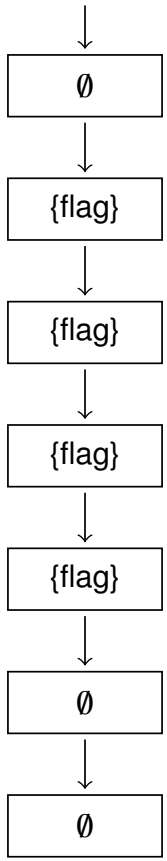
verification task



error path

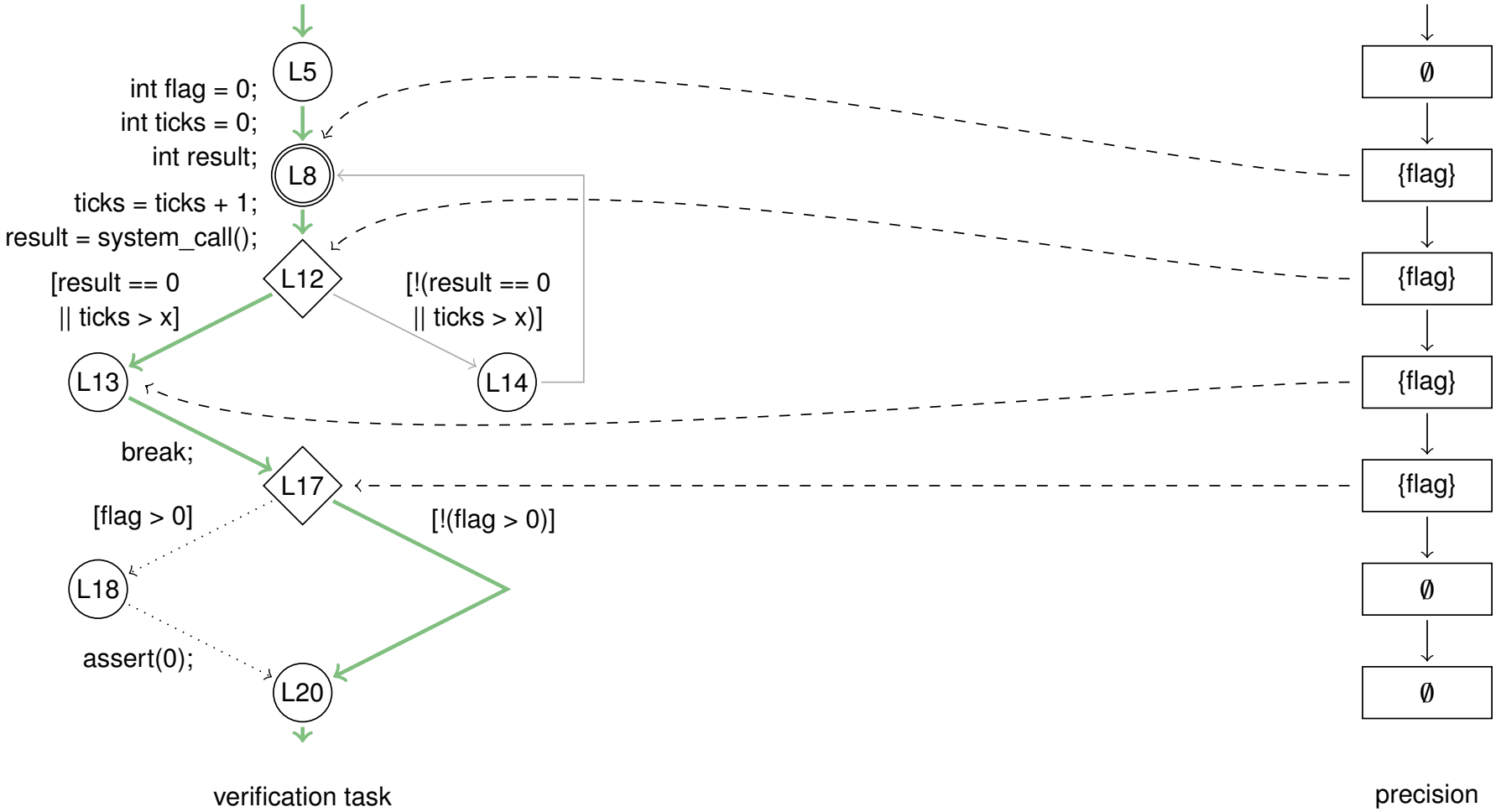


interpolants

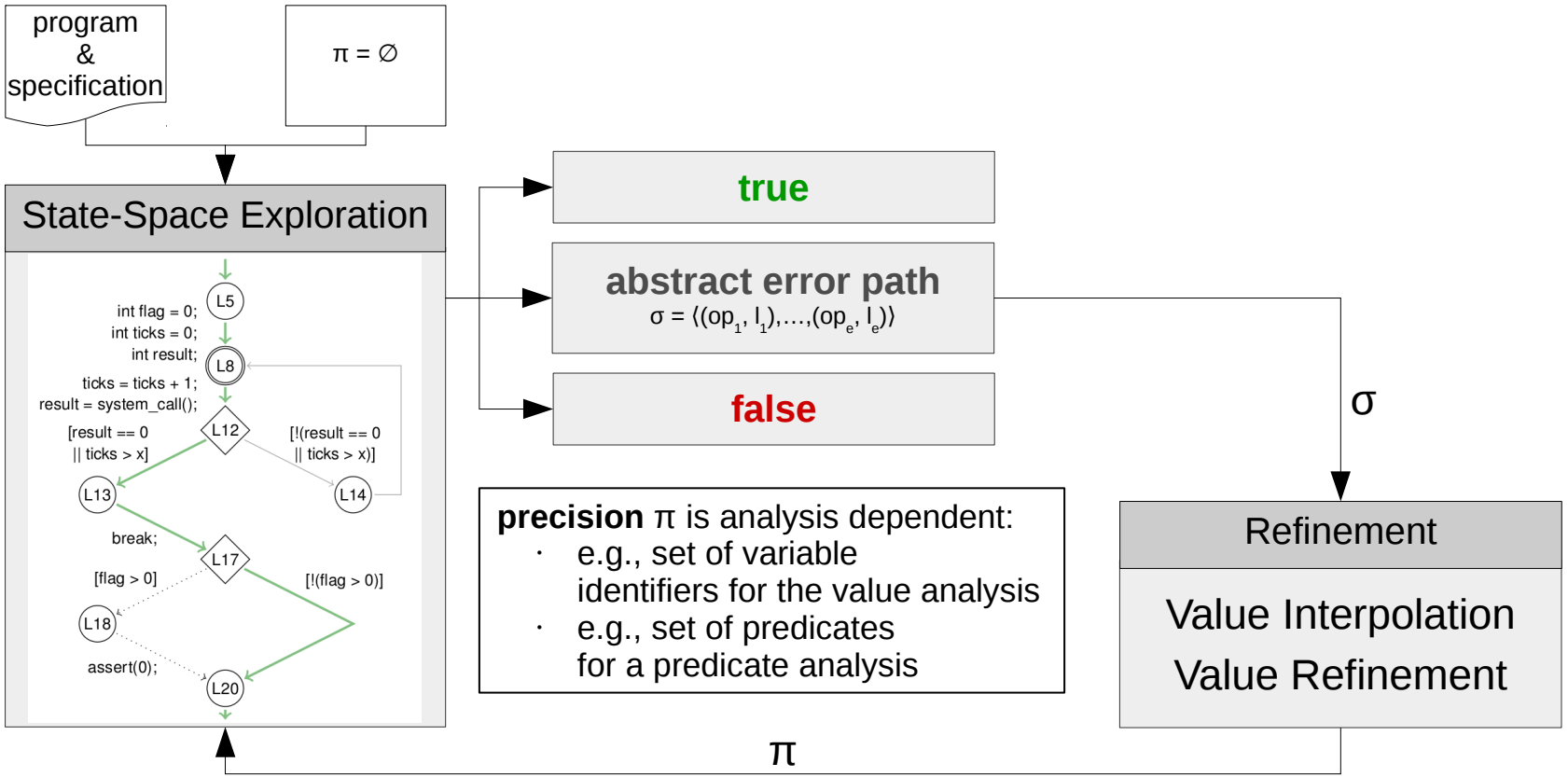


precision

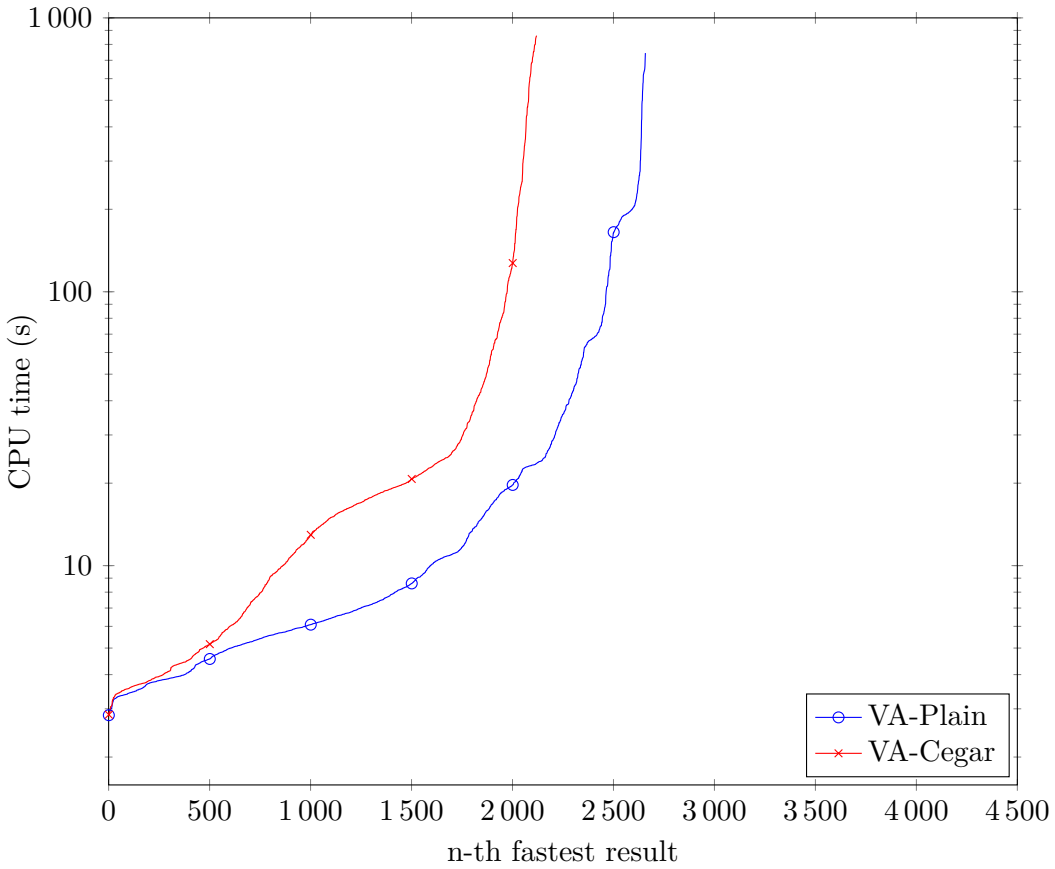
Value Refinement Avoids Counterexample and Loop



One Iteration of the CEGAR Loop Completed



Value Analysis Based on CEGAR in Action



- Significant improvements for some verification tasks
- Significant regressions for other verification tasks
- Value Analysis based on CEGAR solves about 500 verification task *less*

Value Analysis Based on CEGAR in Action

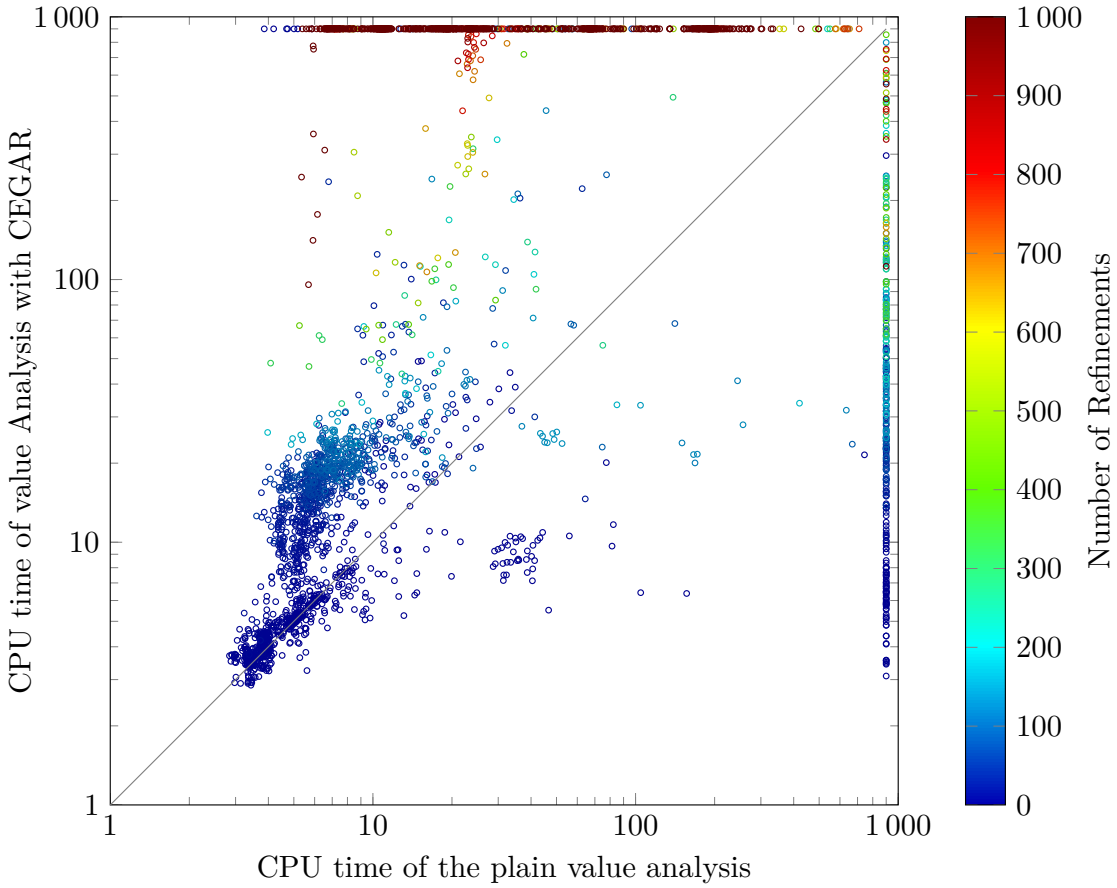
New Problem

The Value Analysis based on CEGAR is, in total, less effective

Proposition

Inspect the overhead introduced by CEGAR

Value Analysis Based on CEGAR — Number of Refinements

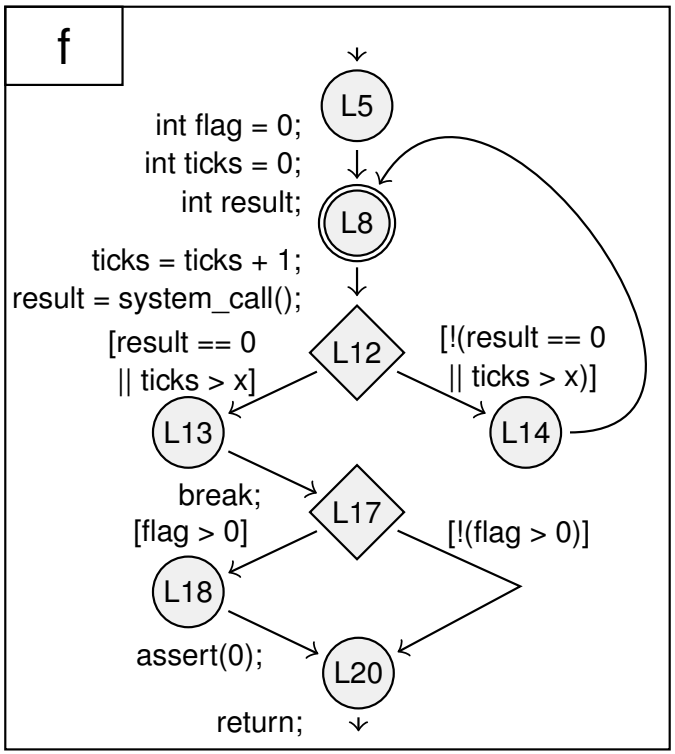


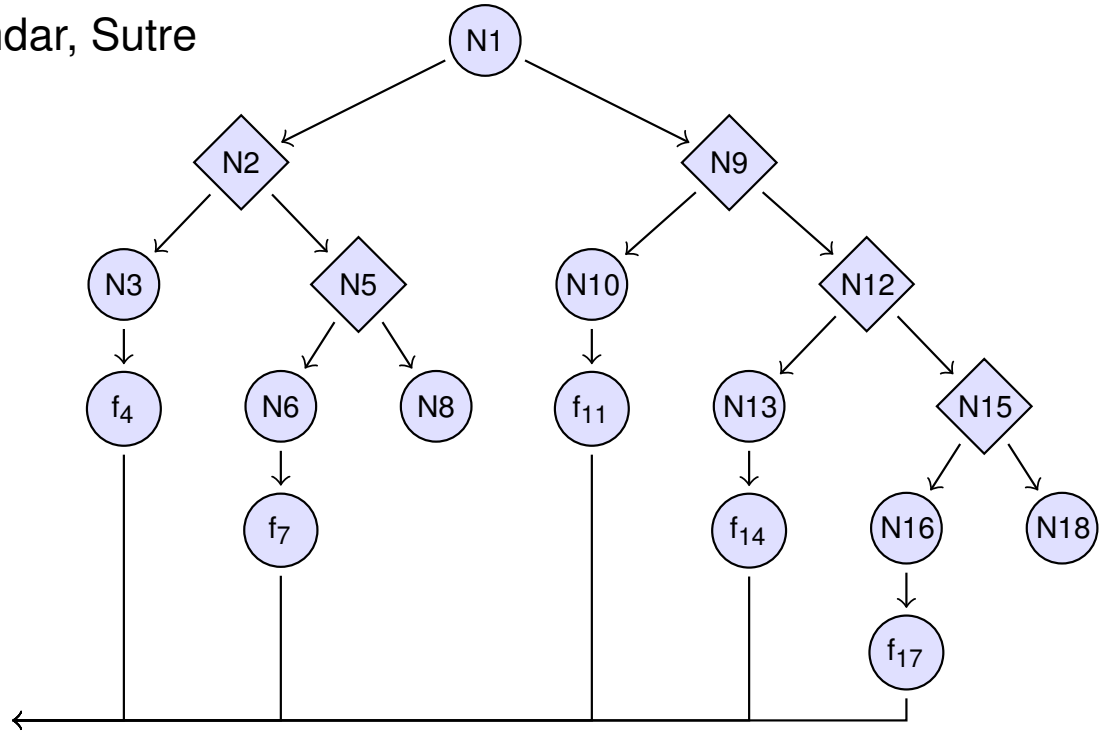
Insights:

- Solved by both: usually < 200 refinements
- Solved only by VA-Cegar: usually < 500 refinements
- Solved only by VA-Plain: usually > 1 000 refinements

Lazy Abstraction — Principle

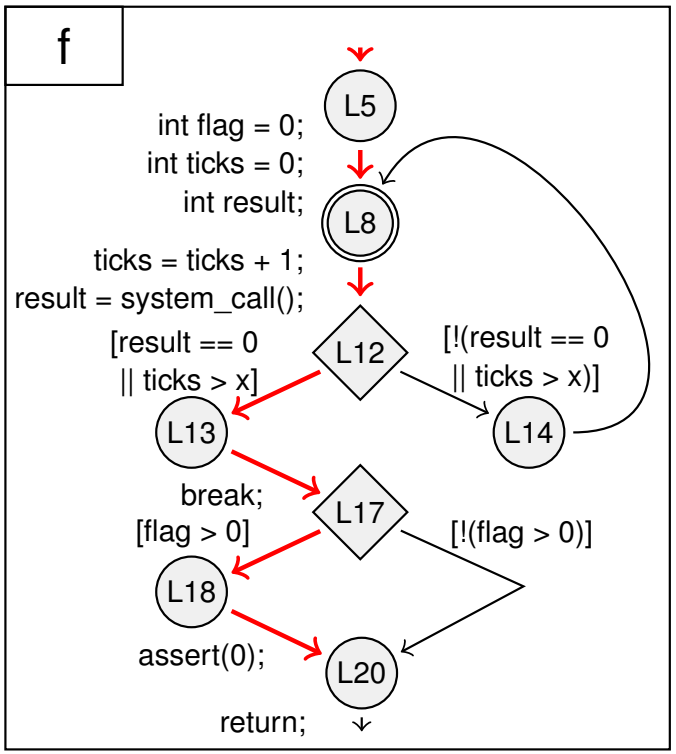
POPL'02, Henzinger, Jhala, Majumdar, Sutre

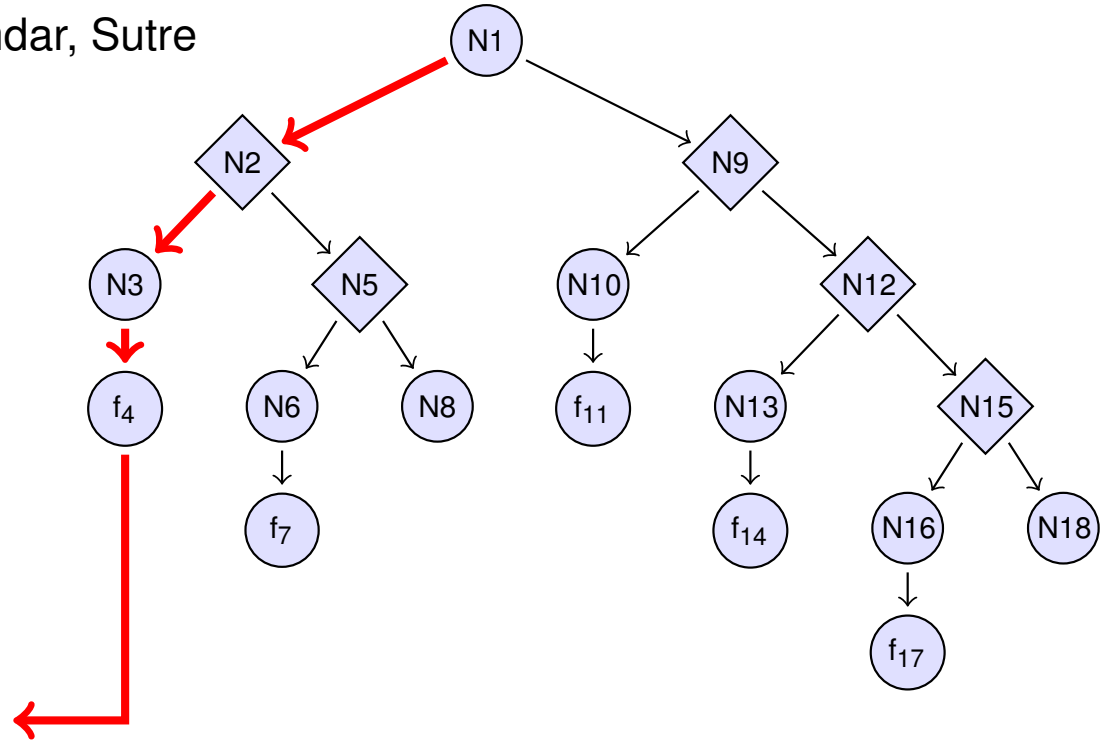




Lazy Abstraction Often Leads to Repeated Refinements

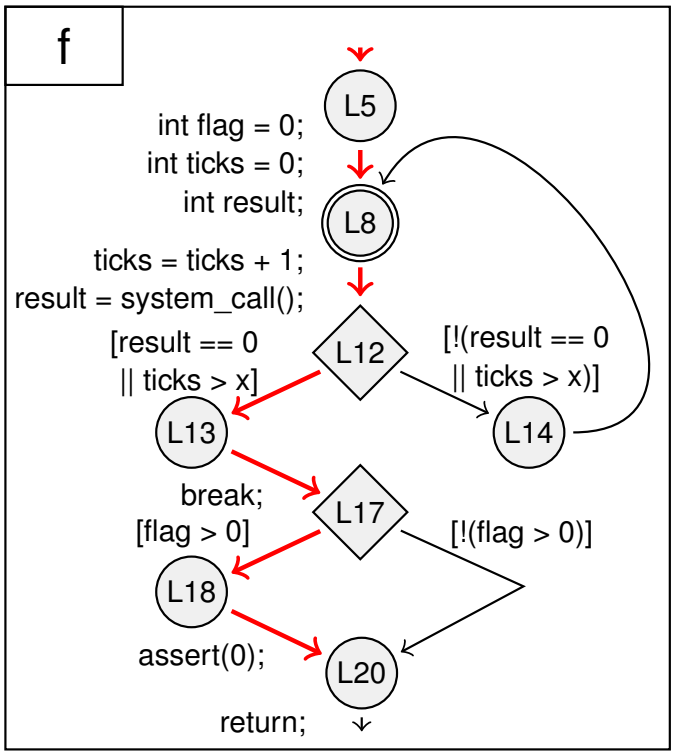
POPL'02, Henzinger, Jhala, Majumdar, Sutre

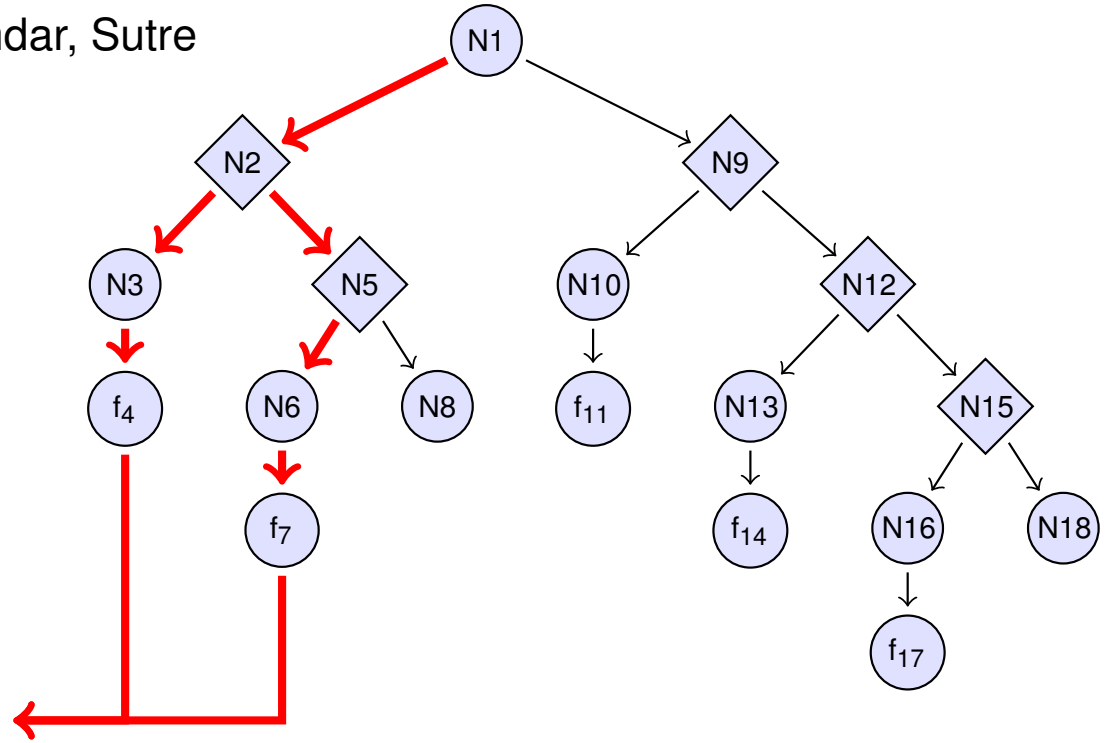




Lazy Abstraction Often Leads to Repeated Refinements

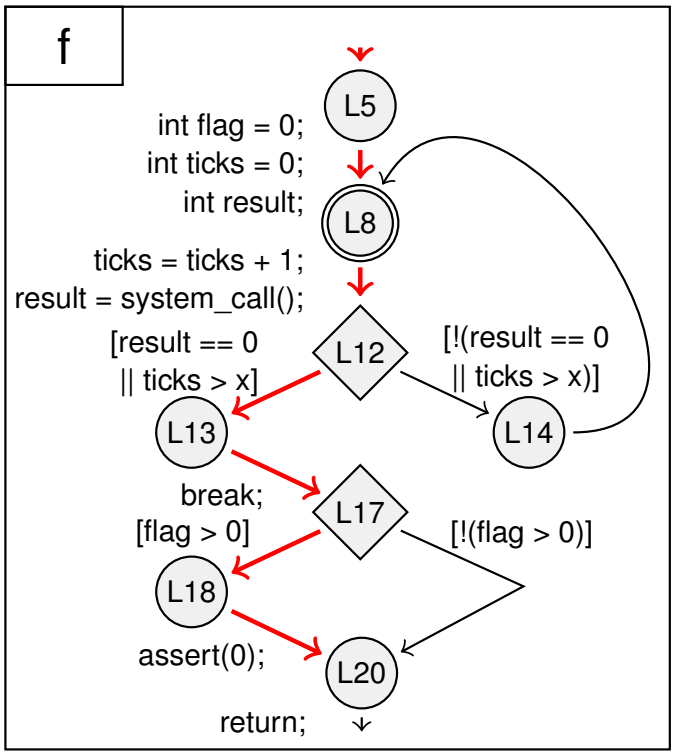
POPL'02, Henzinger, Jhala, Majumdar, Sutre

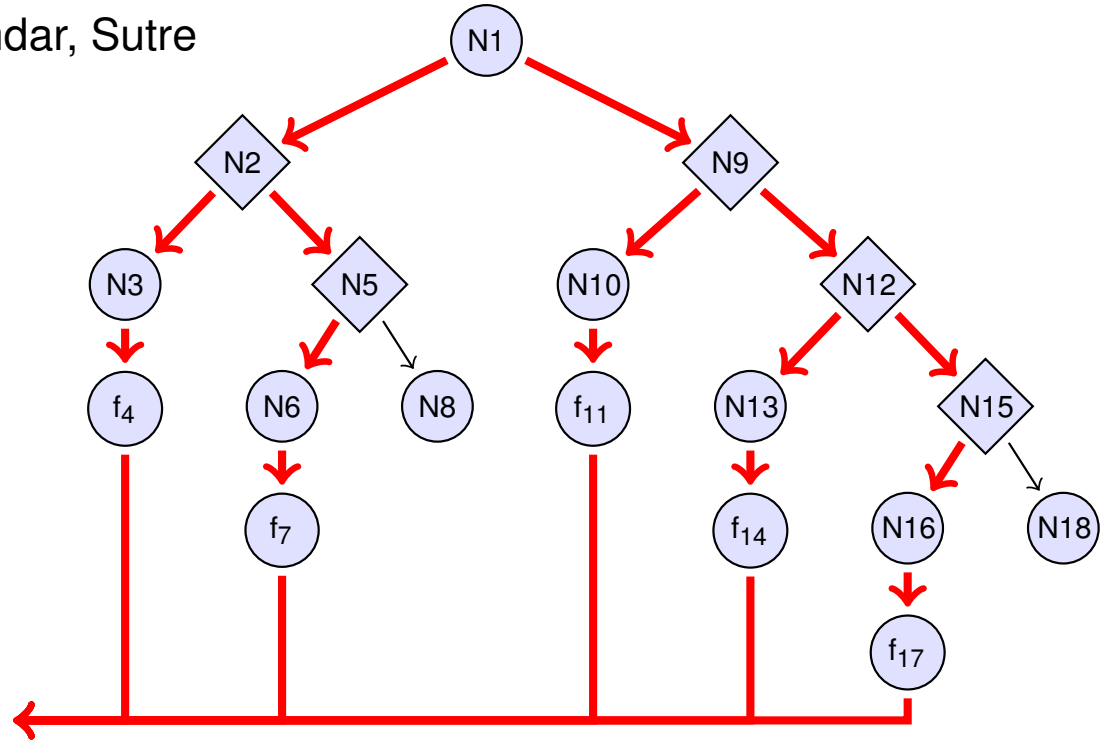




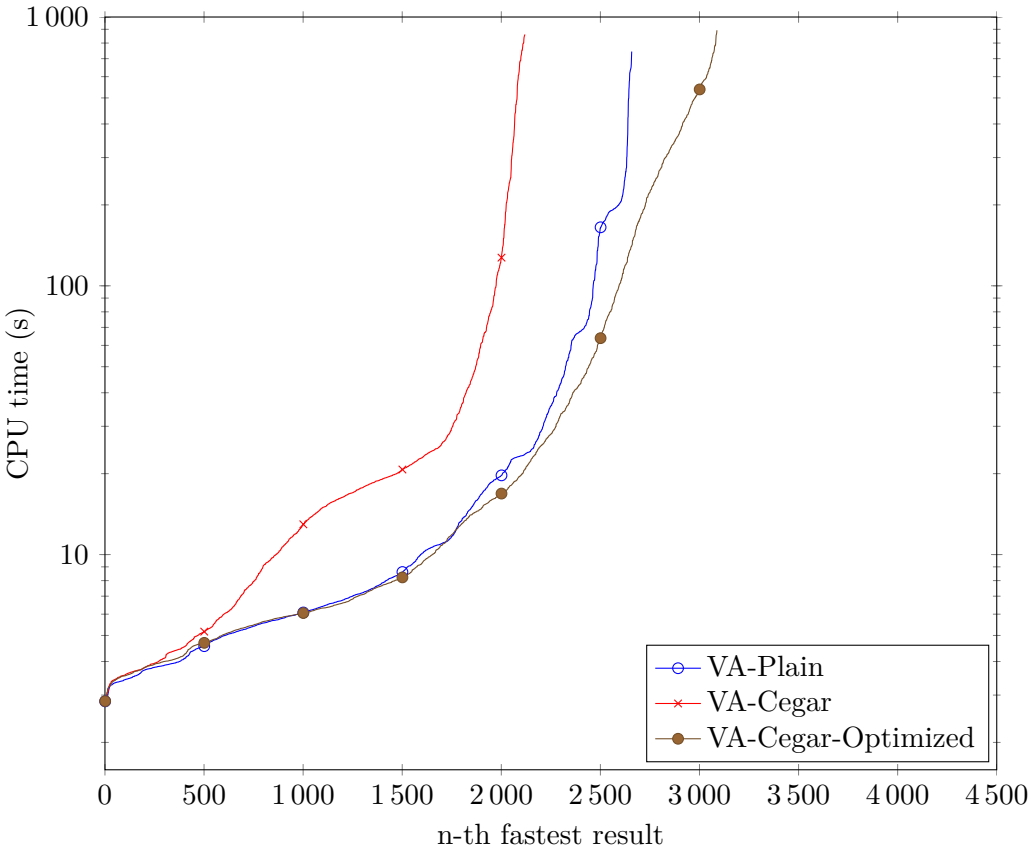
Lazy Abstraction Often Leads to Repeated Refinements

POPL'02, Henzinger, Jhala, Majumdar, Sutre



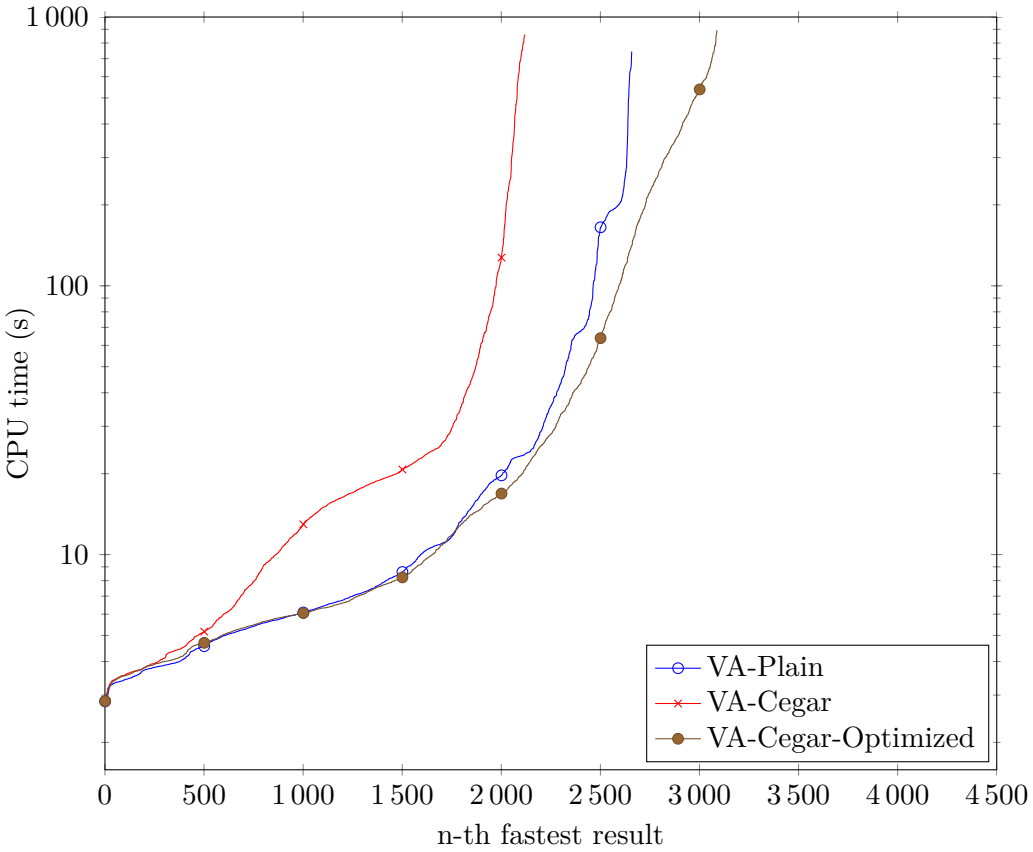


Value Analysis Based on Optimized CEGAR in Action



- Lazy abstraction unsuitable
- **Prefer scoped precision** over localized precision
- **Prefer restart at root** over restart deep within state space

Value Analysis Based on Optimized CEGAR in Action



- Lazy abstraction unsuitable
- **Prefer scoped precision** over localized precision
- **Prefer restart at root** over restart deep within state space

- Value Refinement applicable to other analyses
  - Octagon Analysis
  - Symbolic Execution

Value Analysis Based on CEGAR — Summary

Solution

The integration of CEGAR results in a more effective analysis

Problem

The Value Analysis still cannot handle non-determinism well

```
1 #include <assert.h>
2 int main() {
3   int a;
4   if (a != 1) {
5     if (a == 1)
6       assert(0);
7   }
8 }
```

The Value Analysis wrongly reports FALSE for this verification task

Value Analysis Based on CEGAR — Summary

Solution

The integration of CEGAR results in a more effective analysis

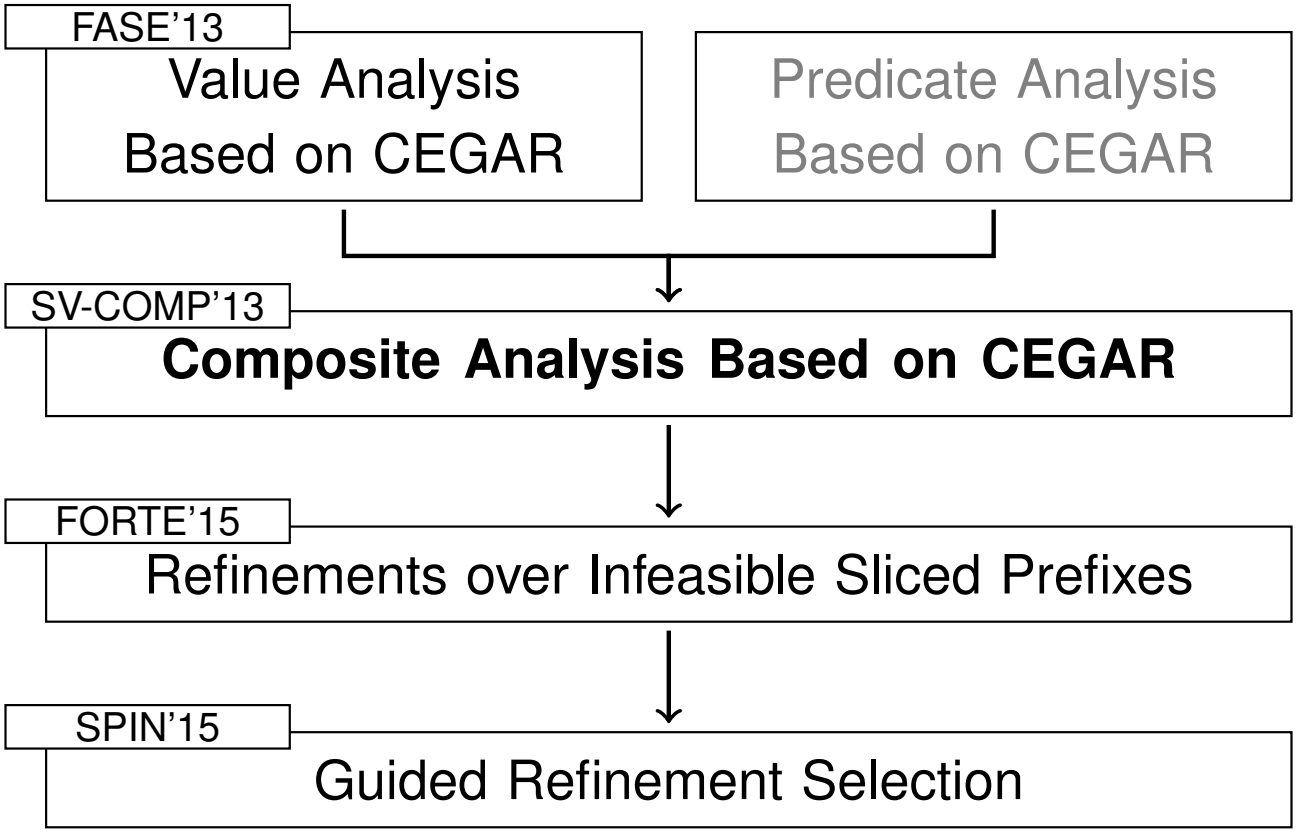
Problem

The Value Analysis still cannot handle non-determinism well

```
1 #include <assert.h>
2 int main() {
3   int a;
4   if (a != 1) {
5     if (a == 1)
6       assert(0);
7   }
8 }
```

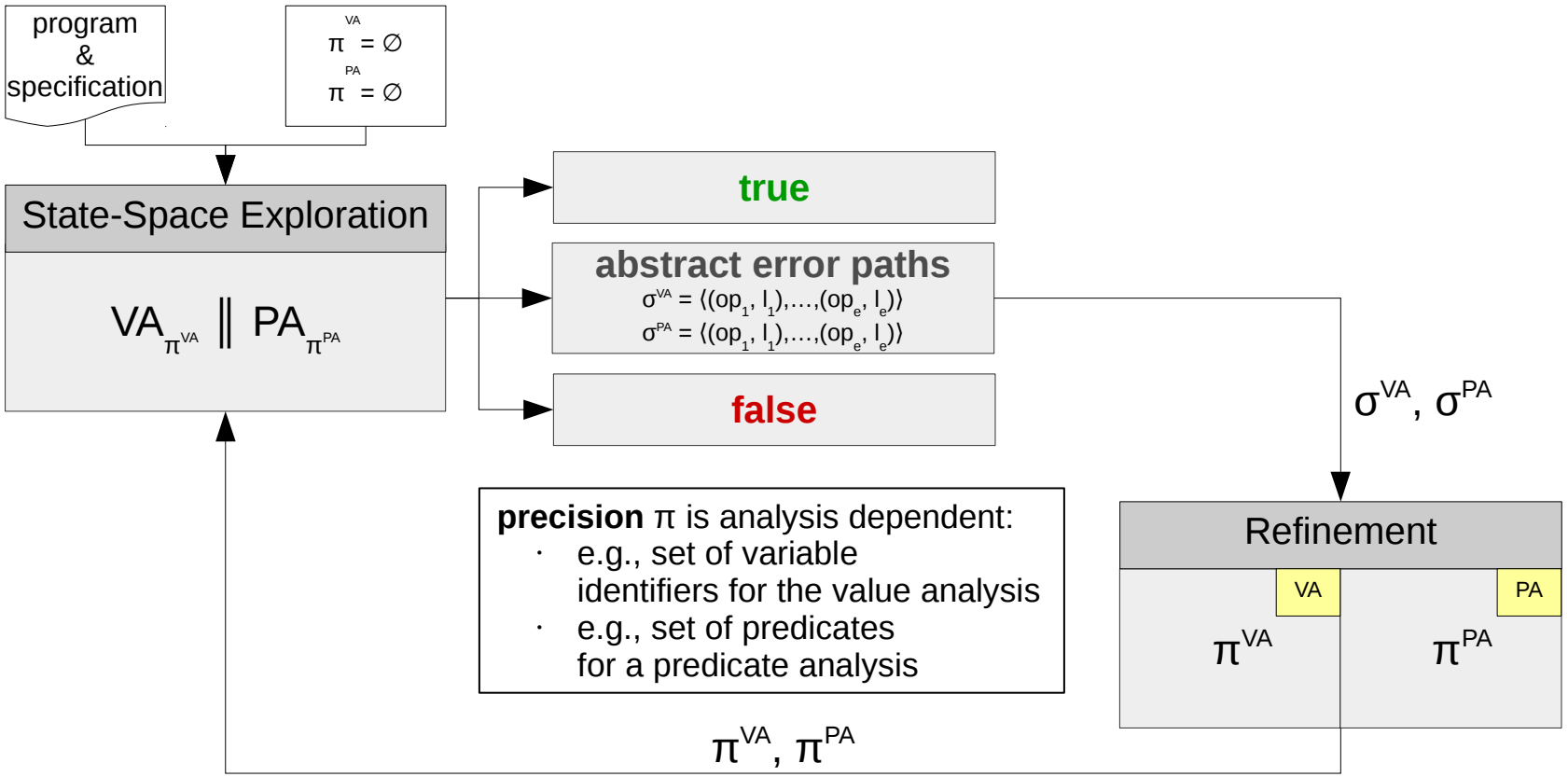
The Value Analysis wrongly reports FALSE for this verification task

Outline

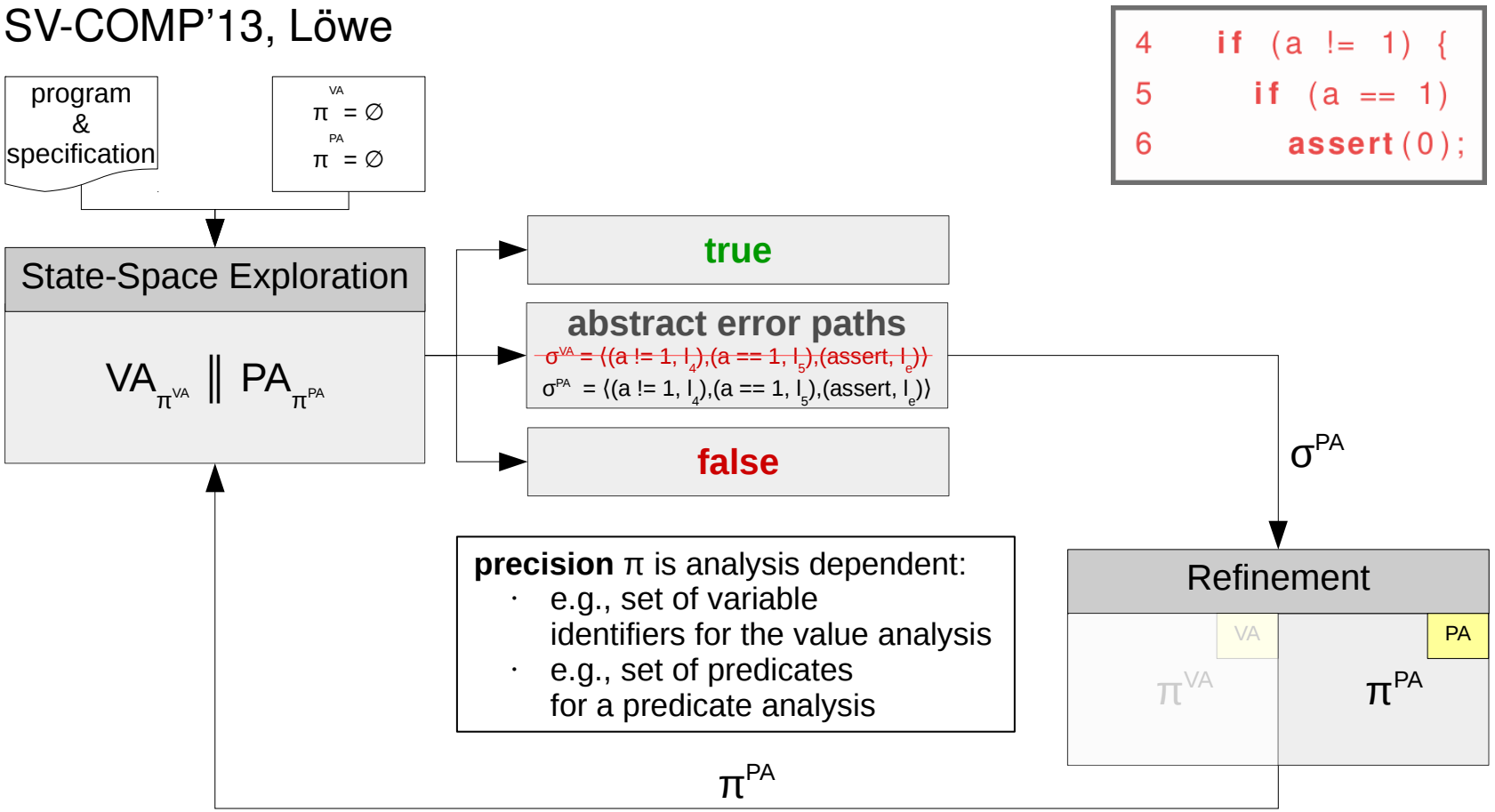

FASE'13
Value Analysis Based on CEGAR
Predicate Analysis Based on CEGAR
SV-COMP'13
Composite Analysis Based on CEGAR
FORTE'15
Refinements over Infeasible Sliced Prefixes
SPIN'15
Guided Refinement Selection

Composite Analysis Based on CEGAR

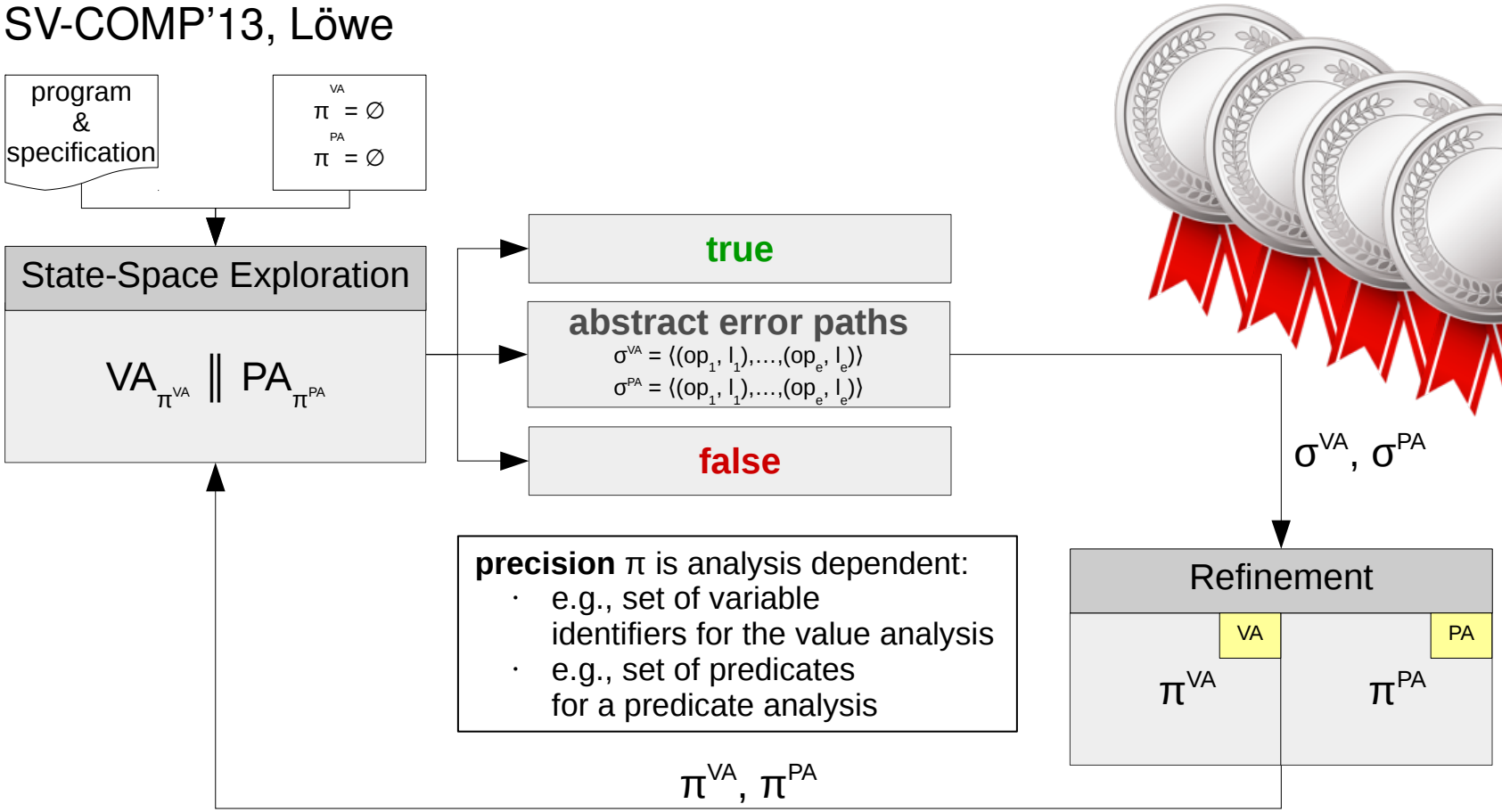
SV-COMP'13, Löwe



Composite Analysis Based on CEGAR

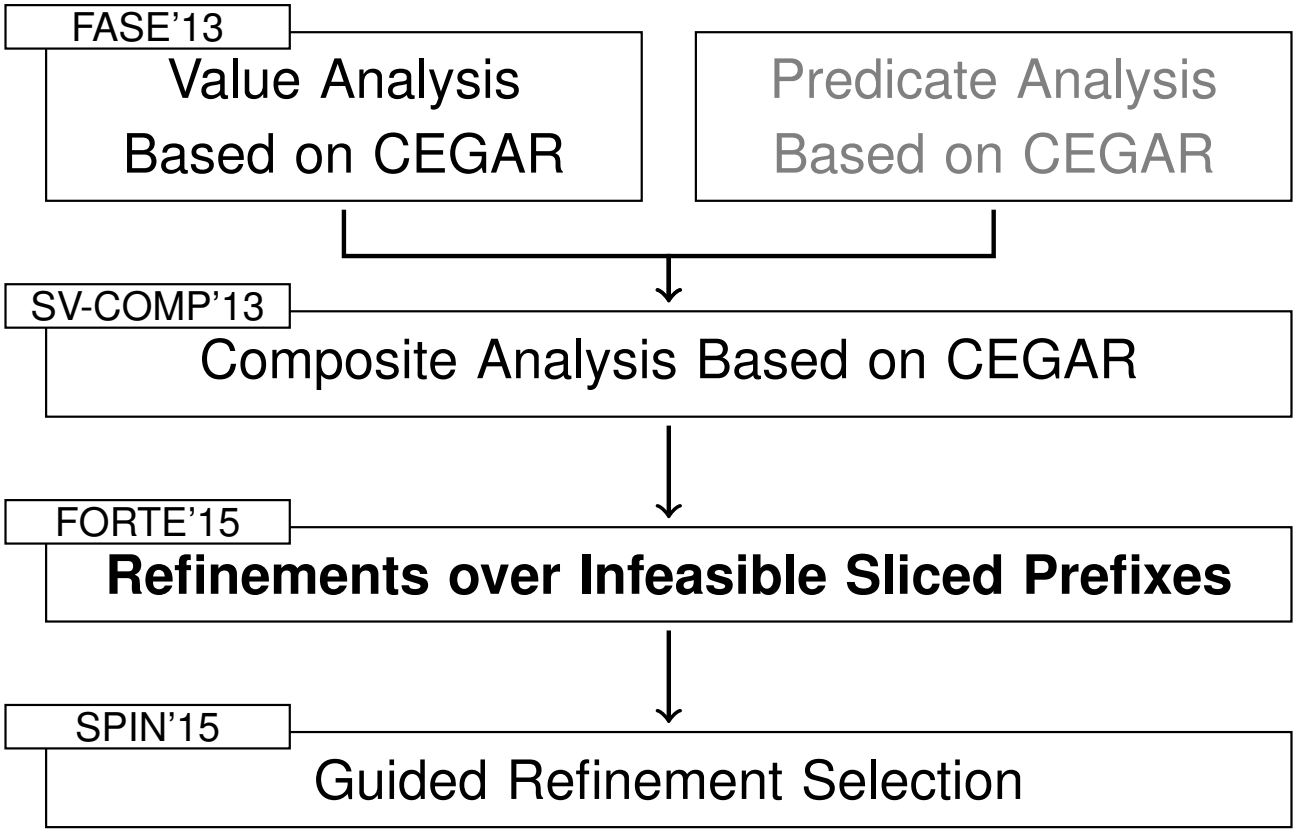


Composite Analysis Based on CEGAR



Wins four silver medals in SV-COMP'13, including "Overall"

Outline


FASE'13
Value Analysis Based on CEGAR
Predicate Analysis Based on CEGAR
SV-COMP'13
Composite Analysis Based on CEGAR
FORTE'15
Refinements over Infeasible Sliced Prefixes
SPIN'15
Guided Refinement Selection

Interpolation-Based Refinement

Statement

Interpolation is ideally suited for refinement of precision

Interpolation-Based Refinement

Statement

Interpolation is ideally suited for refinement of precision

Unfortunately, this is only true in theory

Problem

In practice, we distinguish between “good” or “bad” refinements which are the result of “good” or “bad” interpolants

“Good” and “Bad” Interpolants

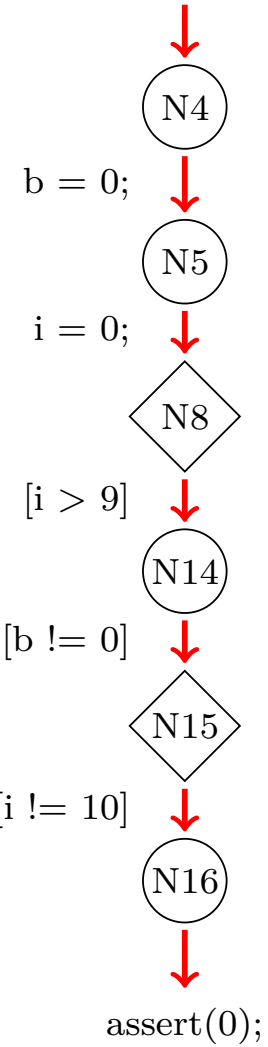
```
 1 #include <assert.h>
 2 extern int f(int x);
 3 int main() {
 4   int b = 0;
 5   int i = 0;
 6
 7   while(1) {
 8     if(i > 9) {
 9       break;
10     }
11     f(i++);
12   }
13
14   if(b != 0) {
15     if(i != 10) {
16       assert(0);
17     }
18   }
19 }
```

verification task

“Good” and “Bad” Interpolants

```
 1 #include <assert.h>
 2 extern int f(int x);
 3 int main() {
 4   int b = 0;
 5   int i = 0;
 6
 7   while(1) {
 8     if(i > 9) {
 9       break;
10     }
11     f(i++);
12   }
13
14   if(b != 0) {
15     if(i != 10) {
16       assert(0);
17     }
18   }
19 }
```

verification task

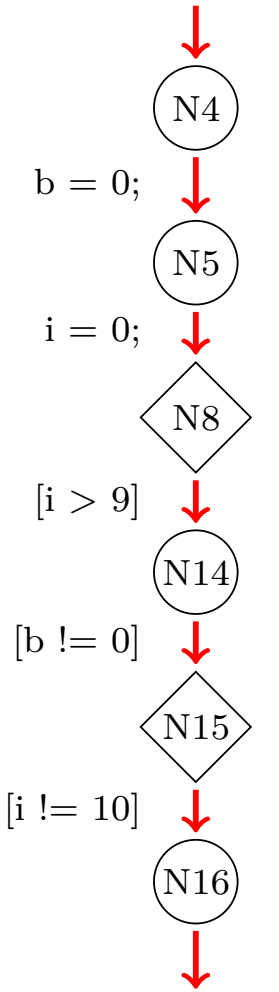


error path

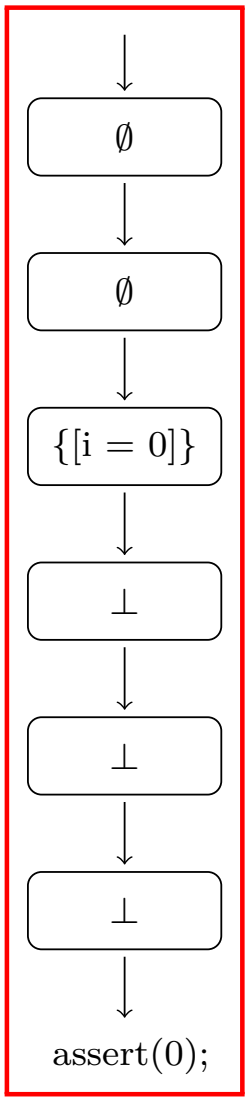
“Good” and “Bad” Interpolants

```
1  #include <assert.h>
2  extern int f(int x);
3  int main() {
4    int b = 0;
5    int i = 0;
6
7    while(1) {
8      if(i > 9) {
9        break;
10     }
11     f(i++);
12   }
13
14   if(b != 0) {
15     if(i != 10) {
16       assert(0);
17     }
18   }
19 }
```

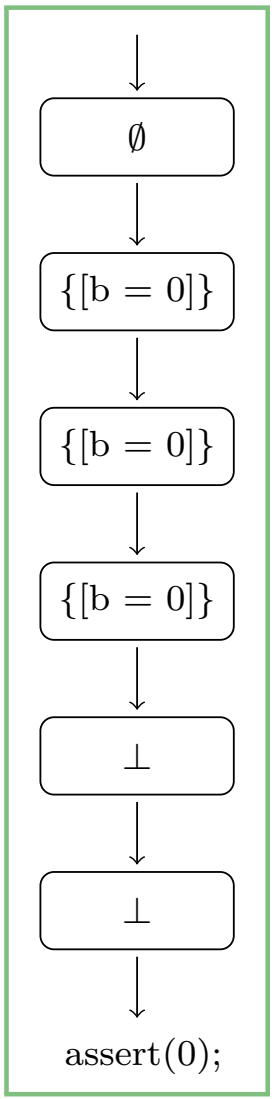
verification task



error path



bad sequence

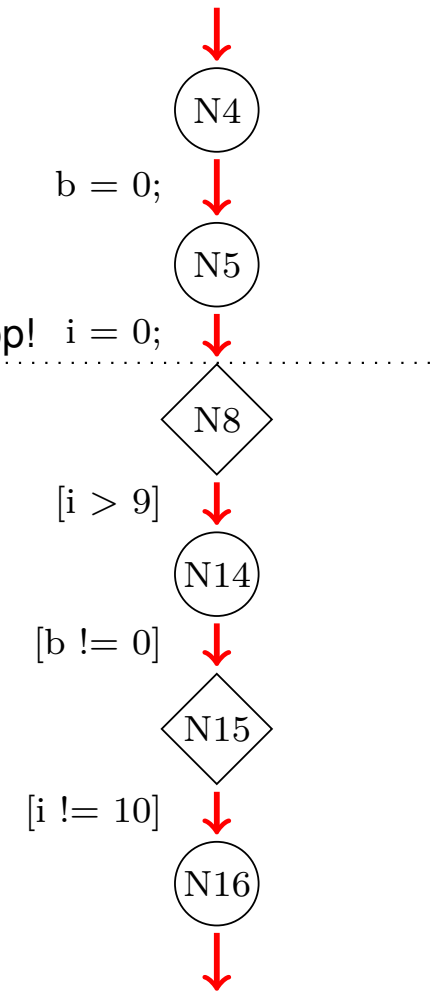


good sequence

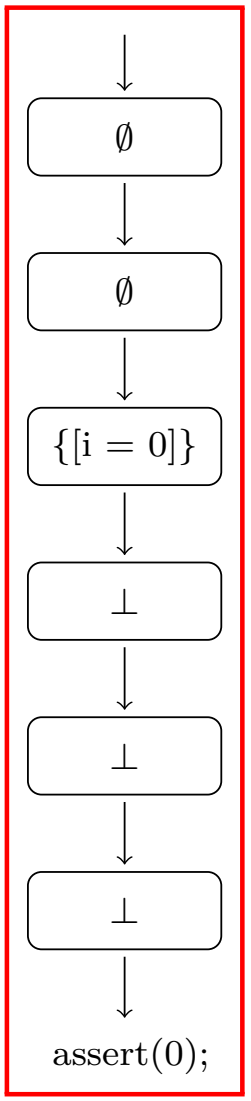
“Good” and “Bad” Interpolants

```
1  #include <assert.h>
2  extern int f(int x);
3  int main() {
4    int b = 0;
5    int i = 0;
6
7    while(1) {
8      if(i > 9) {
9        break;
10     }
11     f(i++);
12   }
13
14   if(b != 0) {
15     if(i != 10) {
16       assert(0);
17     }
18   }
19 }
```

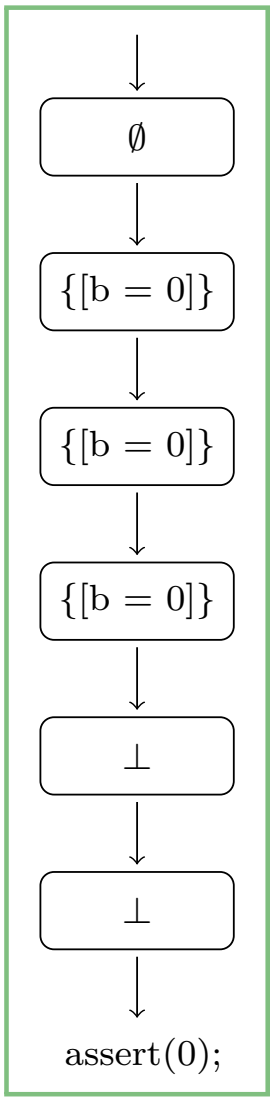
verification task



error path



bad sequence



good sequence

Standard Interpolation Engines are Black Boxes

- Typically, interpolation engines use internal heuristics
- Resolution process cannot be controlled from outside

Problem

We are stuck to the result of the interpolation engine,
be it good or bad

Proposition

Remodel interpolation problem
such that different interpolant sequences can be extracted

Standard Interpolation Engines are Black Boxes

- Typically, interpolation engines use internal heuristics
- Resolution process cannot be controlled from outside

Problem

We are stuck to the result of the interpolation engine,
be it good or bad

Proposition

Remodel interpolation problem
such that different interpolant sequences can be extracted

Standard Interpolation Engines are Black Boxes

- Typically, interpolation engines use internal heuristics
- Resolution process cannot be controlled from outside

Problem

We are stuck to the result of the interpolation engine,
be it good or bad

Proposition

Remodel interpolation problem
such that different interpolant sequences can be extracted

Extraction of Infeasible Sliced Prefixes — Algorithm

FORTE'15, Beyer, Löwe, Wendler

Algorithm 2: ExtractInfeasibleSlicedPrefixes(σ)

Input : an infeasible path $\sigma = \langle (op_1, l_1), \ldots, (op_m, l_m) \rangle$
Output : a non-empty set $\Sigma = \{\sigma_1, \ldots, \sigma_n\}$ of infeasible sliced prefixes of σ
Variables: a path σ_f that is always feasible

$\Sigma := \emptyset$
$\sigma_f := \langle\rangle$

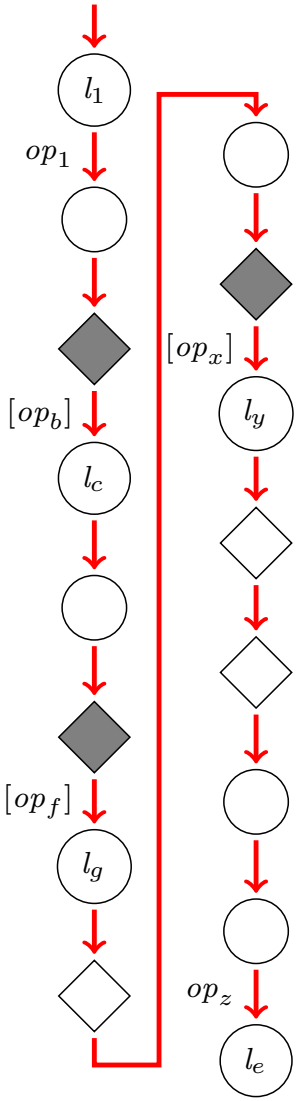
foreach $(op, l) \in \sigma$ **do**
  if $\widehat{\mathrm{SP}}_{\sigma_f \wedge (op,l)}(\top) = \bot$ **then**
    // add $\sigma_f \wedge (op, l)$ to set Σ of infeasible sliced prefixes
    $\Sigma := \Sigma \cup \{\sigma_f \wedge (op, l)\}$

    // append no-op
    $\sigma_f := \sigma_f \wedge ([true], l)$

  else
    // append original pair
    $\sigma_f := \sigma_f \wedge (op, l)$

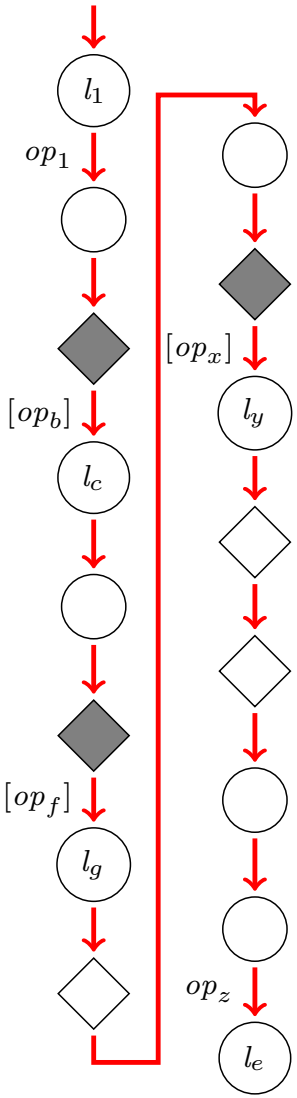
return Σ

Extraction of Infeasible Sliced Prefixes — Example

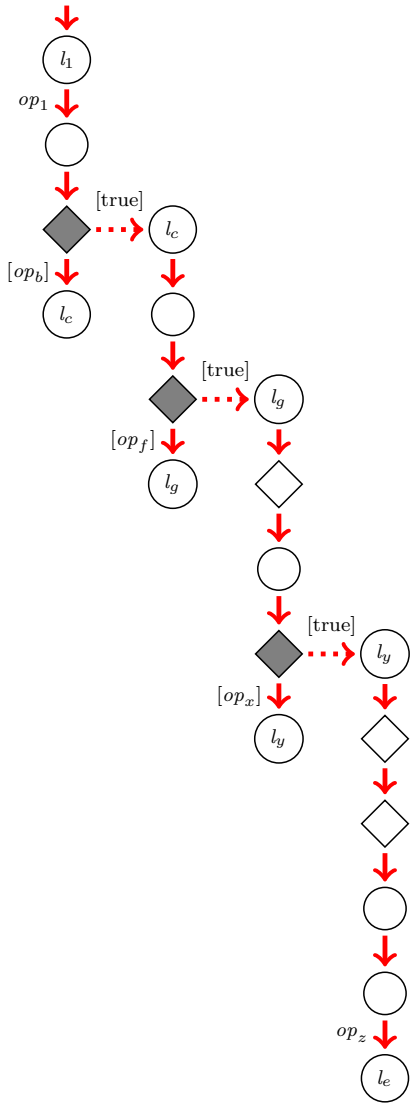


infeasible error path

Extraction of Infeasible Sliced Prefixes — Example

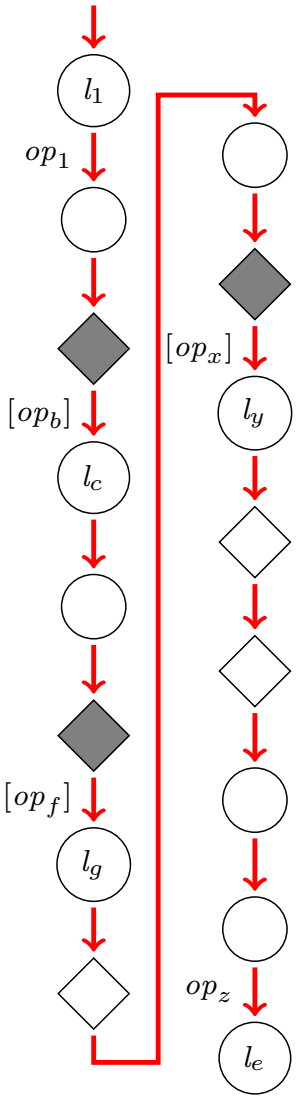


infeasible error path

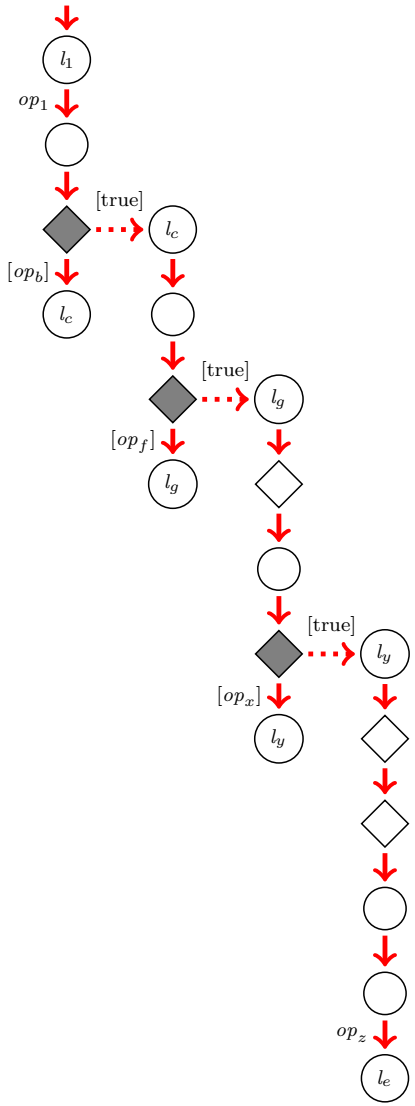


cascade of sliced prefixes

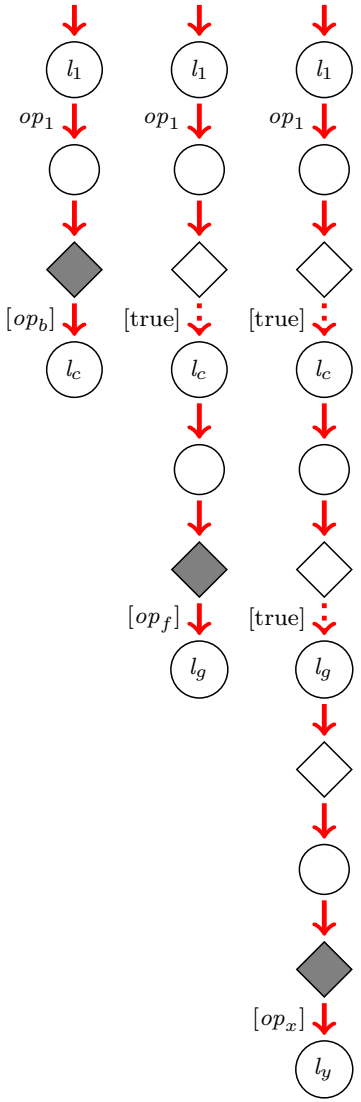
Extraction of Infeasible Sliced Prefixes — Example



infeasible error path



cascade of sliced prefixes



infeasible sliced prefixes

Infeasible Sliced Prefixes

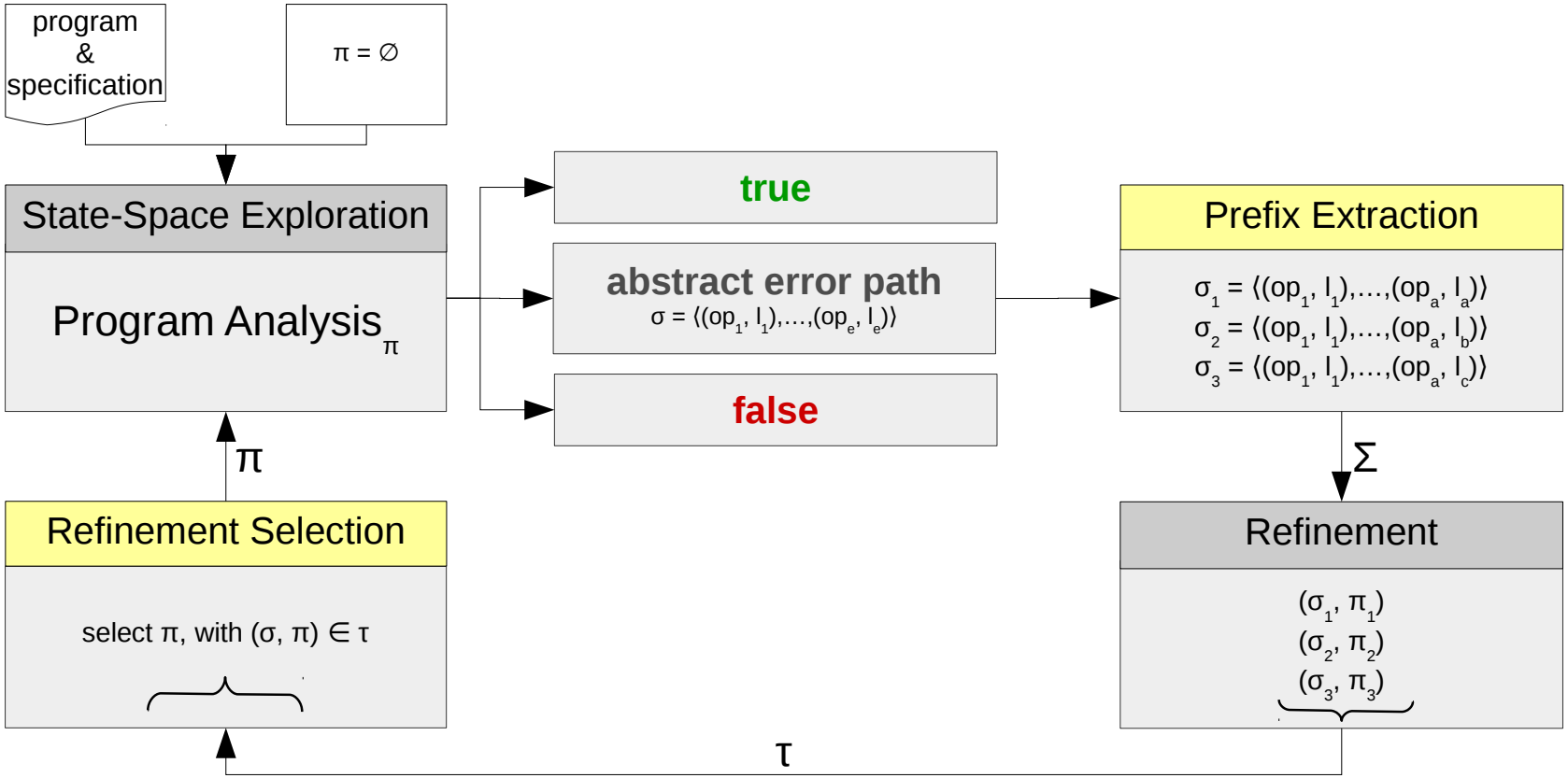
Theorem

Any infeasible sliced prefix φ,
that is extracted from an infeasible error path σ,
can be used for interpolation to exclude the original error path σ

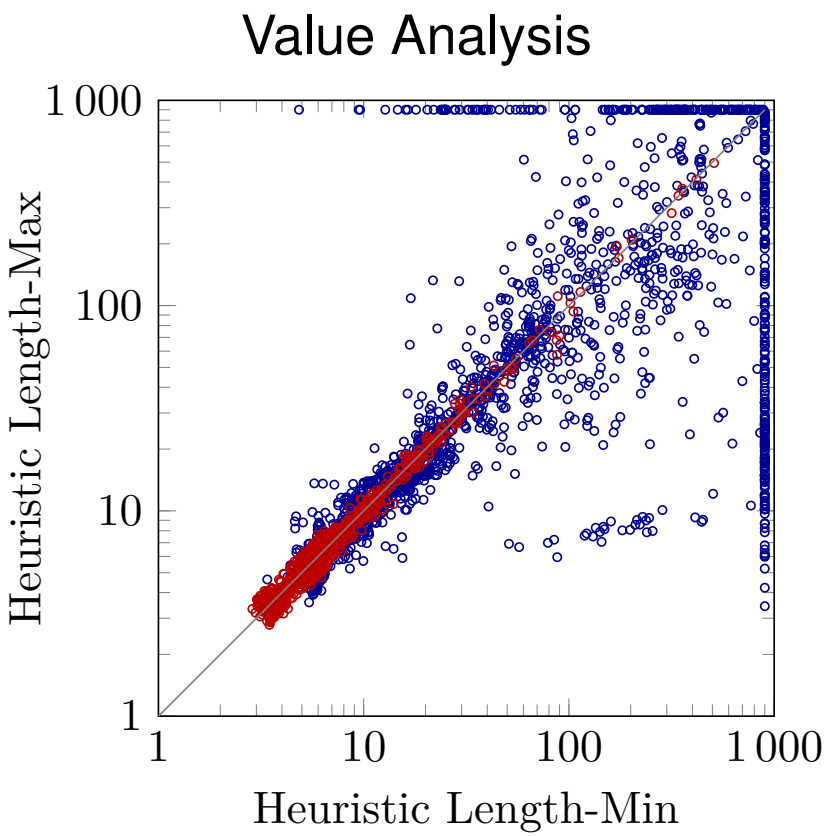
Proposition

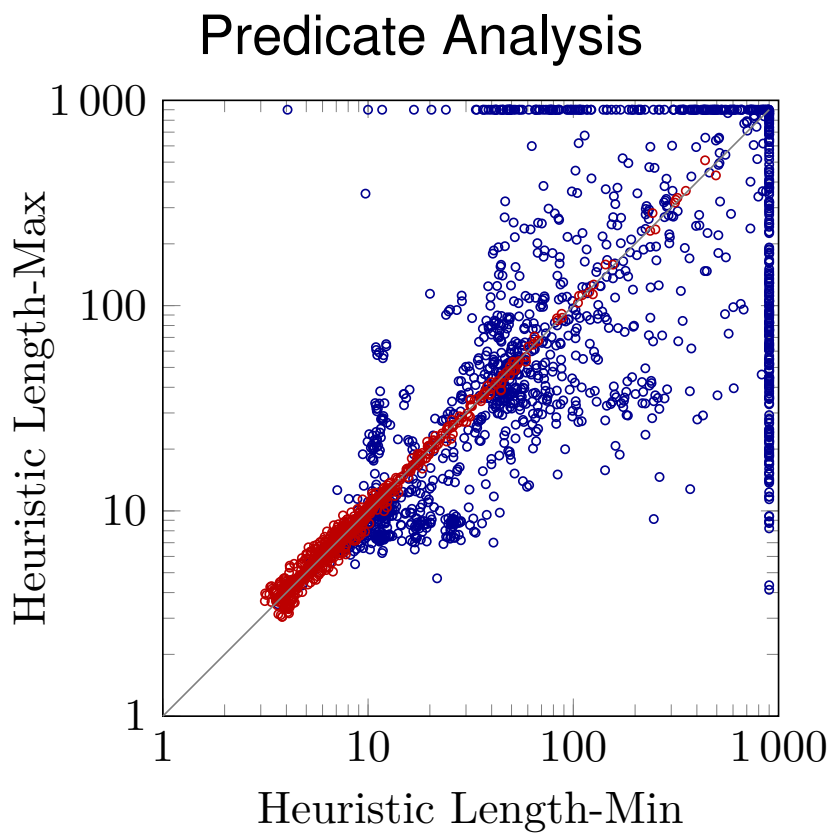
Use different infeasible sliced prefixes to get different interpolants

CEGAR using Infeasible Sliced Prefixes

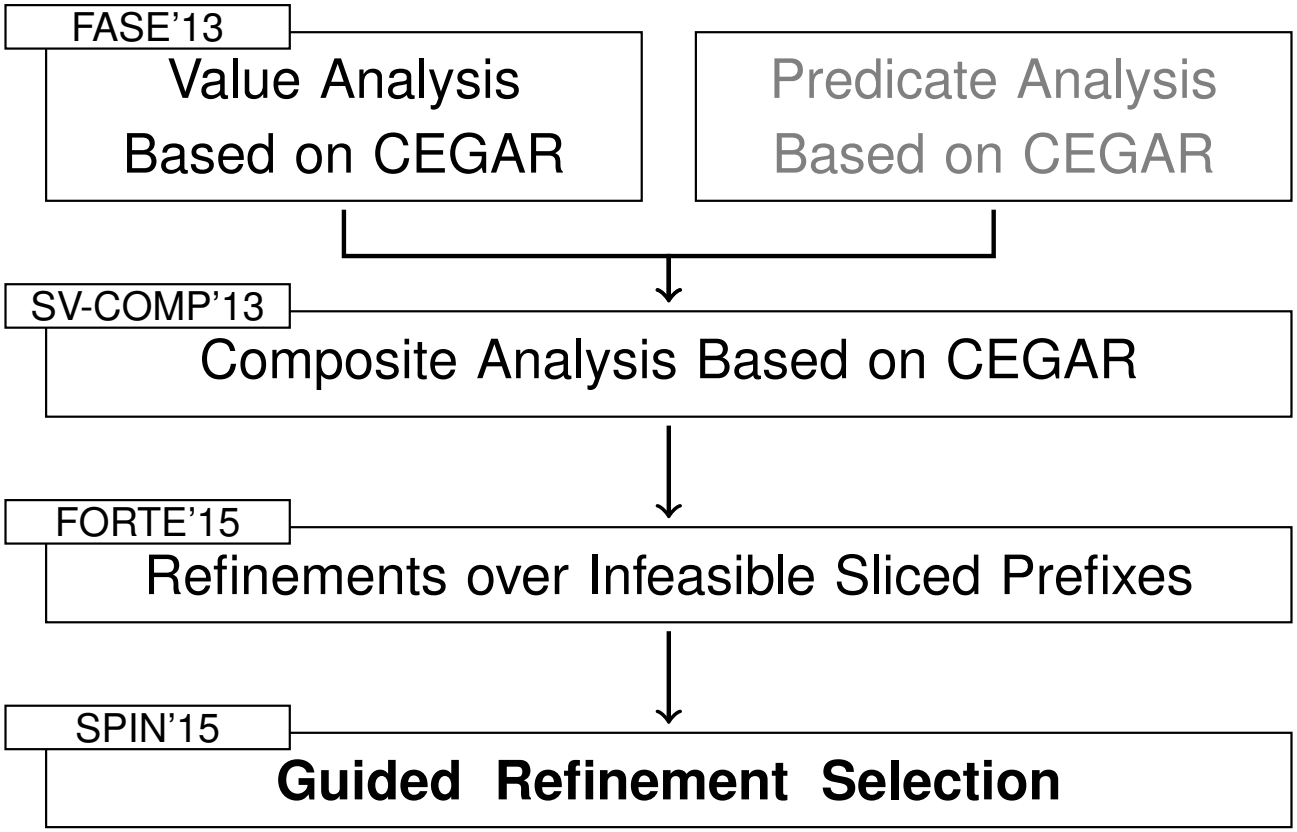


Infeasible Sliced Prefixes — Impact





Outline

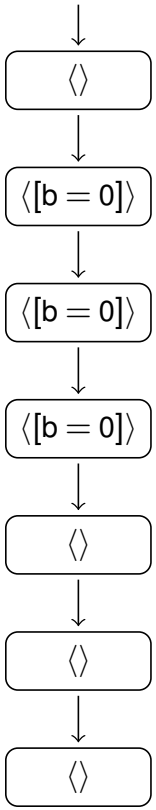

FASE'13
Value Analysis Based on CEGAR
Predicate Analysis Based on CEGAR
SV-COMP'13
Composite Analysis Based on CEGAR
FORTE'15
Refinements over Infeasible Sliced Prefixes
SPIN'15
Guided Refinement Selection

Refinement Selection

Proposition

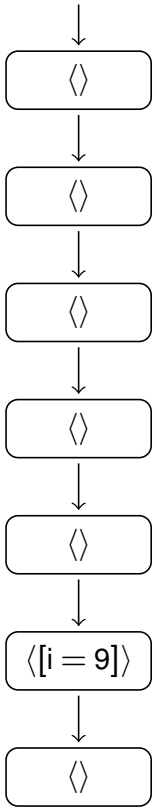
Try to select a favorable refinement
from the set of available refinement choices

Refinement Selection by Domain-Type Score



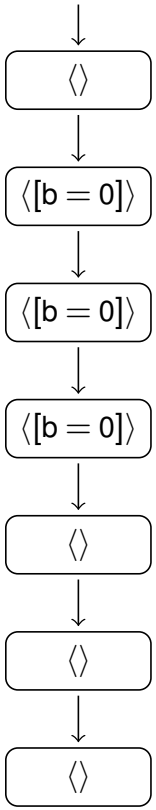
Min

- Rationale: tracking either boolean or loop-counter variables has influence on verification process
- Implementation: compute score of precision, as aggregate of the domain-types of variables referenced in precision
- Scale:
  - Min: prefer tracking variables with few valuations
  - Max: prefer tracking variables with many valuations



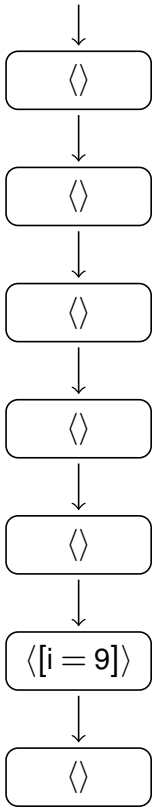
Max

Refinement Selection by Width of Precision



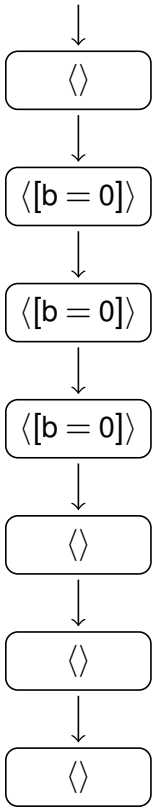
Max

- Rationale: tracking information either over a narrow or wide sequence has influence on verification process
- Implementation: compute width of precision, as range over which new information has to be tracked
- Scale:
  - Min: prefer tracking information over few transitions
  - Max: prefer tracking information over many transitions



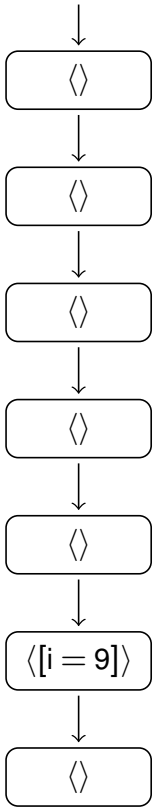
Min

Refinement Selection by Depth of Pivot Location



Min

- Rationale: tracking information either close to initial or close to error location has influence on verification process
- Implementation: compute depth of precision, as offset from which on new information has to be tracked
- Scale:
  - Min: prefer tracking information close to initial location
  - Max: prefer tracking information close to error location



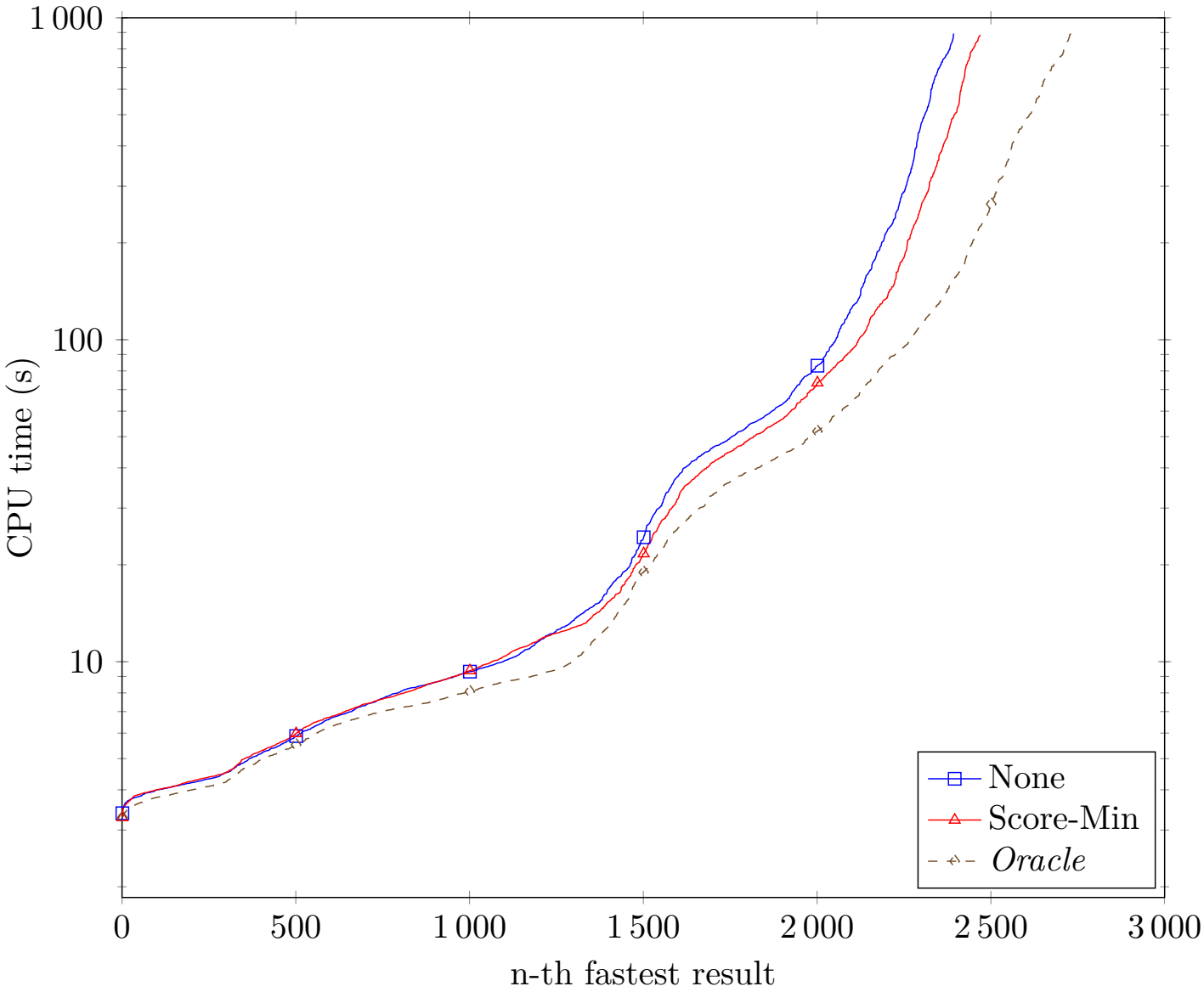
Max

Refinement Selection — More Effective

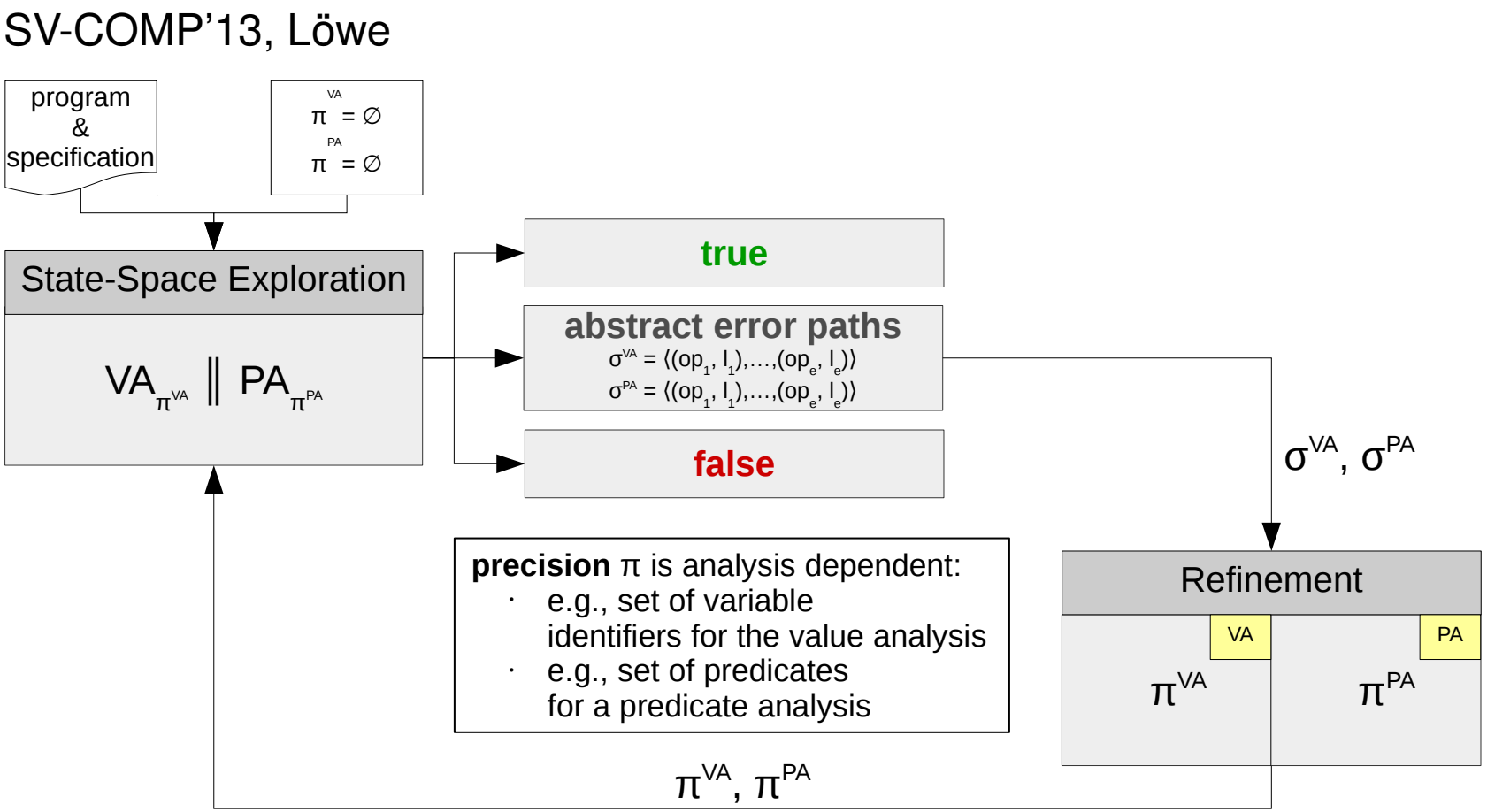
		ControlFlow	DD64	ECA	ProductLines	Sequential	Overall
total		48	2 120	1 140	597	62	4 283
Length	Min	**43**	1 471	177	356	27	2 328
	Max	31	1 564	194	332	20	2 385
Width	Min	37	1 464	**280**	346	24	2 406
	Max	36	1 545	174	312	27	2 337
Depth	Min	34	1 526	152	**358**	26	2 342
	Max	36	1 468	206	329	24	2 317
Score	Min	40	**1592**	217	339	**29**	**2470**
	Max	23	1 459	191	302	20	2 238
None		42	1 525	218	324	28	2 393
Optimal		43	1 592	280	358	29	2 561
Oracle		43	1 650	337	412	31	2 732

Number of solved verification tasks for the predicate analysis with refinement selection using different heuristics

Refinement Selection — More Efficient



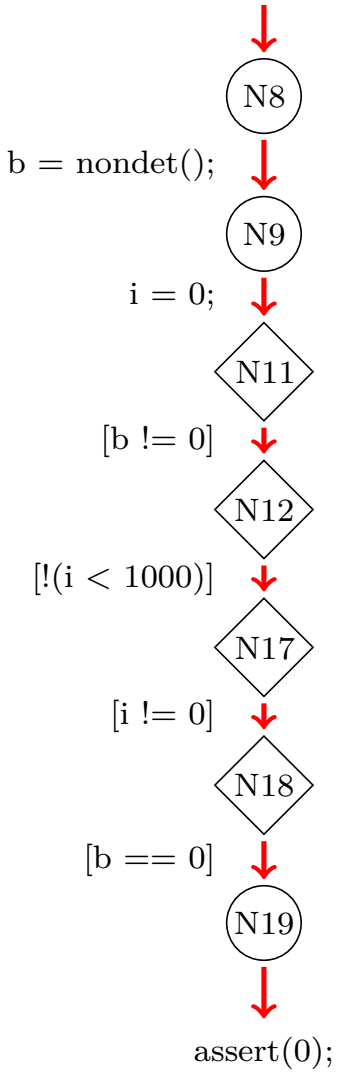
Bringing Refinement Selection to the Next Level



Inter-Analysis Refinement Selection

```
1  #include <assert.h>
2
3  extern int nondet();
4  extern int f(int x);
5
6  int main() {
7
8    int b = nondet();
9    int i = 0;
10
11   if (b != 0) {
12     while (i < 1000) {
13       f(i++);
14     }
15   }
16
17   if (i != 0) {
18     if (b == 0) {
19       assert(0);
20     }
21   }
22 }
```

verification task

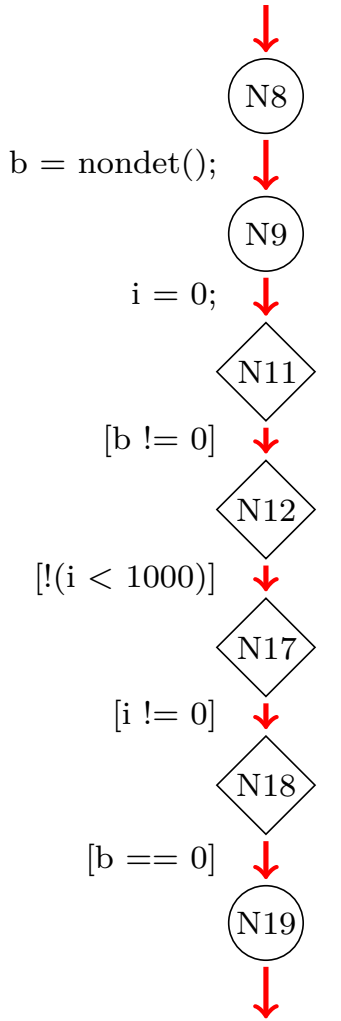


error path

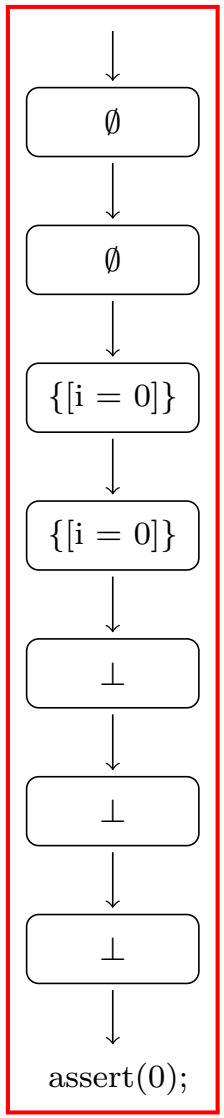
Inter-Analysis Refinement Selection

```
1  #include <assert.h>
2
3  extern int nondet();
4  extern int f(int x);
5
6  int main() {
7
8    int b = nondet();
9    int i = 0;
10
11   if (b != 0) {
12     while (i < 1000) {
13       f(i++);
14     }
15   }
16
17   if (i != 0) {
18     if (b == 0) {
19       assert(0);
20     }
21   }
22 }
```

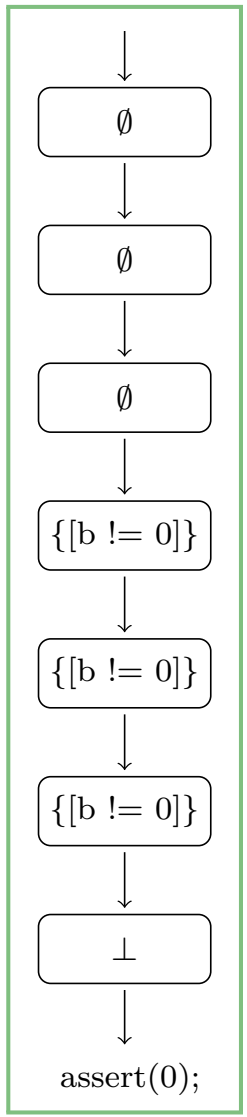
verification task



error path

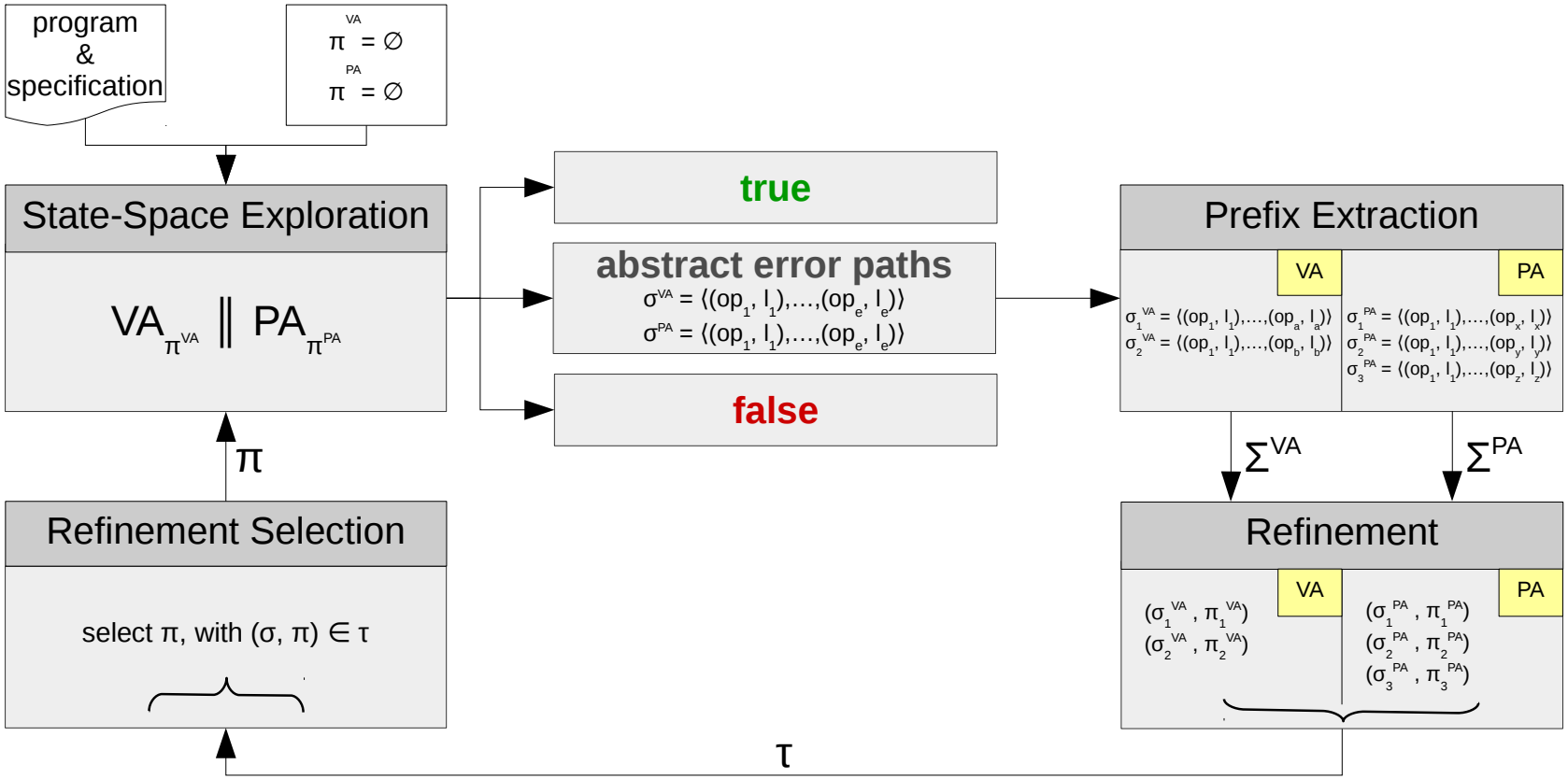


bad sequence



good sequence

CEGAR with Inter-Analysis Refinement Selection



Inter-Analysis Refinement Selection — Most Effective

	BitVectors	ControlFlow	DD64	ECA	Floats	Loops	ProductLines	Sequential	Simple	Overall
total	48	48	2 120	1 140	81	141	597	62	46	4 283
PA	43	42	1 525	218	73	97	321	28	43	2 390
VA-PA-Composition	43	**45**	1 561	**484**	73	89	**390**	**38**	**46**	2 769
VA-PA-Composition–RefSel	**44**	**45**	**1652**	472	**74**	**98**	373	32	45	**2835**

SV-COMP'16, Löwe

Verifier based on inter-analysis refinement selection wins gold medal in category "DeviceDrivers64"

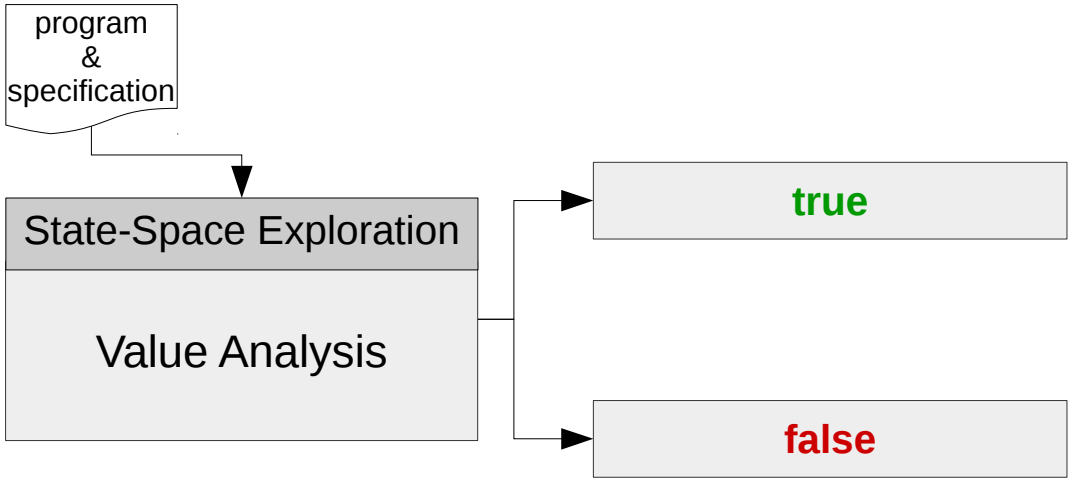
Inter-Analysis Refinement Selection — Most Effective

	BitVectors	ControlFlow	DD64	ECA	Floats	Loops	ProductLines	Sequential	Simple	Overall
total	48	48	2 120	1 140	81	141	597	62	46	4 283
PA	43	42	1 525	218	73	97	321	28	43	2 390
VA-PA-Composition	43	**45**	1 561	**484**	73	89	**390**	**38**	**46**	2 769
VA-PA-Composition–RefSel	**44**	**45**	**1652**	472	**74**	**98**	373	32	45	**2835**

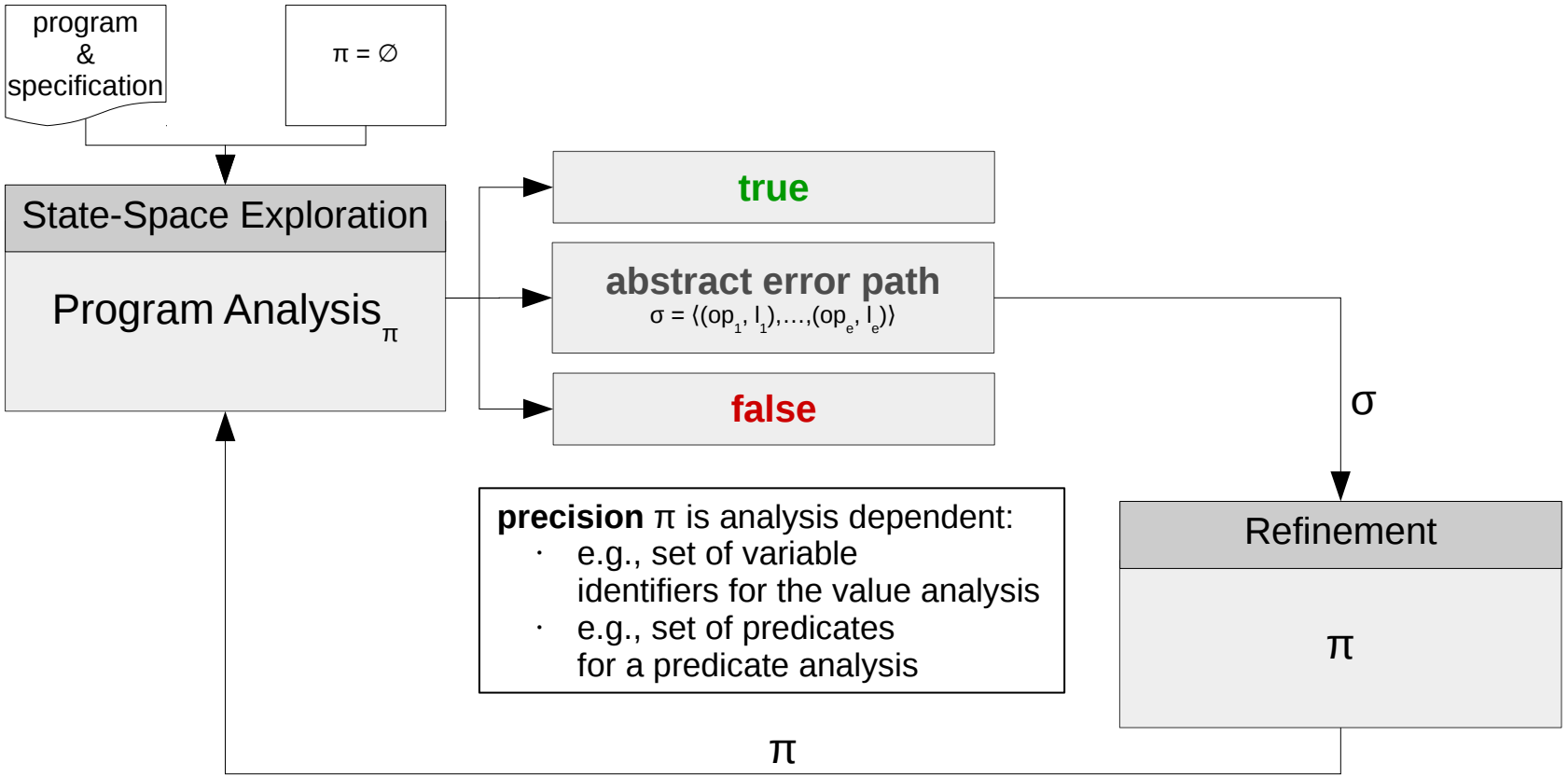
SV-COMP'16, Löwe

Verifier based on inter-analysis refinement selection wins gold medal in category "DeviceDrivers64"

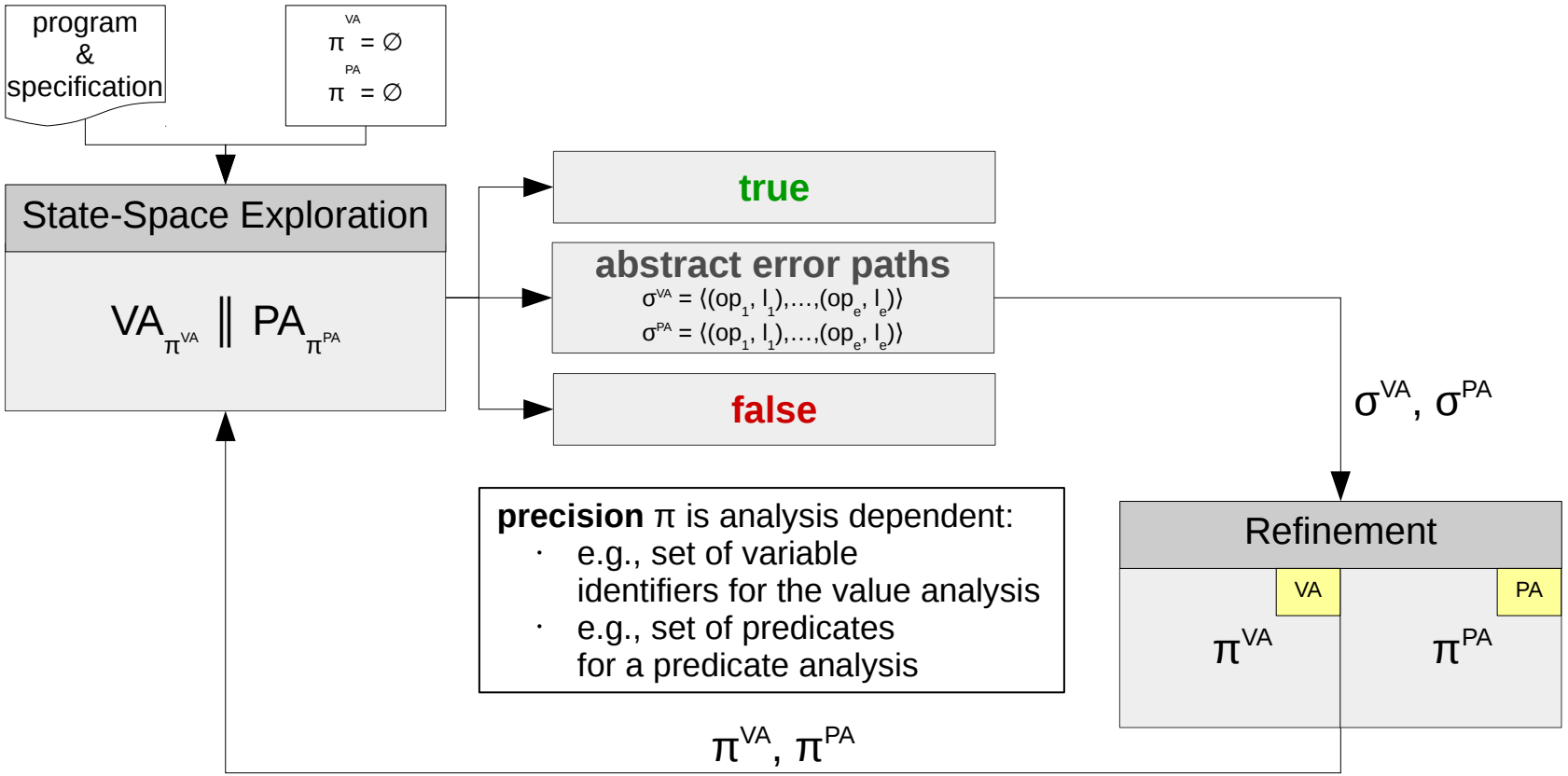
Summary: Value Analysis Plain and Simple



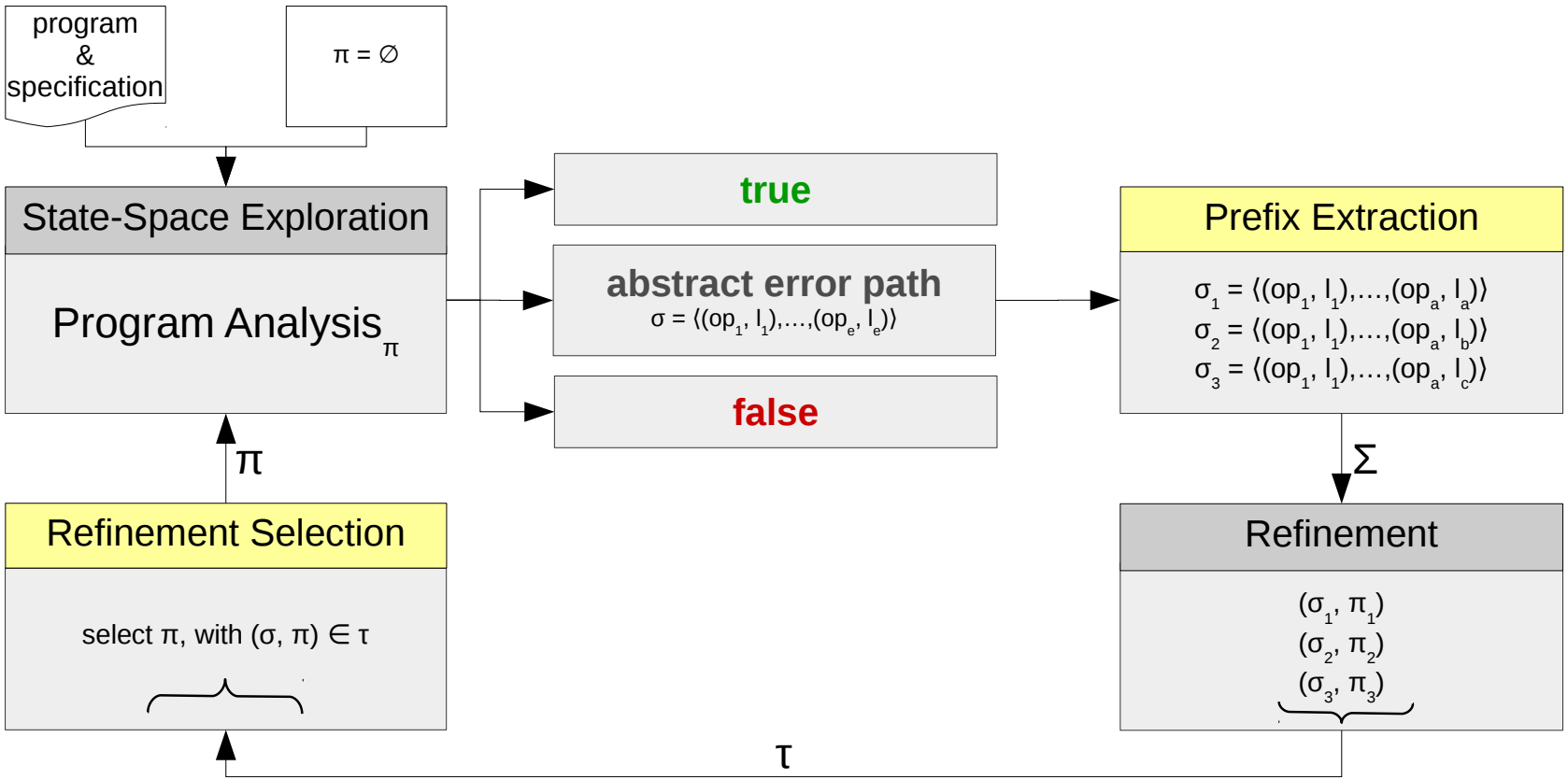
Summary: Value Analysis Based on CEGAR



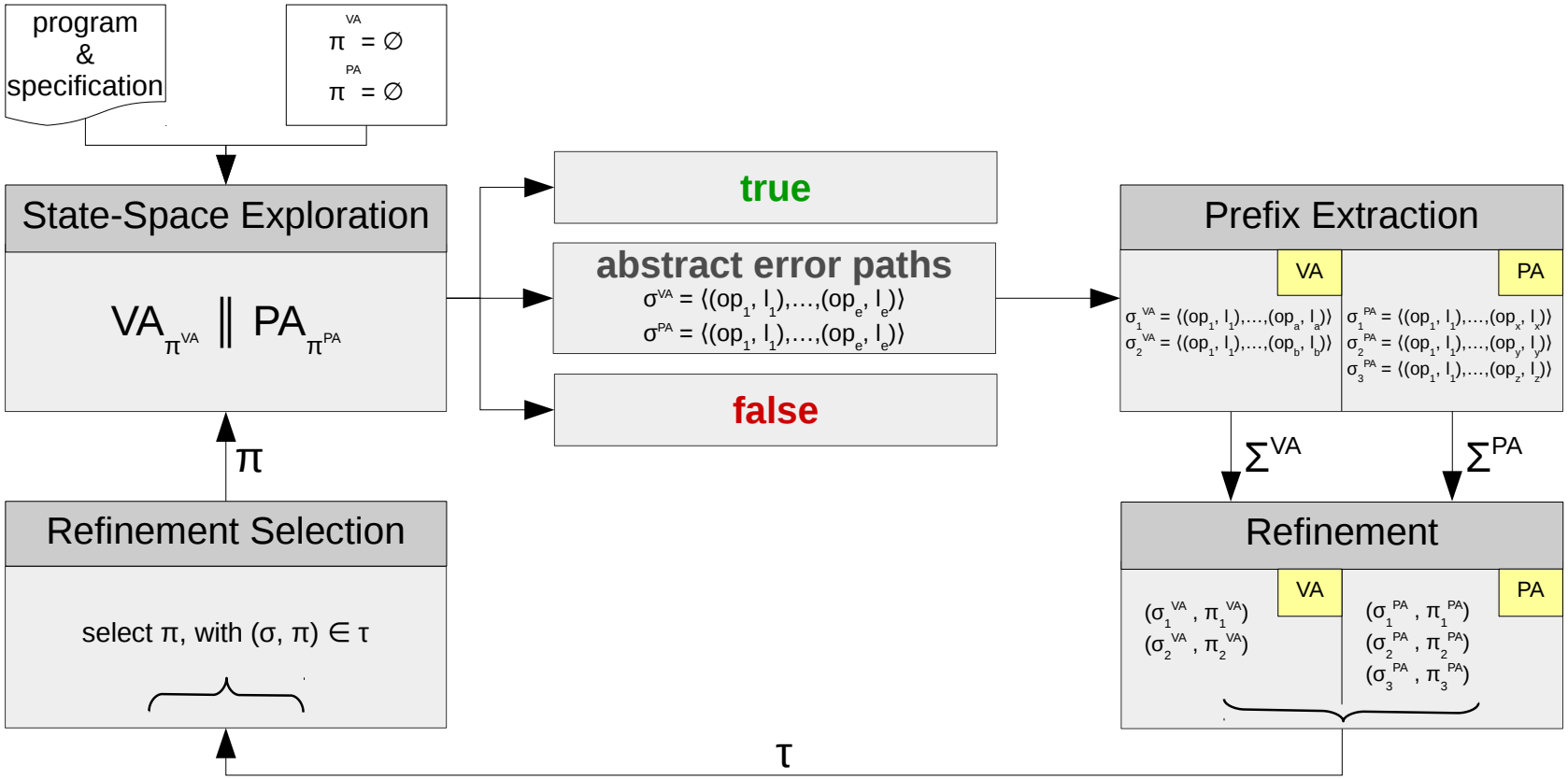
Summary: Composite Analysis Based on CEGAR



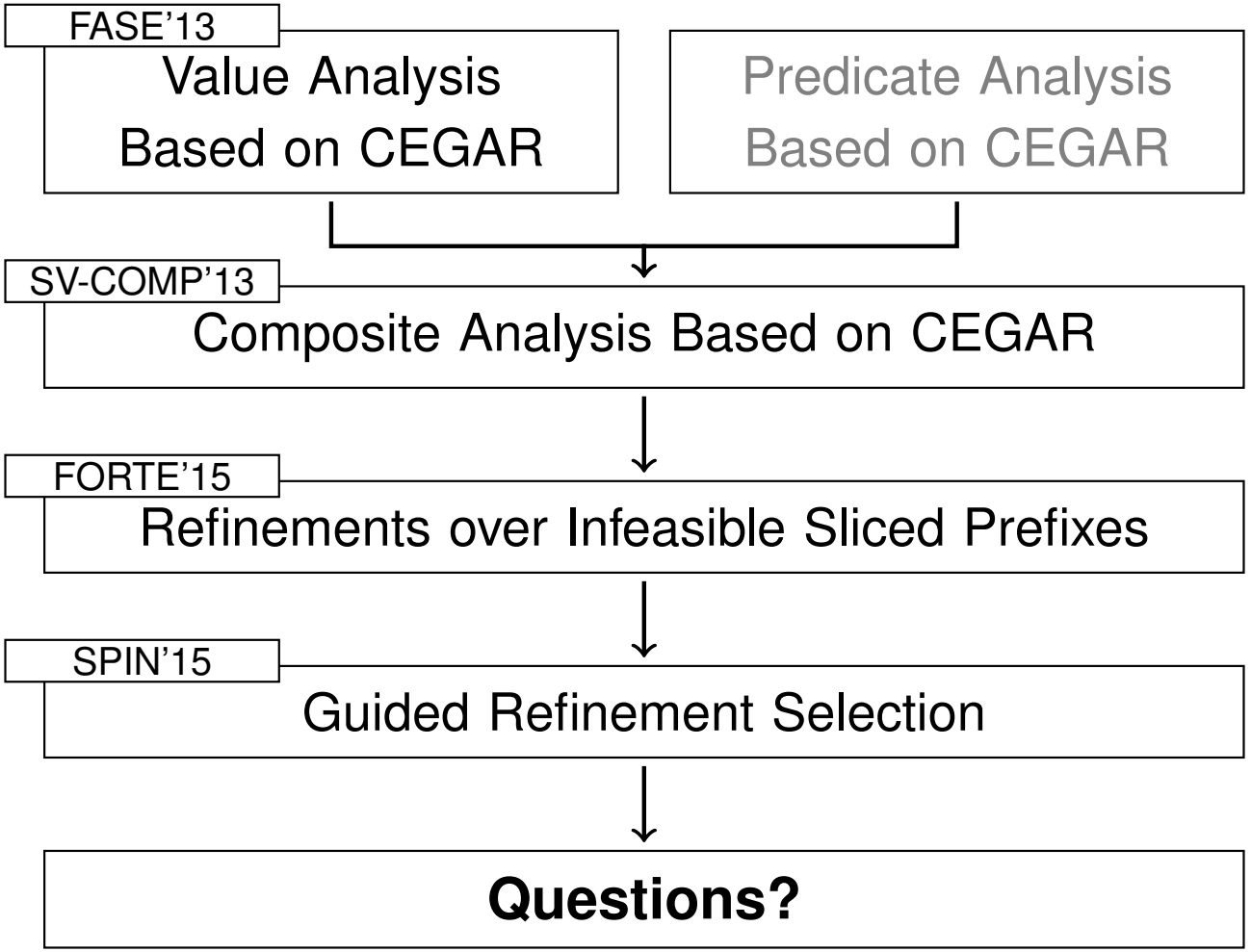
Summary: CEGAR with Guided Refinement Selection



Summary: CEGAR with Inter-Analysis Refinement Selection



The End


FASE'13
Value Analysis Based on CEGAR
Predicate Analysis Based on CEGAR
SV-COMP'13
Composite Analysis Based on CEGAR
FORTE'15
Refinements over Infeasible Sliced Prefixes
SPIN'15
Guided Refinement Selection
Questions?

Achievements

- Value Analysis Based on CEGAR (FASE'13)
- CPAchecker-Explicit (SV-COMP'13)
- Infeasible Sliced Prefixes (FORTE'15)
- Refinement Selection (SPIN'15)
- CPA-RefSel (SV-COMP'16)
- Precision Reuse (FSE'13)
- BenchExec (SPIN'15)
- Wins & Medals at SV-COMP (SV-COMP'12 – '16)
- Gödel-Medal (VSL'14)
- Linux Driver Verification (2012 — 2016)

Craig Interpolation

JSL'57, Craig

Theorem

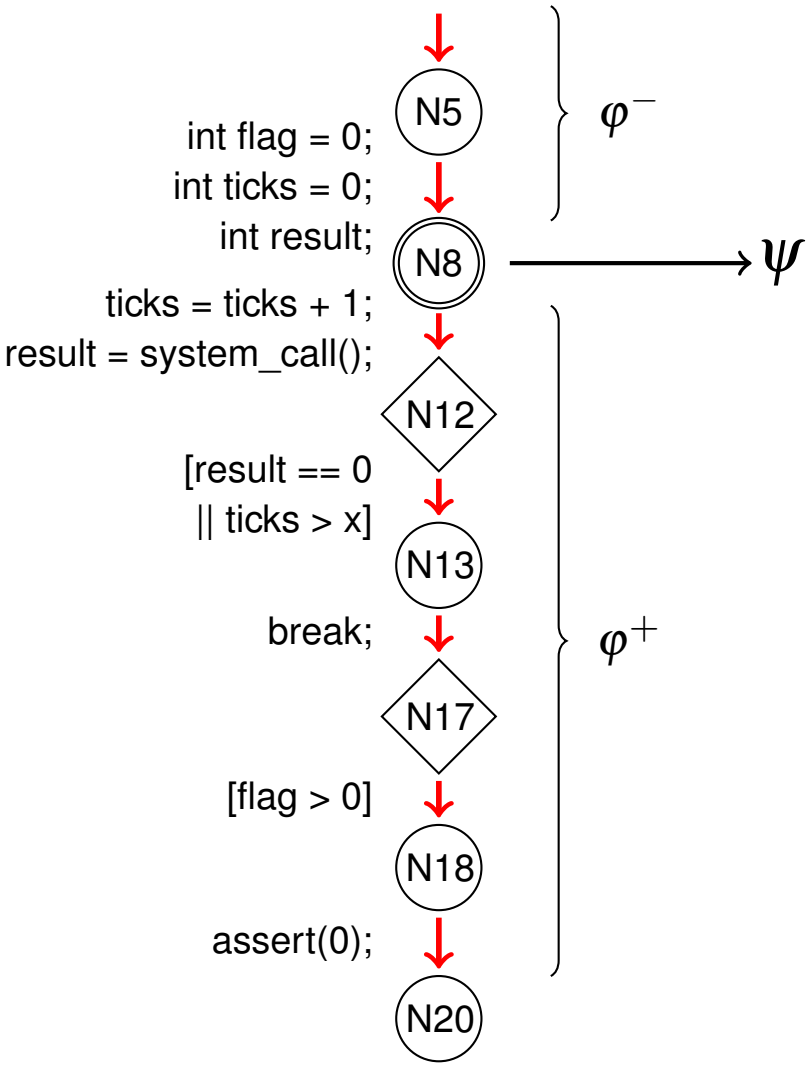
For a pair of formulas φ^- and φ^+ such that $\varphi^- \wedge \varphi^+$ is unsatisfiable, a Craig interpolant ψ is a formula that fulfills the following three requirements:

1. *the implication $\varphi^- \implies \psi$ holds,*
2. *the conjunction $\psi \wedge \varphi^+$ is unsatisfiable, and*
3. *ψ only contains symbols that occur in both φ^- and φ^+*

Such an interpolant is guaranteed to exist for many useful theories

Interpolation to Obtain Abstractions from Proofs

POPL'04, Henzinger, Jhala, Majumdar, McMillan

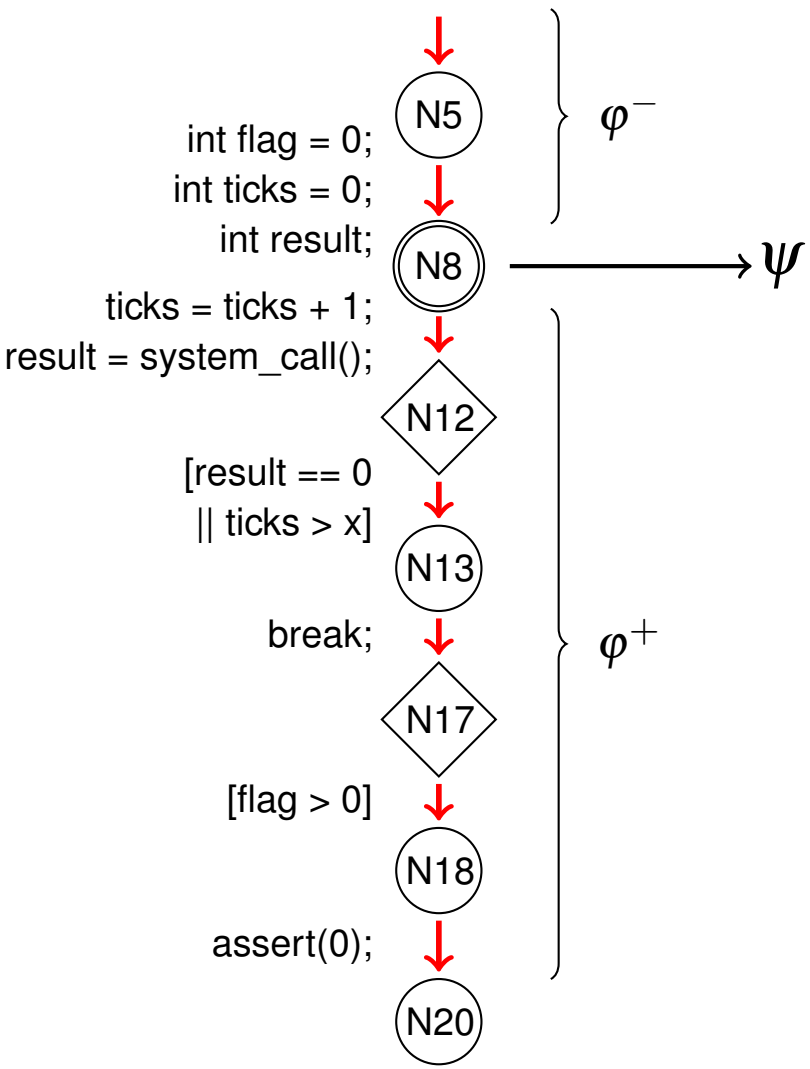


The interpolant ψ at N8 could be:

- $[\text{flag} = 0]$
- $[\text{flag} \leq 0]$
- $[\text{flag} < 1]$
- $[\text{flag} = 0 \wedge \text{ticks} = 0]$
- $[\text{flag} \leq 0 \wedge \text{ticks} = 0]$
- $[\text{flag} < 1 \wedge \text{ticks} = 0]$
- ...

Interpolation to Obtain Abstractions from Proofs

POPL'04, Henzinger, Jhala, Majumdar, McMillan



The interpolant ψ at N8 could be:

- [flag = 0]
- [flag $\leq$ 0]
- [flag $<$ 1]
- [flag = 0 $\wedge$ ticks = 0]
- [flag $\leq$ 0 $\wedge$ ticks = 0]
- [flag $<$ 1 $\wedge$ ticks = 0]
- ...

Value Interpolation does not Compute a Value Refinement

Problem

The Value Interpolation does not return a refined precision

Proposition

Use the Value Interpolation to compute a Value Refinement

Value Interpolation does not Compute a Value Refinement

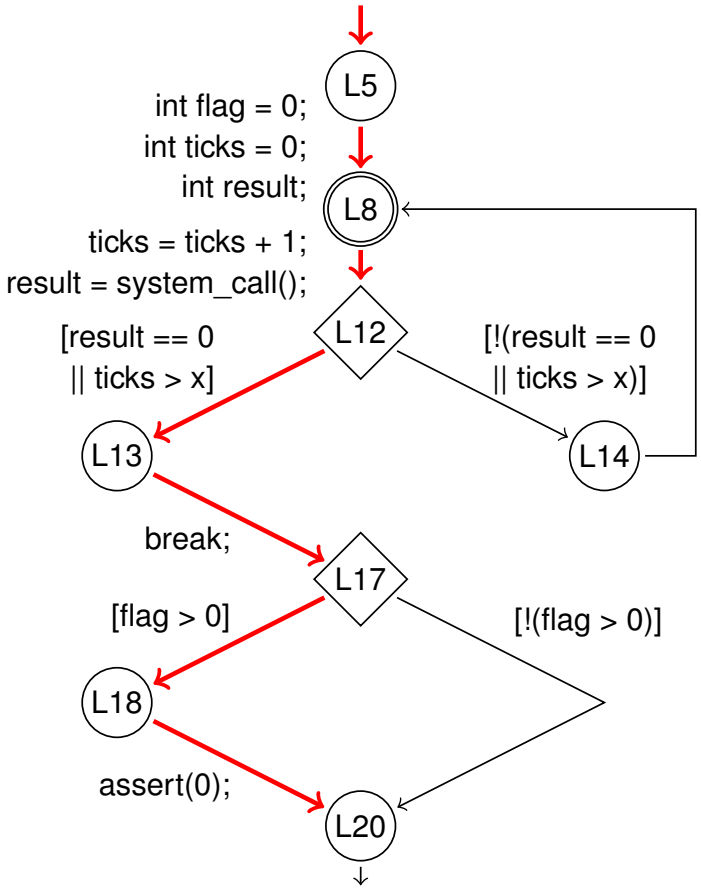
Problem

The Value Interpolation does not return a refined precision

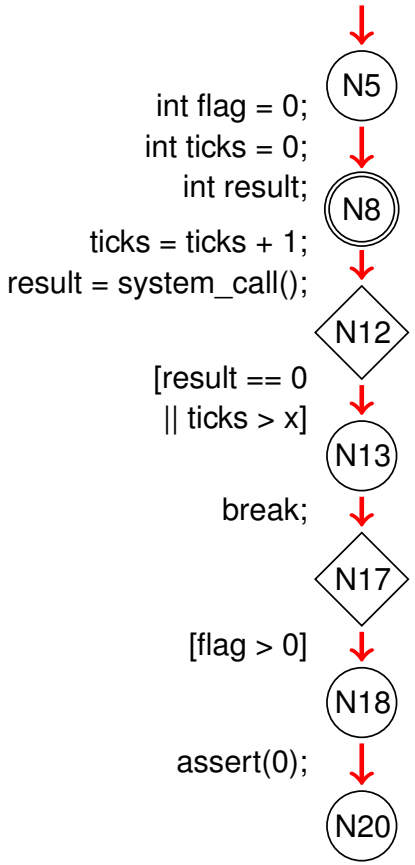
Proposition

Use the Value Interpolation to compute a Value Refinement

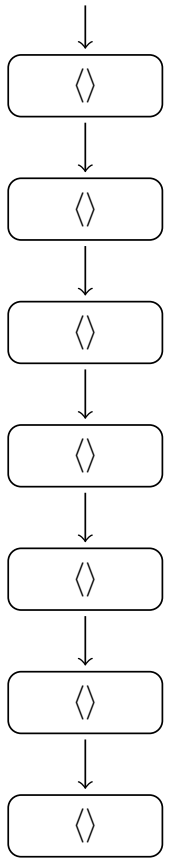
Value Refinement



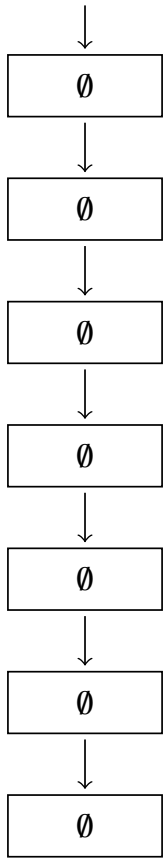
verification task



error path

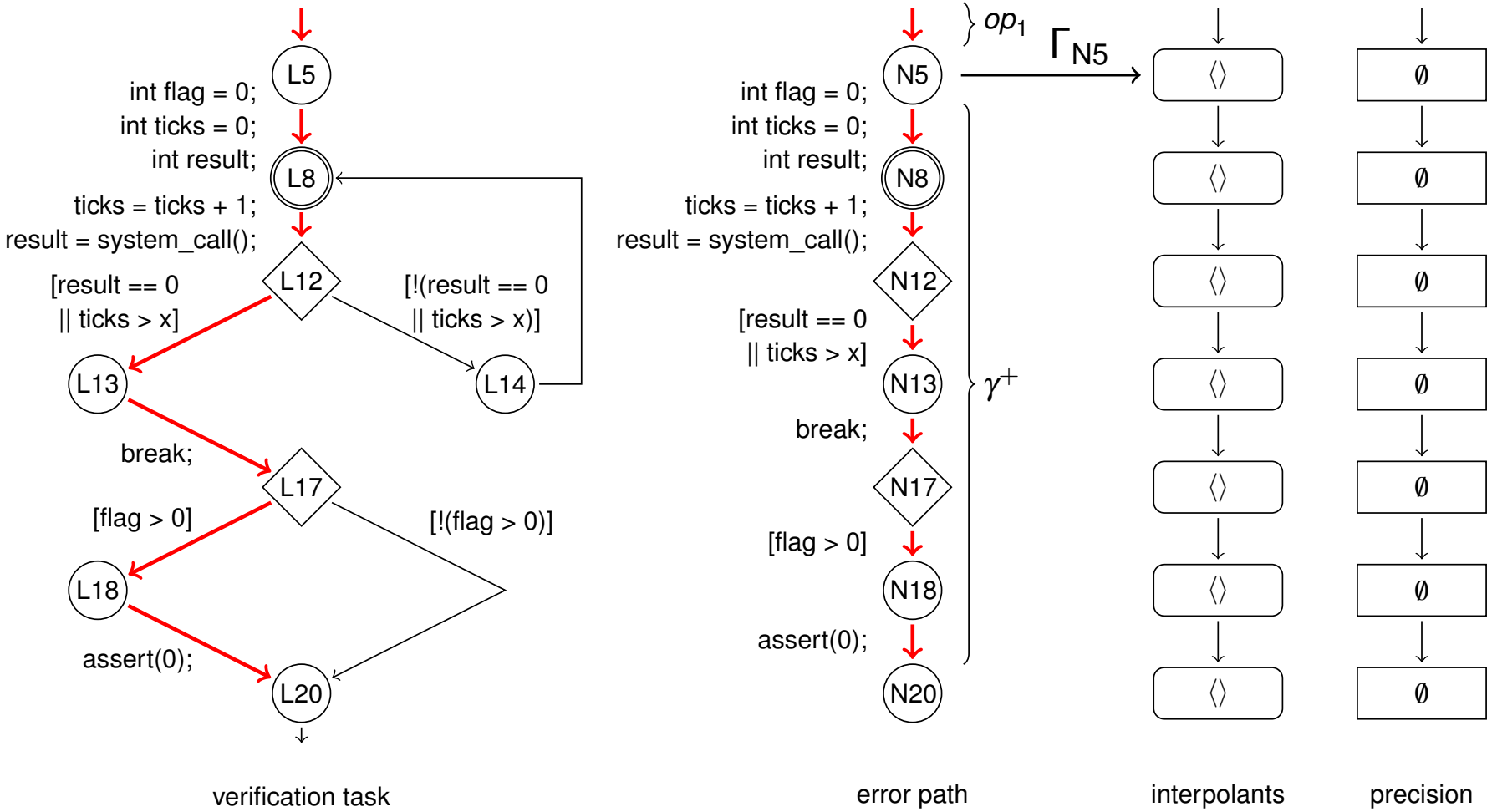


interpolants

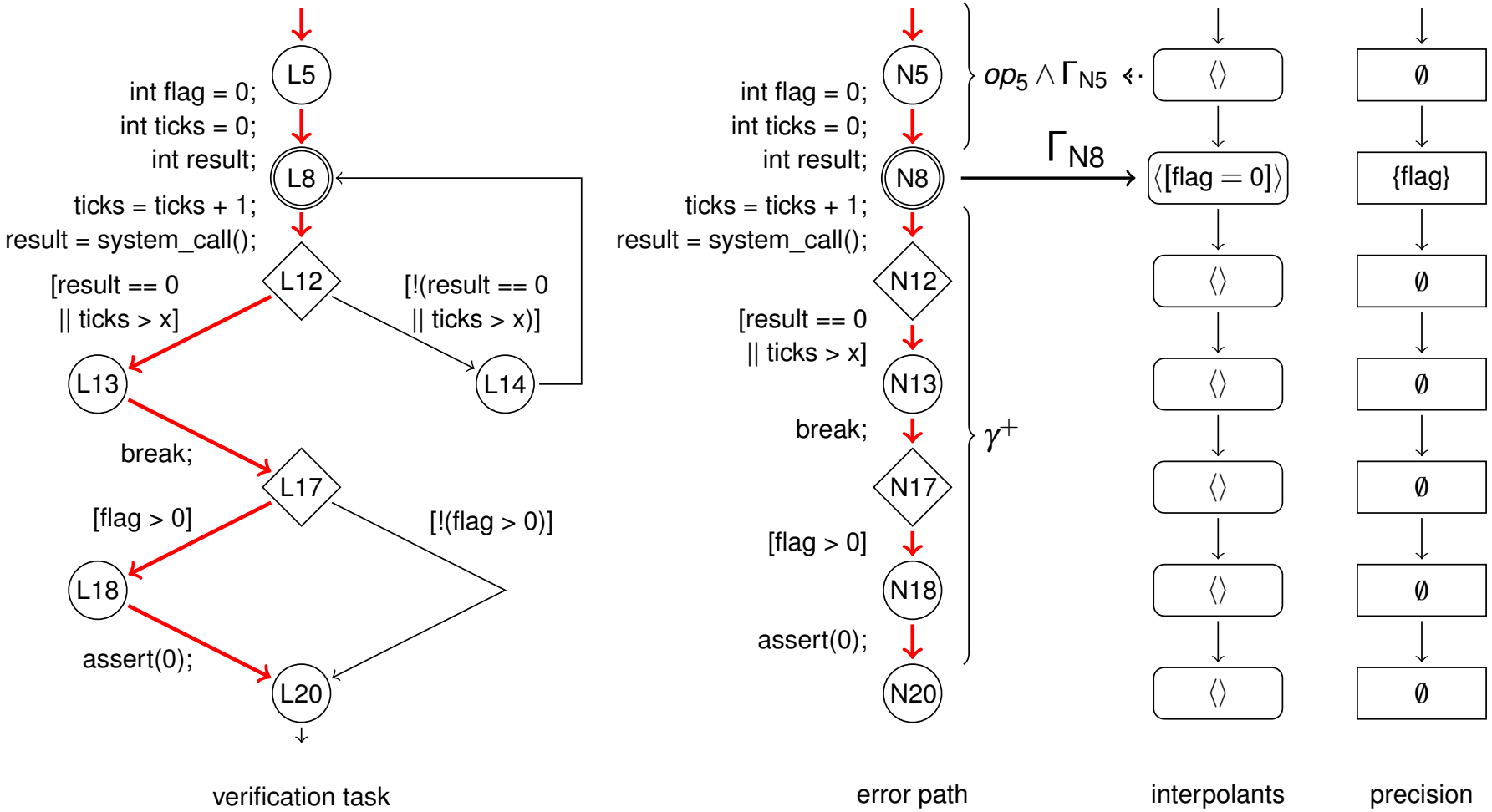


precision

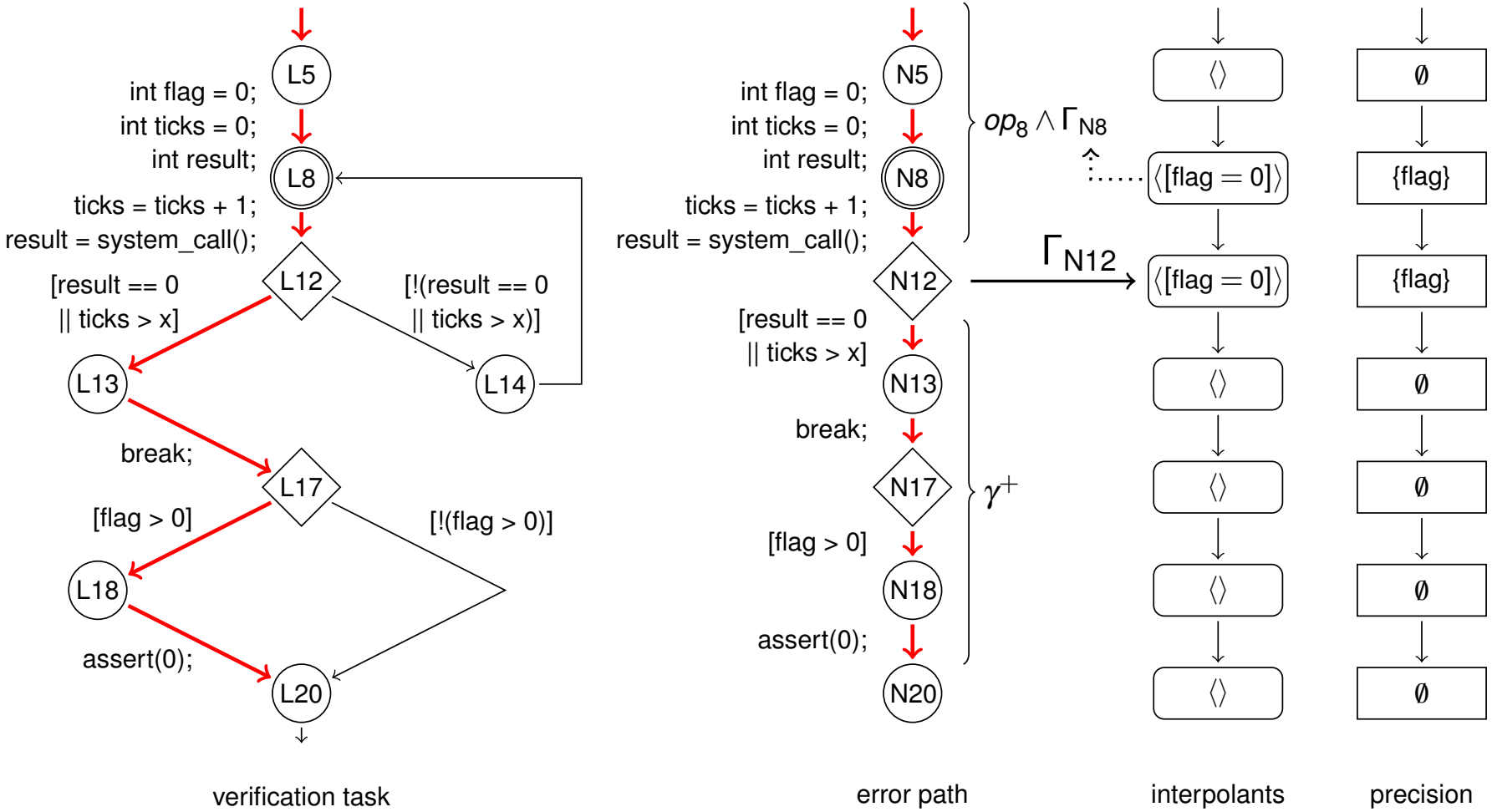
Value Refinement at Node 5



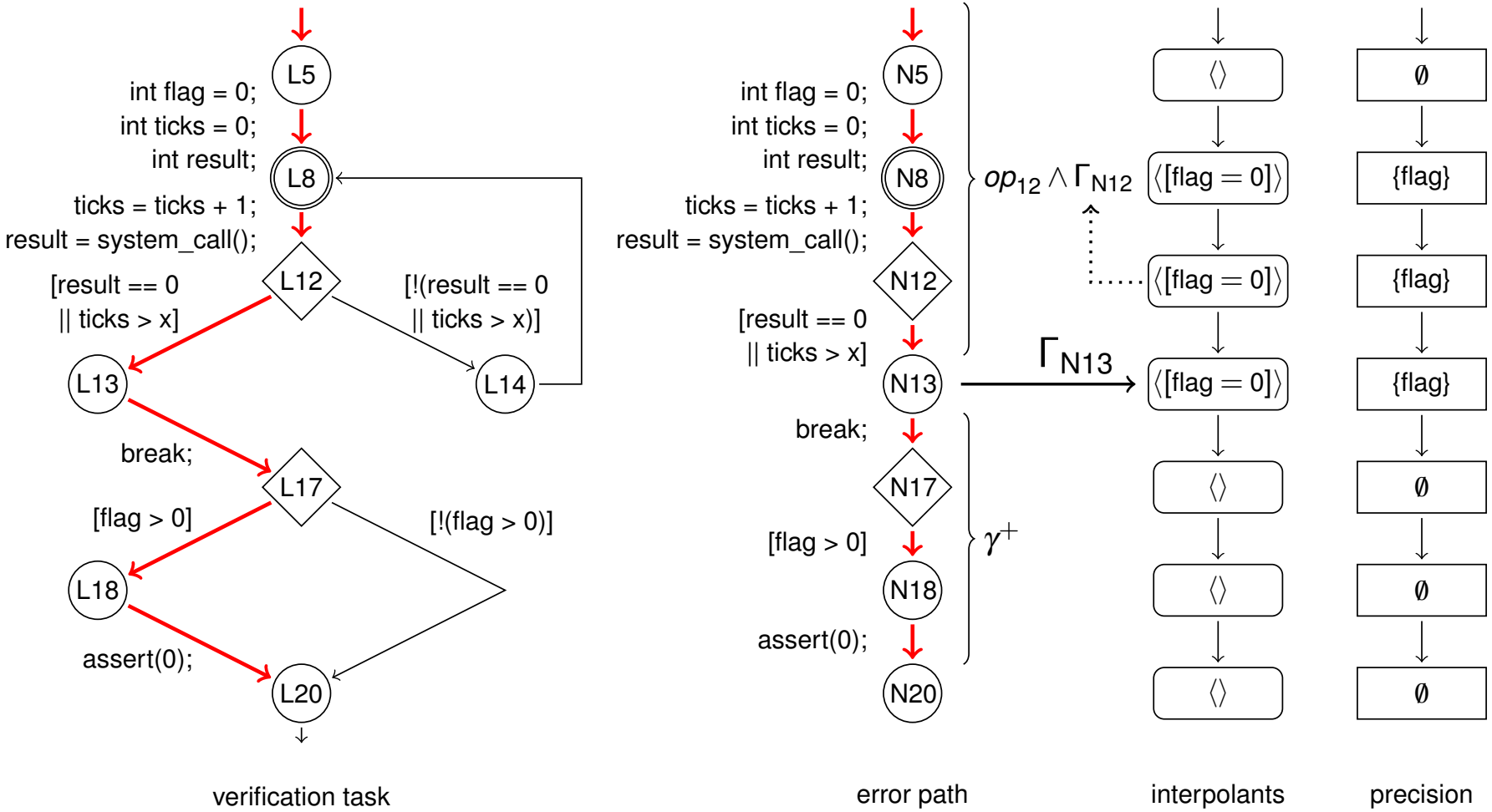
Value Refinement at Node 8



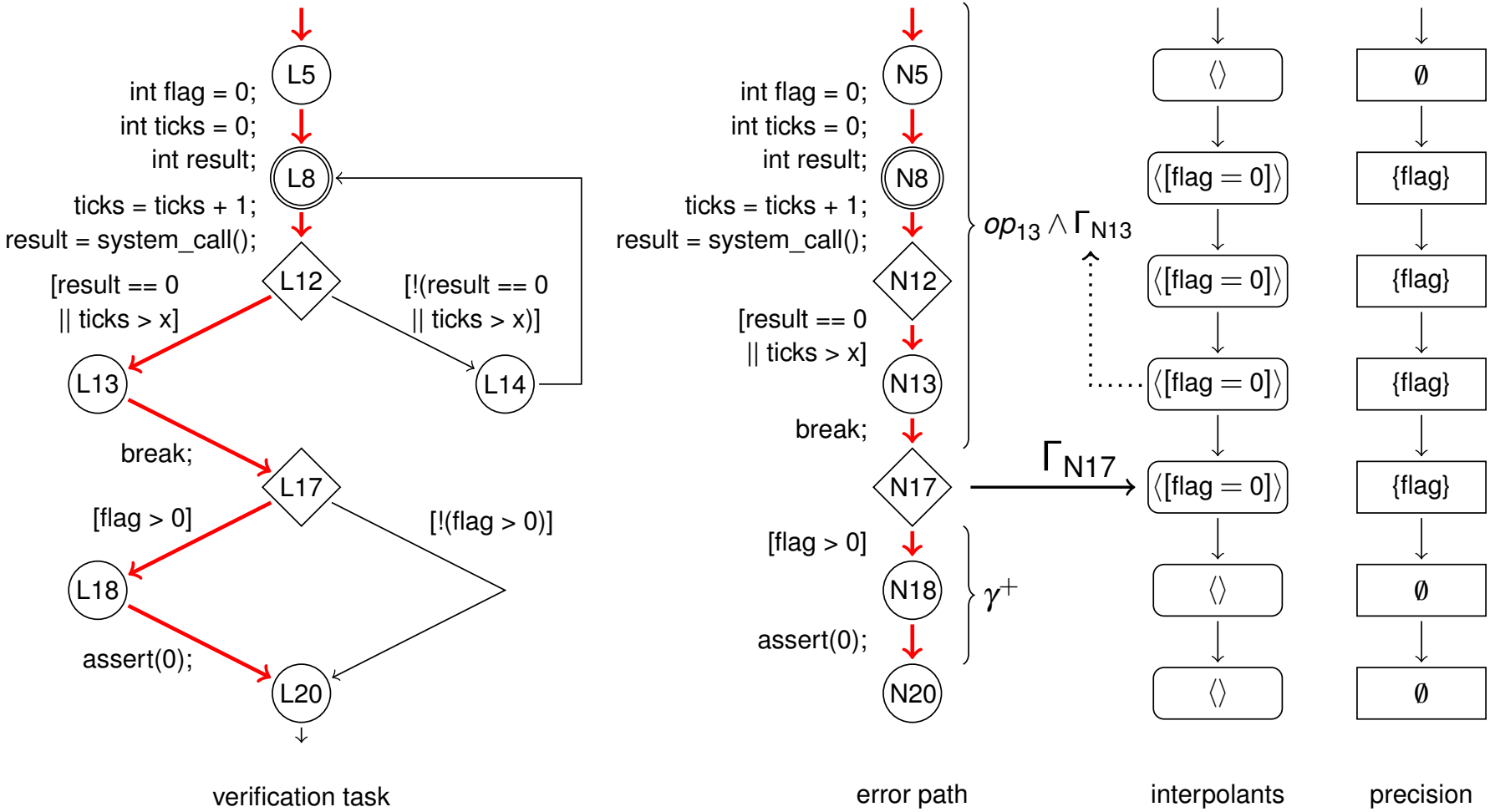
Value Refinement at Node 12



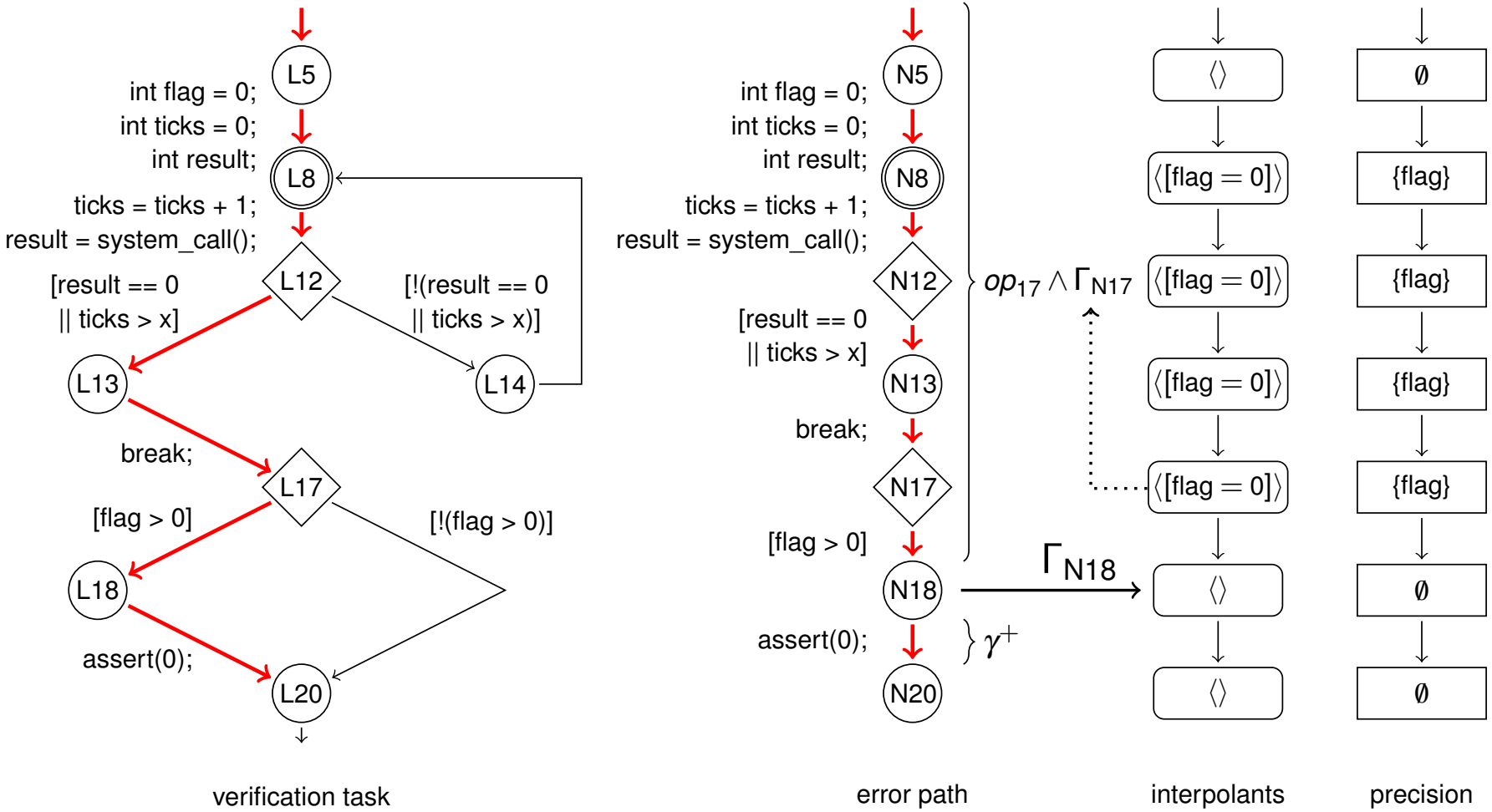
Value Refinement at Node 13



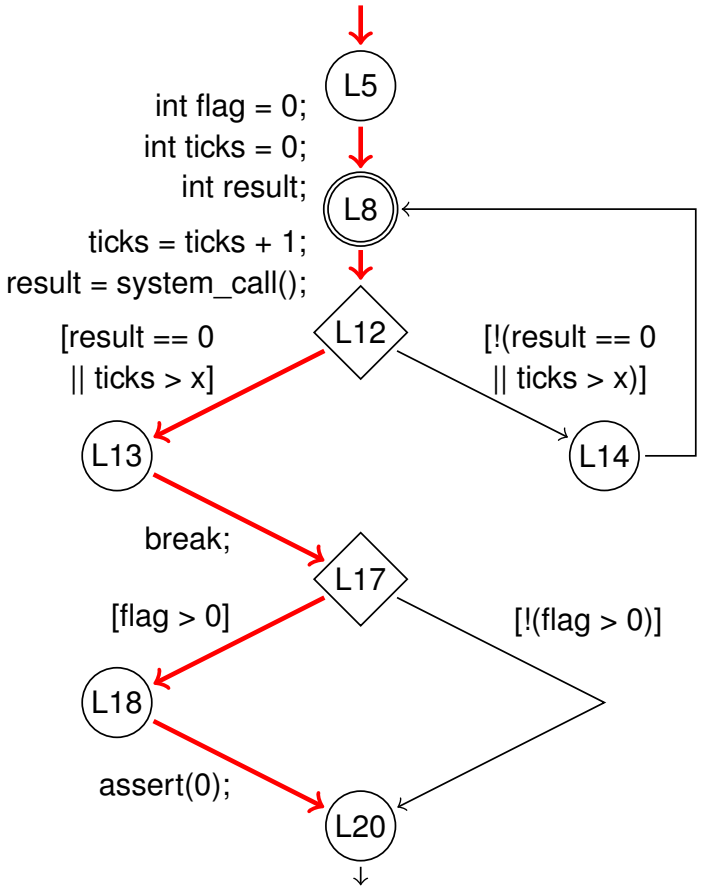
Value Refinement at Node 17



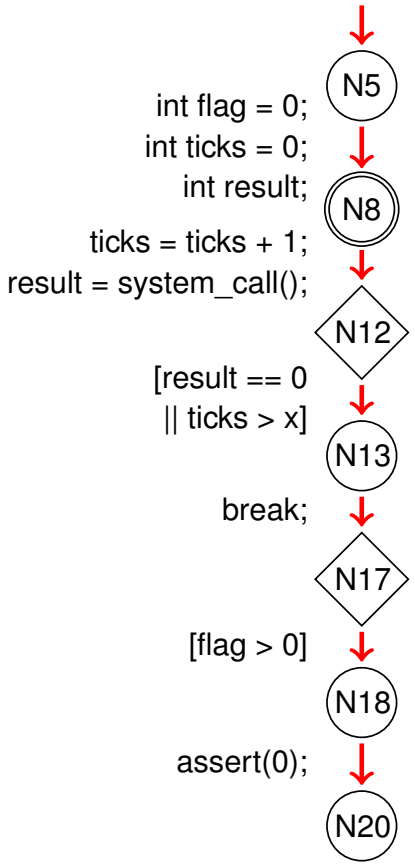
Value Refinement at Node 18



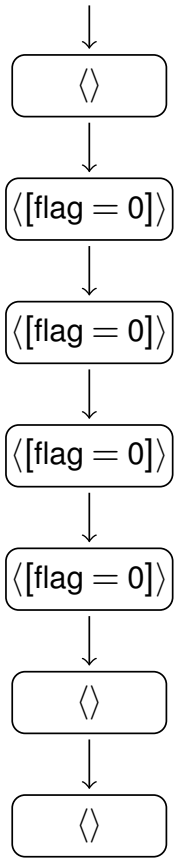
Value Refinement Completed



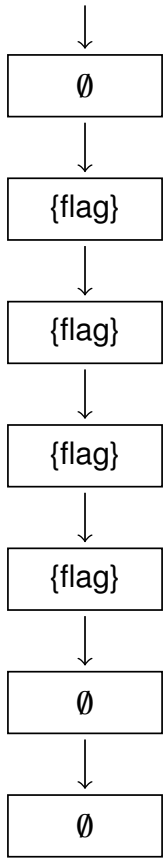
verification task



error path

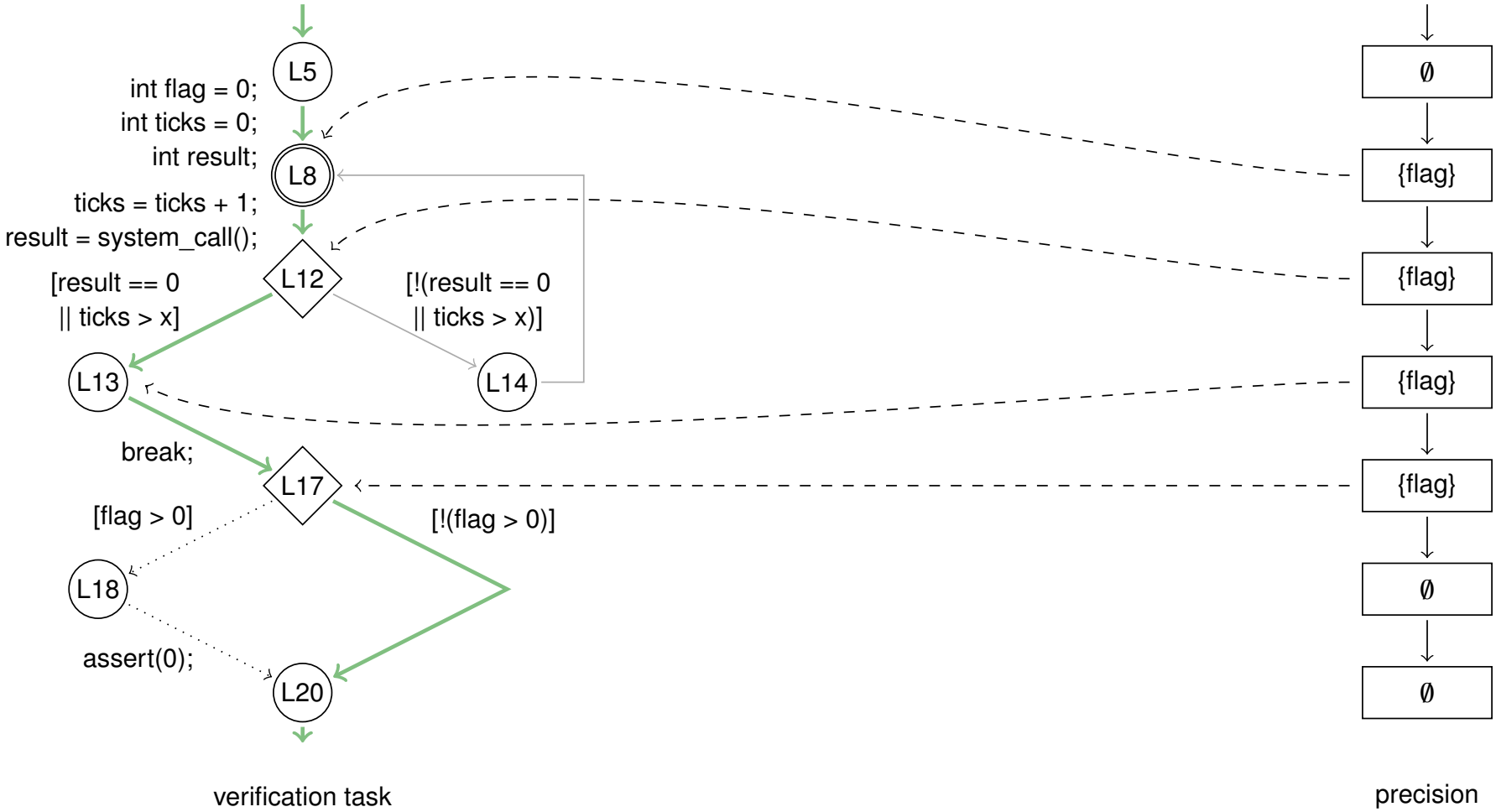


interpolants



precision

Value Refinement Avoids Counterexample and Loop



Refinement Selection by Other Heuristics

- Length of infeasible sliced path
- Combinations of heuristics
- Domain knowledge

Refinement Selection for Value Analysis — More Effective

		DD64	ECA	ProductLines	Sequential	Overall
total		2 120	1 140	597	62	4 283
Length	Min	1 646	489	449	37	2 951
	Max	**1764**	507	361	31	2 995
Width	Min	1 661	508	469	39	3 007
	Max	1 746	481	357	35	2 950
Depth	Min	1 724	530	388	**42**	3 014
	Max	1 665	519	448	38	3 002
Score	Min	**1764**	534	414	37	3 081
	Max	1 665	394	364	29	2 784
Width & Score		1 665	510	**472**	39	3 016
None		1 661	**575**	453	**42**	3 062
Optimal		1 764	575	472	42	3 186
Oracle		1 809	631	502	47	3 322

Refinement Selection for Value Analysis — More Efficient