



Leveraging Function Contracts for Blame Analysis

TUHH
Hamburg
University of
Technology

 **Universität
Bremen**


Carl von Ossietzky
**Universität
Oldenburg**

Nicola Thoben, Heike Wehrheim
Carl von Ossietzky Universität Oldenburg

Published at ISoLA 2026

Explanations of Misbehavior

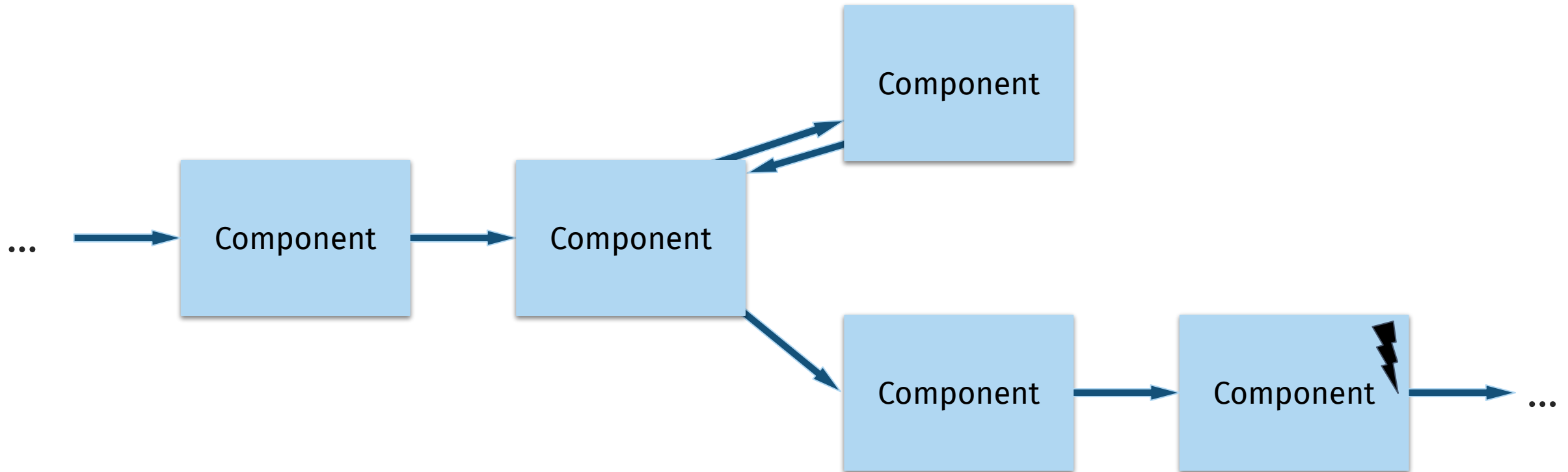
Overall goal:

Can software explain its behaviour in faulty situations?

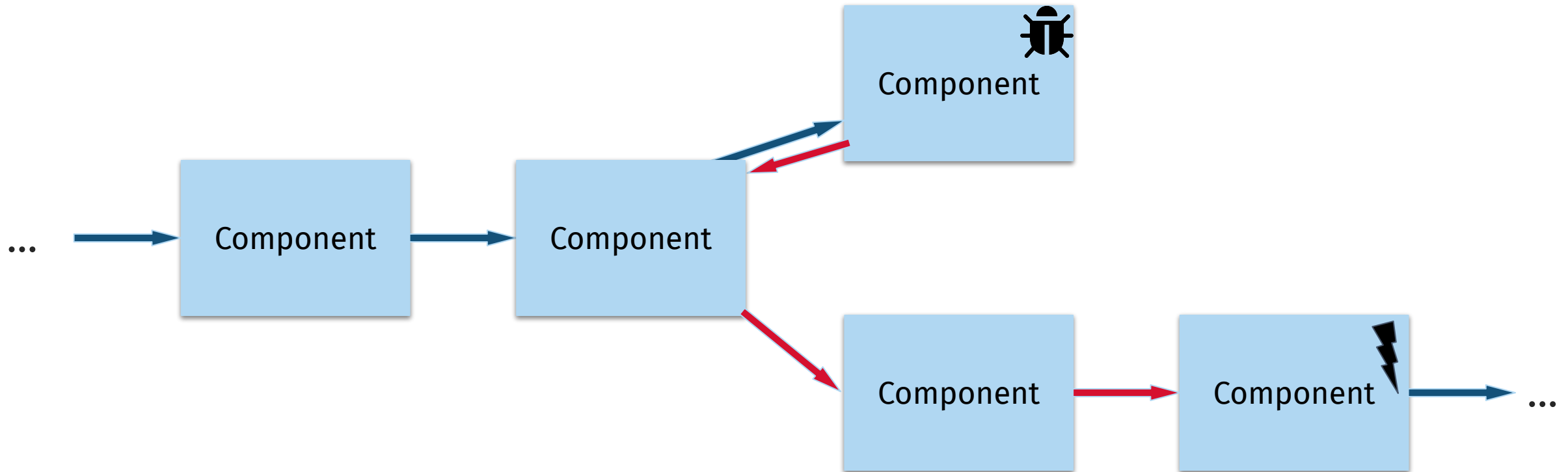
Today:

How can we locate **where** a failure originates?

Blame Analysis



Blame Analysis



Blame Analysis

- locate **where** a failure originates
- differentiate between triggering component and responsible component

Blame Analysis: Example

- locate **where** a failure originates
- differentiate between triggering component and responsible component

```
int cut(int pitch){
    if (pitch>90) {
        pitch = 90;
    }
    return pitch;
}

int calc(int x, int y){
    while (x>0) {
        x--; y--;
    }
    return y;
}

int main(int pitch, int error){
    int out_pitch = cut(pitch);
    int degree = calc(pitch, error);
    assert(out_pitch <= 90 &&
        degree == pitch - error);
    return 0;
}
```

Blame Analysis: Example

- locate **where** a failure originates
- differentiate between triggering component and responsible component

```
int cut(int pitch){
    if (pitch>90) {
        pitch = 90;
    }
    return pitch;
}

int calc(int x, int y){
    while (x>0) {
        x--; y--;
    }
    return y;
}

int main(int pitch, int error){
    int out_pitch = cut(pitch);
    int degree = calc(pitch, error);
    assert(out_pitch <= 90 &&
        degree == pitch - error);
    return 0;
}
```



Blame Analysis: Example

- locate **where** a failure originates
- differentiate between triggering component and responsible component
- use verification
- blame at **function level**

```
int cut(int pitch){
    if (pitch>90) {
        pitch = 90;
    }
    return pitch;
}

int calc(int x, int y){
    while (x>0) {
        x--; y--;
    }
    return y;
}

int main(int pitch, int error){
    int out_pitch = cut(pitch);
    int degree = calc(pitch, error);
    assert(out_pitch <= 90 &&
        degree == pitch - error);
    return 0;
}
```



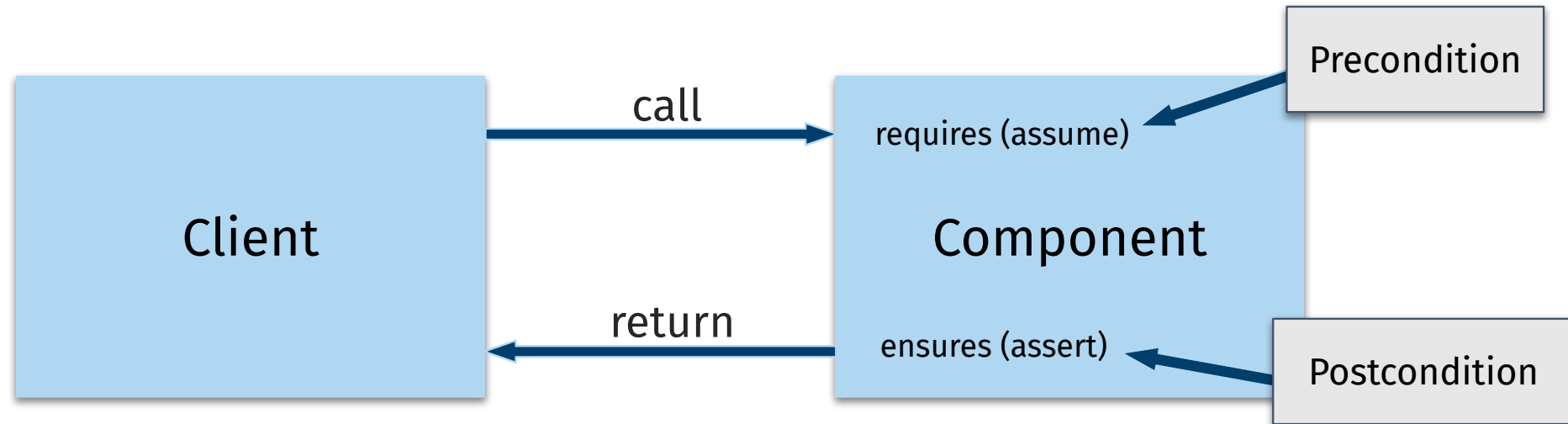
Using Contracts for Blame Analysis

- full program analysis can be expensive
- using verifier to obtain a counterexample can take a lot of time (and may lead to timeouts)

Idea:

- using contract information as abstraction for blame analysis

Design by Contract



ACSL

- specification language for C programs
- preconditions, postconditions and loop invariants for functions
- supports quantified properties over data structures

```
//@ requires x >= 0;  
//@ ensures \result == \old(y) - \old(x);  
int calc(int x, int y){  
...  
}
```

Folding

Function signature

`fn(arg1, ..., argn)`

```
//@ requires x>=0;  
//@ ensures \result == \old(y) - \old(x);  
int calc(int x, int y){  
    ...  
    return y;  
}
```

Function call

`type x = fn(x1, ..., xn)`

```
int degree = calc(pitch, error);
```

Contract replacement

```
type _tmpfn = nondet();  
if (pre[arg1/x1, ..., argn/xn])  
    assume(post[\result/_tmpfn,  
        \old(arg1)/x1, ..., \old(argn)/xn]);  
type x = _tmpfn;
```

```
int _tmpB;  
if (pitch >= 0){  
    assume(_tmpB == error - pitch);  
}  
int degree = _tmpB;
```

Folding: Example

```
//@ requires pitch>=0;
//@ ensures \result <= 90;
int cut(int pitch){
    if (pitch>90)
        pitch = 90;
    return pitch;
}

//@ requires x>=0;
//@ ensures \result == \old(y)-\old(x);
int calc(int x, int y){
    while (x>0) {
        x--; y--;
    }
    return y;
}

int main(int pitch, int error){
    assume(pitch>=0);
    int out_pitch = cut(pitch);
    int degree = calc(pitch, error);
    assert(out_pitch <= 90 &&
        degree == pitch - error);
    return 0;
}
```

Folding: Example

```
//@ requires pitch>=0;
//@ ensures \result <= 90;
int cut(int pitch){
    if (pitch>90)
        pitch = 90;
    return pitch;
}

//@ requires x>=0;
//@ ensures \result == \old(y)-\old(x);
int calc(int x, int y){
    while (x>0) {
        x--; y--;
    }
    return y;
}

int main(int pitch, int error){
    assume(pitch>=0);
    int out_pitch = cut(pitch);
    int degree = calc(pitch, error);
    assert(out_pitch <= 90 &&
        degree == pitch - error);
    return 0;
}
```



```
int main(int pitch, int error){
    assume(pitch >= 0);
    int _tmpA;
    if (pitch >= 0){
        assume(_tmpA <= 90);
    }
    int out_pitch = _tmpA;

    int _tmpB;
    if (pitch >= 0){
        assume(_tmpB == error - pitch);
    }
    int degree = _tmpB;

    assert(out_pitch <= 90 &&
        degree == pitch - error);
    return 0;
}
```

Folding: Example

```
//@ requires pitch>=0;
//@ ensures \result <= 90;
int cut(int pitch){
    if (pitch>90)
        pitch = 90;
    return pitch;
}

//@ requires x>=0;
//@ ensures \result == \old(y)-\old(x);
int calc(int x, int y){
    while (x>0) {
        x--; y--;
    }
    return y;
}

int main(int pitch, int error){
    assume(pitch>=0);
    int out_pitch = cut(pitch);
    int degree = calc(pitch, error);
    assert(out_pitch <= 90 &&
        degree == pitch - error);
    return 0;
}
```



```
int main(int pitch, int error){
    assume(pitch >= 0);
    int _tmpA;
    if (pitch >= 0){
        assume(_tmpA <= 90);
    }
    int out_pitch = _tmpA;

    int _tmpB;
    if (pitch >= 0){
        assume(_tmpB == error - pitch);
    }
    int degree = _tmpB;

    assert(out_pitch <= 90 &&
        degree == pitch - error);
    return 0;
}
```

Counterexample

```
int main(int pitch, int error){
    assume(pitch >= 0);
    int _tmpA;
    if (pitch >= 0){
        assume(_tmpA <= 90);
    }
    int out_pitch = _tmpA;

    int _tmpB;
    if (pitch >= 0){
        assume(_tmpB == error - pitch);
    }
    int degree = _tmpB;

    assert(out_pitch <= 90 &&
           degree == pitch - error);
    return 0;
}
```

$$\begin{aligned} &pitch^0 = 3 \wedge error^0 = 2 \\ &\wedge pitch^0 \geq 0 \\ &\wedge _tmpA^0 = 5 \\ &\wedge pitch^0 \geq 0 \wedge _tmpA^0 \leq 90 \\ &\wedge out_pitch^1 = _tmpA^0 \\ &\wedge _tmpB^0 = -1 \\ &\wedge pitch^0 \geq 0 \wedge _tmpB^0 = error^0 - pitch^0 \\ &\wedge degree^1 = _tmpB^0 \\ &\wedge \neg(out_pitch^1 \leq 90 \\ &\quad \wedge degree^1 = pitch^0 - error^0) \end{aligned}$$

Counterexample

```
int main(int pitch, int error){
    assume(pitch >= 0);
    int _tmpA;
    if (pitch >= 0){
        assume(_tmpA <= 90);
    }
    int out_pitch = _tmpA;

    int _tmpB;
    if (pitch >= 0){
        assume(_tmpB == error - pitch);
    }
    int degree = _tmpB;

    assert(out_pitch <= 90 &&
        degree == pitch - error);
    return 0;
}
```

$$pitch^0 = 3 \wedge error^0 = 2$$

$$\wedge pitch^0 \geq 0$$

$$\wedge _tmpA^0 = 5$$

$$\wedge pitch^0 \geq 0 \wedge _tmpA^0 \leq 90$$

$$\wedge out_pitch^1 = _tmpA^0$$

$$\wedge _tmpB^0 = -1$$

$$\wedge pitch^0 \geq 0 \wedge _tmpB^0 = error^0 - pitch^0$$

$$\wedge degree^1 = _tmpB^0$$

$$\wedge \neg(out_pitch^1 \leq 90$$

$$\wedge degree^1 = pitch^0 - error^0)$$

Inputs

Counterexample

```
int main(int pitch, int error){
    assume(pitch >= 0);
    int _tmpA;
    if (pitch >= 0){
        assume(_tmpA <= 90);
    }
    int out_pitch = _tmpA;

    int _tmpB;
    if (pitch >= 0){
        assume(_tmpB == error - pitch);
    }
    int degree = _tmpB;

    assert(out_pitch <= 90 &&
           degree == pitch - error);
    return 0;
}
```

$$pitch^0 = 3 \wedge error^0 = 2$$

$$\wedge pitch^0 \geq 0$$

$$\wedge _tmpA^0 = 5$$

$$\wedge pitch^0 \geq 0 \wedge _tmpA^0 \leq 90$$

$$\wedge out_pitch^1 = _tmpA^0$$

$$\wedge _tmpB^0 = -1$$

$$\wedge pitch^0 \geq 0 \wedge _tmpB^0 = error^0 - pitch^0$$

$$\wedge degree^1 = _tmpB^0$$

$$\wedge \neg(out_pitch^1 \leq 90$$

$$\wedge degree^1 = pitch^0 - error^0)$$

Inputs

Precondition

Counterexample

```
int main(int pitch, int error){
  assume(pitch >= 0);
  int _tmpA;
  if (pitch >= 0){
    assume(_tmpA <= 90);
  }
  int out_pitch = _tmpA;

  int _tmpB;
  if (pitch >= 0){
    assume(_tmpB == error - pitch);
  }
  int degree = _tmpB;

  assert(out_pitch <= 90 &&
    degree == pitch - error);
  return 0;
}
```

$$pitch^0 = 3 \wedge error^0 = 2$$

$$\wedge pitch^0 \geq 0$$

$$\wedge _tmpA^0 = 5$$

$$\wedge pitch^0 \geq 0 \wedge _tmpA^0 \leq 90$$

$$\wedge out_pitch^1 = _tmpA^0$$

$$\wedge _tmpB^0 = -1$$

$$\wedge pitch^0 \geq 0 \wedge _tmpB^0 = error^0 - pitch^0$$

$$\wedge degree^1 = _tmpB^0$$

$$\wedge \neg(out_pitch^1 \leq 90$$

$$\wedge degree^1 = pitch^0 - error^0)$$

Inputs

Precondition

Contracts

Counterexample

```
int main(int pitch, int error){
    assume(pitch >= 0);
    int _tmpA;
    if (pitch >= 0){
        assume(_tmpA <= 90);
    }
    int out_pitch = _tmpA;

    int _tmpB;
    if (pitch >= 0){
        assume(_tmpB == error - pitch);
    }
    int degree = _tmpB;

    assert(out_pitch <= 90 &&
           degree == pitch - error);
    return 0;
}
```

$$pitch^0 = 3 \wedge error^0 = 2$$

$$\wedge pitch^0 \geq 0$$

$$\wedge _tmpA^0 = 5$$

$$\wedge pitch^0 \geq 0 \wedge _tmpA^0 \leq 90$$

$$\wedge out_pitch^1 = _tmpA^0$$

$$\wedge _tmpB^0 = -1$$

$$\wedge pitch^0 \geq 0 \wedge _tmpB^0 = error^0 - pitch^0$$

$$\wedge degree^1 = _tmpB^0$$

$$\wedge \neg(out_pitch^1 \leq 90$$

$$\wedge degree^1 = pitch^0 - error^0)$$

Inputs

Precondition

Contracts

Postcondition

Counterexample

```
int main(int pitch, int error){
    assume(pitch >= 0);
    int _tmpA;
    if (pitch >= 0){
        assume(_tmpA <= 90);
    }
    int out_pitch = _tmpA;

    int _tmpB;
    if (pitch >= 0){
        assume(_tmpB == error - pitch);
    }
    int degree = _tmpB;

    assert(out_pitch <= 90 &&
           degree == pitch - error);
    return 0;
}
```

$$\begin{aligned} & pitch^0 = 3 \wedge error^0 = 2 \\ & \wedge pitch^0 \geq 0 \\ & \wedge _tmpA^0 = 5 \\ & \wedge pitch^0 \geq 0 \wedge _tmpA^0 \leq 90 \\ & \wedge out_pitch^1 = _tmpA^0 \\ & \wedge _tmpB^0 = -1 \\ & \wedge pitch^0 \geq 0 \wedge _tmpB^0 = error^0 - pitch^0 \\ & \wedge degree^1 = _tmpB^0 \\ & \wedge \neg(out_pitch^1 \leq 90 \\ & \quad \wedge degree^1 = pitch^0 - error^0) \end{aligned}$$

SAT

Counterexample

```
int main(int pitch, int error){
    assume(pitch >= 0);
    int _tmpA;
    if (pitch >= 0){
        assume(_tmpA <= 90);
    }
    int out_pitch = _tmpA;

    int _tmpB;
    if (pitch >= 0){
        assume(_tmpB == error - pitch);
    }
    int degree = _tmpB;

    assert(out_pitch <= 90 &&
           degree == pitch - error);
    return 0;
}
```

$$\begin{aligned} & pitch^0 = 3 \wedge error^0 = 2 \\ & \wedge pitch^0 \geq 0 \\ & \wedge _tmpA^0 = 5 \\ & \wedge pitch^0 \geq 0 \wedge _tmpA^0 \leq 90 \\ & \wedge out_pitch^1 = _tmpA^0 \\ & \wedge _tmpB^0 = -1 \\ & \wedge pitch^0 \geq 0 \wedge _tmpB^0 = error^0 - pitch^0 \\ & \wedge degree^1 = _tmpB^0 \\ & \wedge \underline{(out_pitch^1 \leq 90} \\ & \quad \underline{\wedge degree^1 = pitch^0 - error^0)} \end{aligned}$$

UNSAT

Weighted MaxSMT

- starts from SSA-form counterexample
- constraints are **hard or soft**
- MaxSMT keeps **as many constraints as possible**
- weight puts focus on contracts
- removed constraints indicate blame candidates

$$\begin{aligned} & pitch^0 = 3 \wedge error^0 = 2 \\ & \wedge pitch^0 \geq 0 \\ & \wedge _tmpA^0 = 5 \\ & \wedge pitch^0 \geq 0 \wedge _tmpA^0 \leq 90 \\ & \wedge out_pitch^1 = _tmpA^0 \\ & \wedge _tmpB^0 = -1 \\ & \wedge pitch^0 \geq 0 \wedge _tmpB^0 = error^0 - pitch^0 \\ & \wedge degree^1 = _tmpB^0 \\ & \wedge (out_pitch^1 \leq 90 \\ & \quad \wedge degree^1 = pitch^0 - error^0) \end{aligned}$$

Weighted MaxSMT

- starts from SSA-form counterexample
- constraints are **hard or soft**
- MaxSMT keeps **as many constraints as possible**
- weight puts focus on contracts
- removed constraints indicate blame candidates

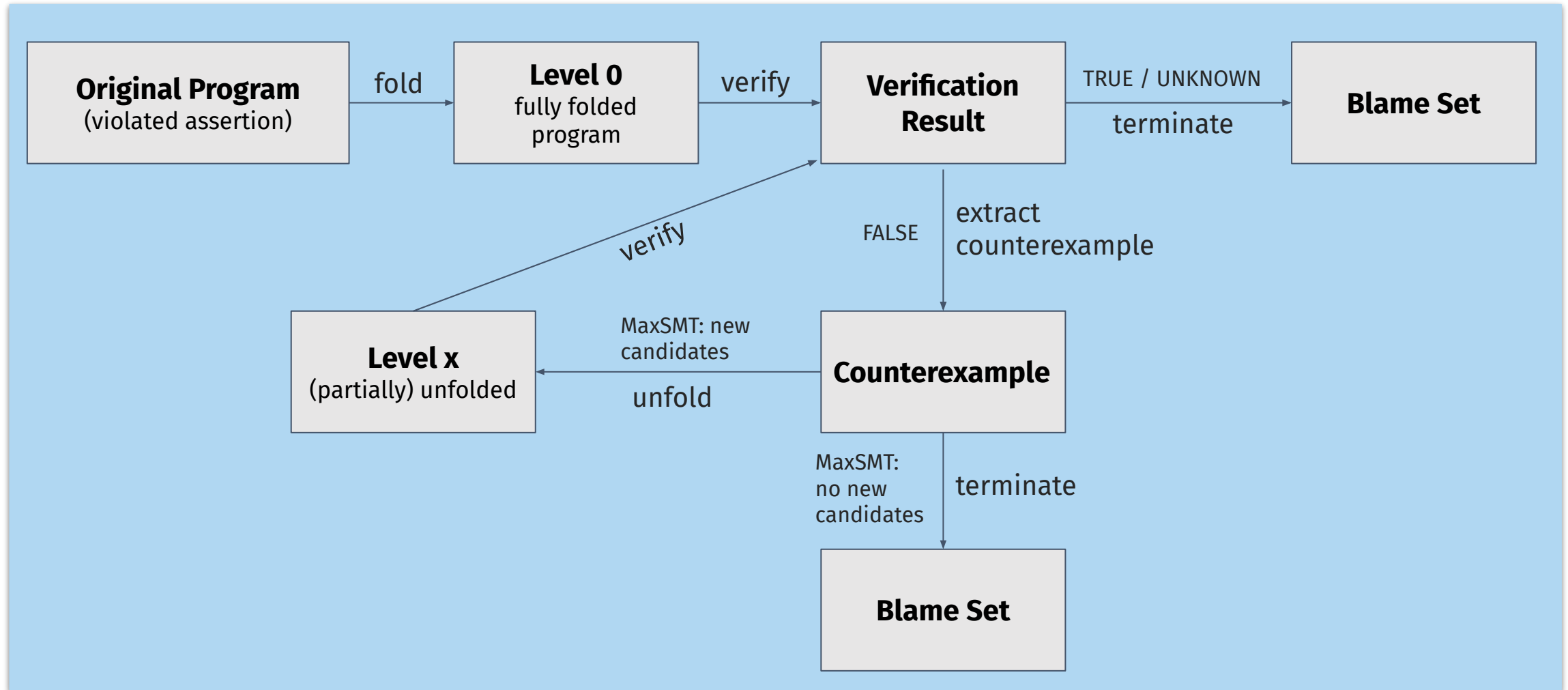
$$\begin{aligned} & pitch^0 = 3 \wedge error^0 = 2 \\ & \wedge pitch^0 \geq 0 \\ & \wedge _tmpA^0 = 5 \\ & \wedge pitch^0 \geq 0 \wedge _tmpA^0 \leq 90 \\ & \wedge out_pitch^1 = _tmpA^0 \\ & \wedge _tmpB^0 = -1 \\ & \wedge pitch^0 \geq 0 \wedge _tmpB^0 = error^0 - pitch^0 \\ & \wedge degree^1 = _tmpB^0 \\ & \wedge (out_pitch^1 \leq 90 \\ & \quad \wedge degree^1 = pitch^0 - error^0) \end{aligned}$$

Weighted MaxSMT

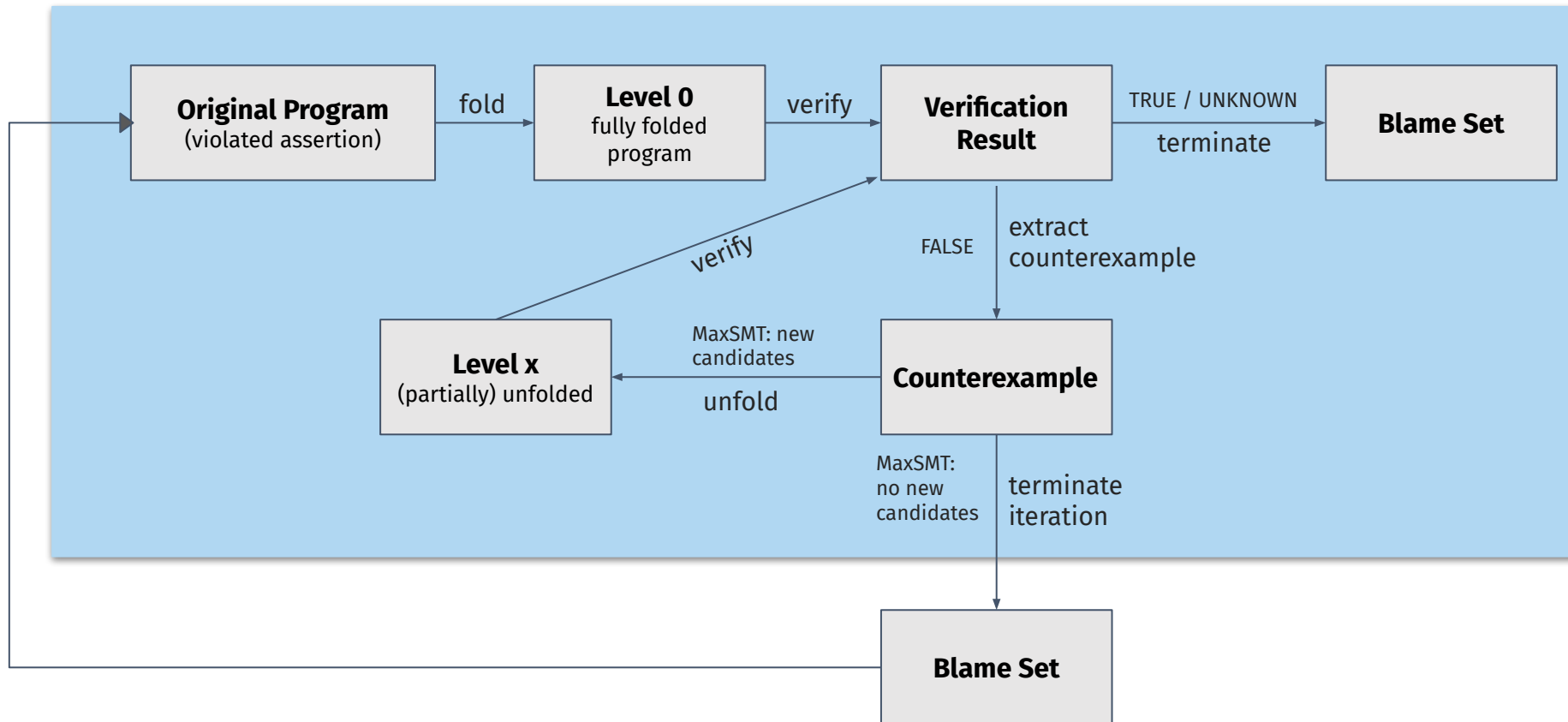
- starts from SSA-form counterexample
- constraints are **hard or soft**
- MaxSMT keeps **as many constraints as possible**
- weight puts focus on contracts
- removed constraints indicate blame candidates

$$\begin{aligned} & pitch^0 = 3 \wedge error^0 = 2 \\ & \wedge pitch^0 \geq 0 \\ & \wedge _tmpA^0 = 5 \\ & \wedge pitch^0 \geq 0 \wedge _tmpA^0 \leq 90 \\ & \wedge out_pitch^1 = _tmpA^0 \\ & \wedge \textcolor{red}{_tmpB^0 = -1} \\ & \wedge pitch^0 \geq 0 \wedge \textcolor{red}{_tmpB^0 = error^0 - pitch^0} \\ & \wedge degree^1 = _tmpB^0 \\ & \wedge (out_pitch^1 \leq 90 \\ & \quad \wedge degree^1 = pitch^0 - error^0) \end{aligned}$$

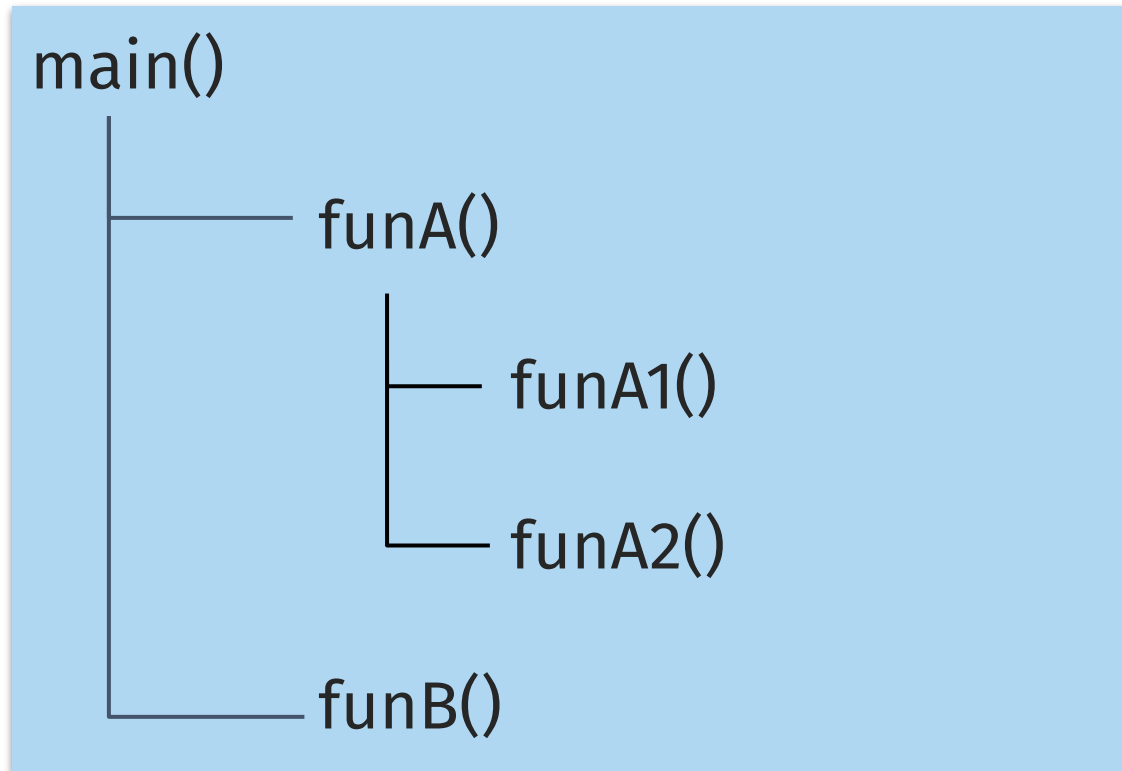
Blame Analysis Algorithm



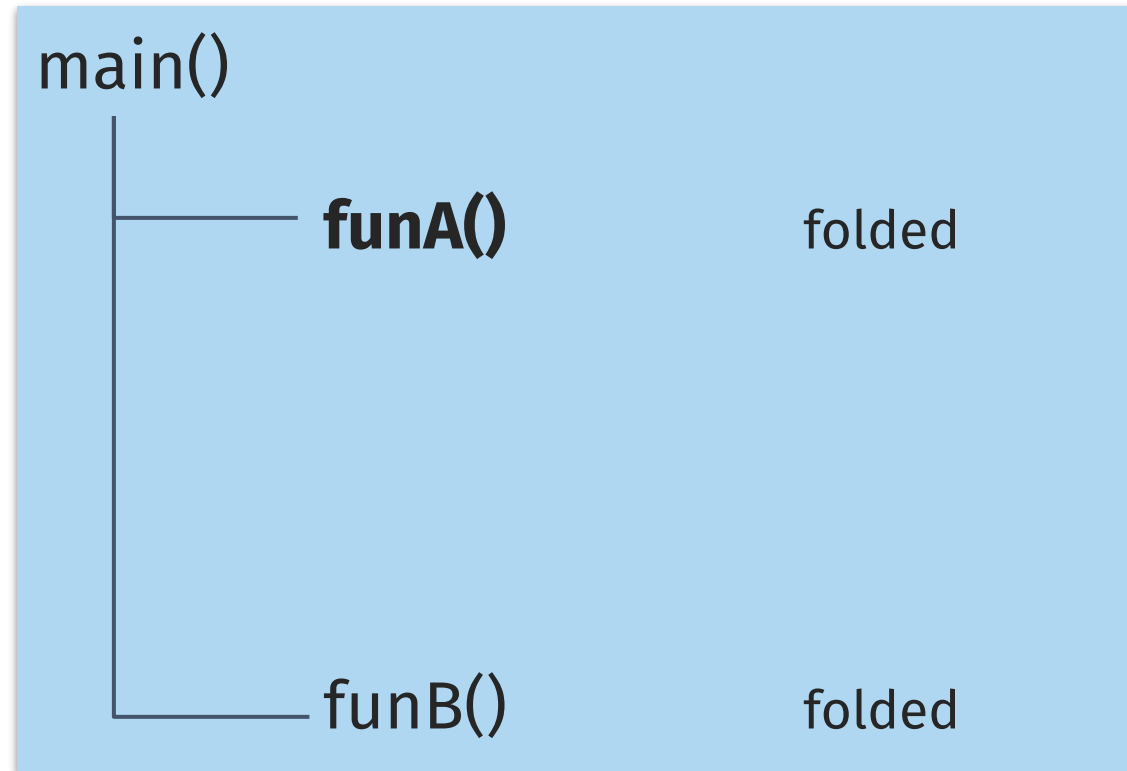
Blame Analysis Algorithm



Blame Analysis Algorithm (Illustration on Example)

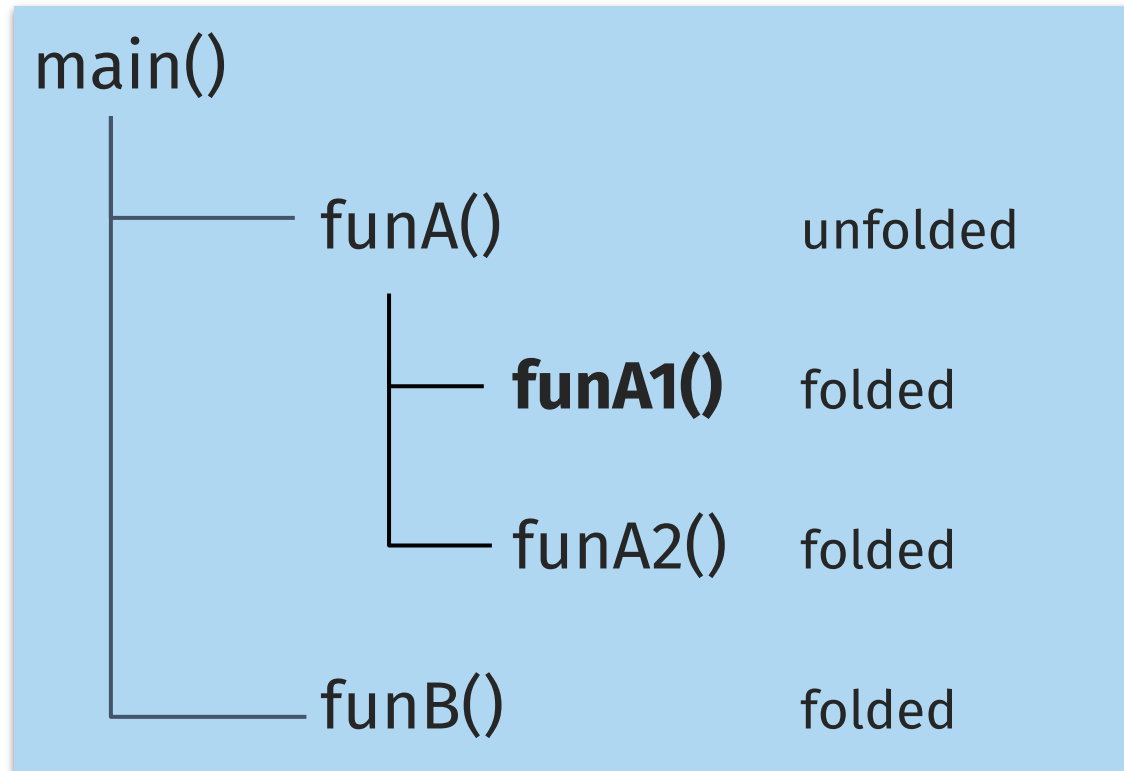


Blame Analysis Algorithm (Illustration on Example)



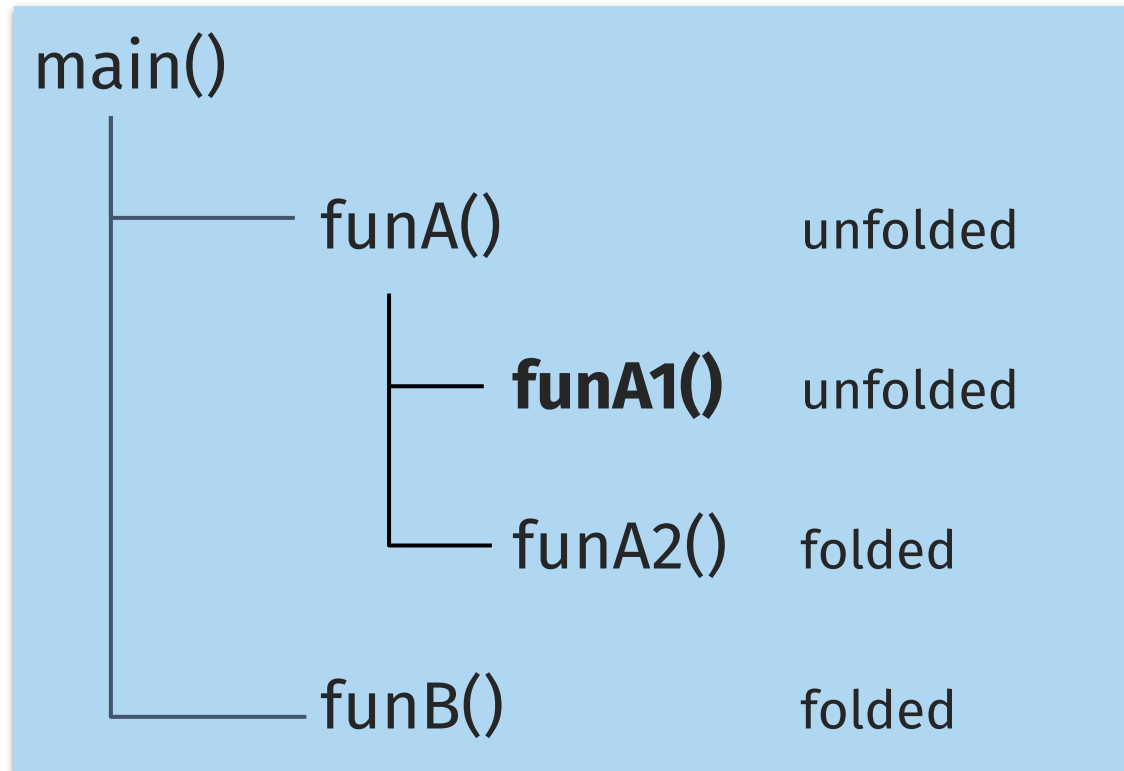
First candidate:
funA()

Blame Analysis Algorithm (Illustration on Example)



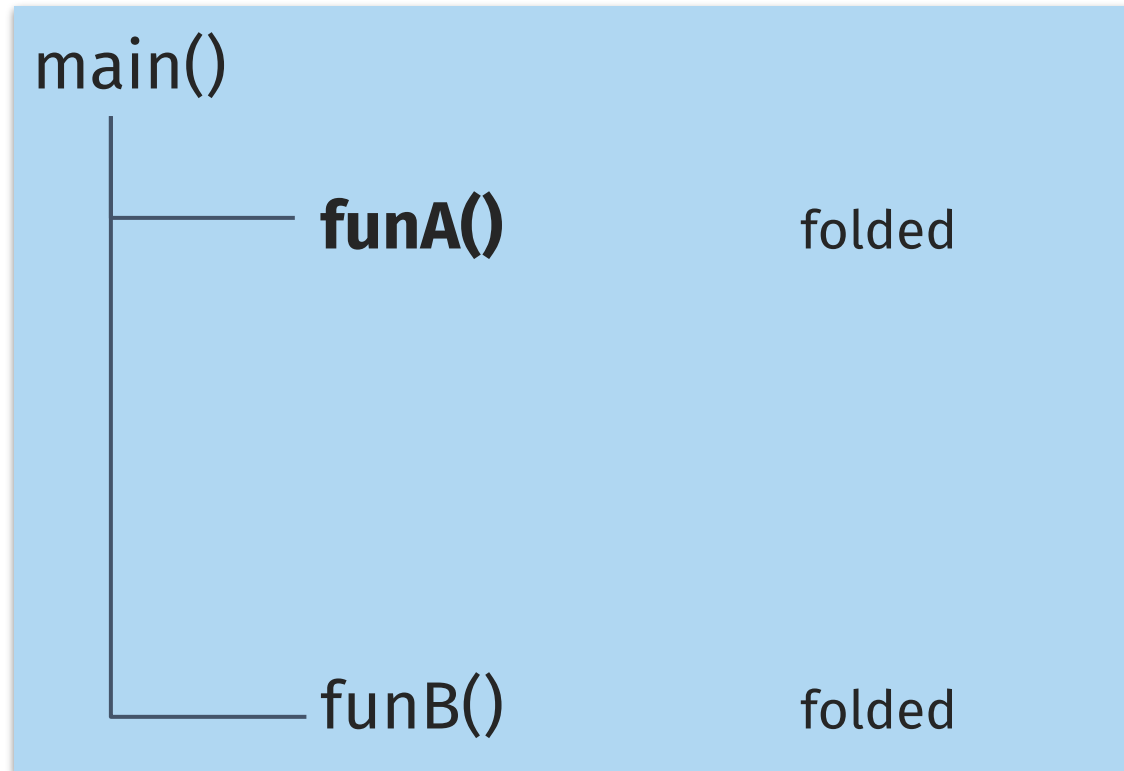
Next candidate:
funA1()

Blame Analysis Algorithm (Illustration on Example)



Blame set:
{{funA1()}}

Blame Analysis Algorithm (Illustration on Example)



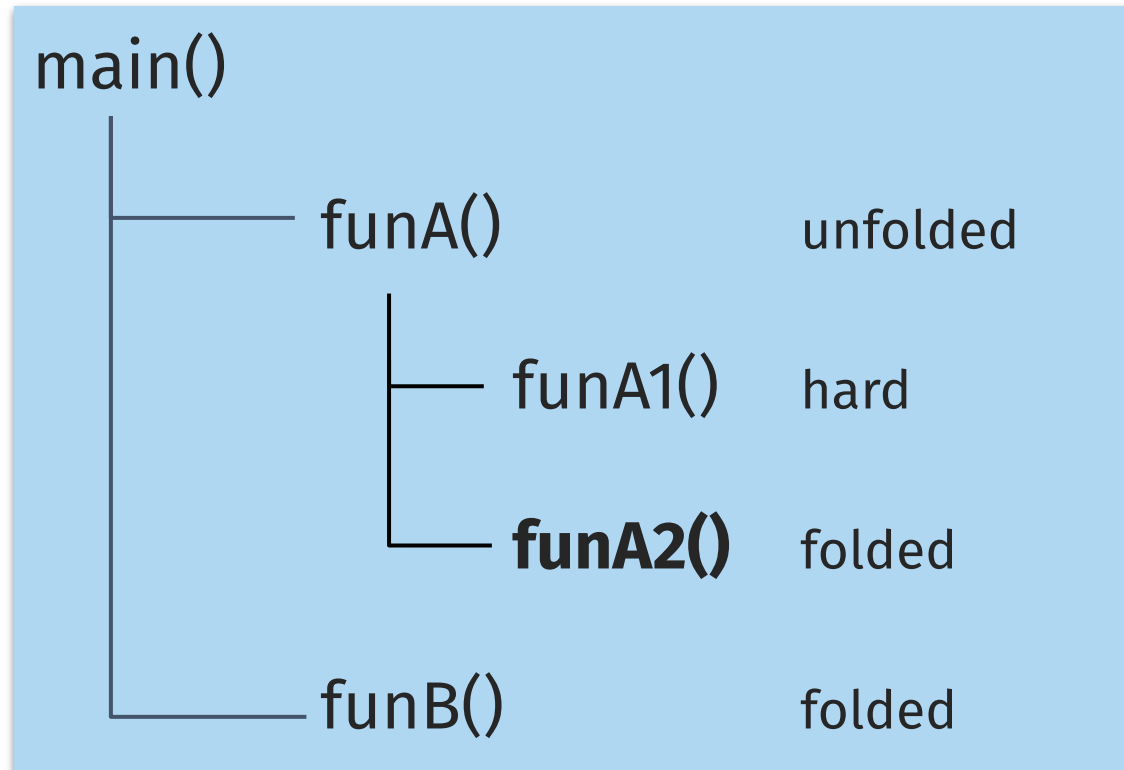
Blame sets:

{{funA1()}}

First candidate:

funA()

Blame Analysis Algorithm (Illustration on Example)



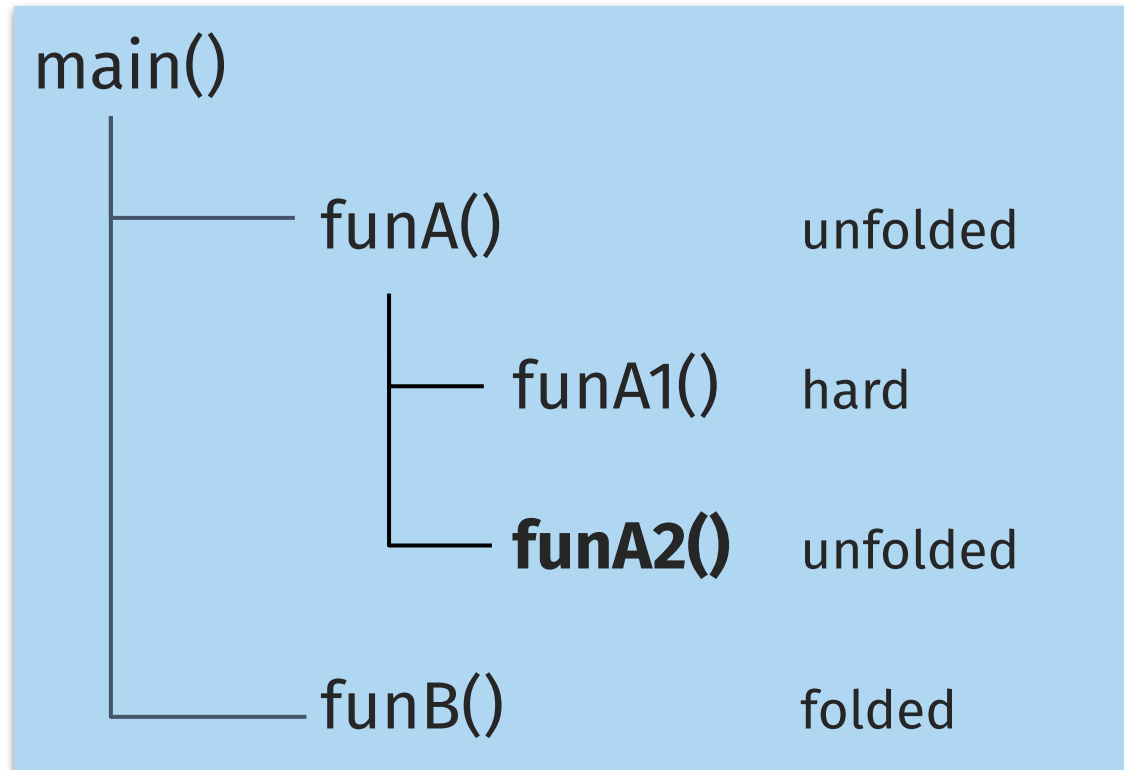
Blame sets:

{{funA1()}}

Next candidate:

funA2()

Blame Analysis Algorithm (Illustration on Example)



Blame sets:
`{{funA1()},`
`{funA2()]}`

Evaluation

RQ1

Can our implementation of blame computation identify functions whose modification would correct the bug?

Evaluation

RQ1

Can our implementation of blame computation identify functions whose modification would correct the bug?

RQ2

To what extent can blame computation derive a result without unfolding all functions, i.e., on a program of reduced size?

Evaluation: Benchmark

26 C programs with ...
... multiple functions
... contracts
... (bugs)

Sources:
CASP benchmark
SV-Comp
LLM-generated
C Tutorials

- one correct and one faulty version of each program (52 in total)
- bugs introduced in one or two functions (up to three lines)
- functions satisfy their contract

Results

verification result
number of functions in program
max number of unfoldings
min number of unfoldings
number of blamed functions
maximal number of blamed functions
correctness of blames

Table 1: Results of blame computation for incorrect (i) and correct (c) versions of all programs

program	verifier i/c	#funs	max unf. i/c	min unf. i/c	blame i (max)	correct?
arithmetic	X /✓	3	2/1	1/1	2 (1)	yes
armstrong	X /✓	5	3/2	3/2	2 (2)	yes
array3	X /✓	2	2 /1	1/1	2 (1)	yes
average	X /✓	4	2/1	1/1	2 (1)	yes
bm01-conjunctive	X /✓	2	2 /1	1/1	2 (1)	yes
bm02_linear	X /✓	2	1/1	1/1	1 (1)	yes
bm02_linear-2	X /✓	3	2/1	1/1	2 (1)	yes
calcexample	X /✓	3	2/1	1/1	2 (1)	yes
conflict-branch	X /✓	2	2 /2	1/2	2 (1)	yes
controller-int	X /✓	4	2/1	1/1	2 (1)	yes
count up down	X /✓	5	2/1	1/1	2 (1)	yes
decorate	X /✓	6	3/1	1/1	2 (1)	yes
diamond_2	X /✓	3	2/2	1/2	2 (1)	yes
double-inc	X /✓	6	6 /3	1/3	4 (1)	yes
final-score	X /✓	5	3/2	1/2	2 (1)	yes
force_two_inc	X /✓	3	2/1	1/1	3 (2)	yes
fourfold	X /✓	6	5/1	1/1	3 (1)	yes
function	X /✓	3	3 /2	1/2	3 (1)	yes
mixnumbers	X /✓	11	8/2	1/2	5 (1)	yes
overflowmod	X /✓	3	3 /1	1/1	2 (1)	yes
sum-diff-loop	X /✓	5	3/1	1/1	3 (1)	yes
sum-diff-simple	X /✓	4	4 /1	1/1	4 (1)	yes
transform-a	X /✓	5	4/3	1/3	3 (1)	yes
transform-b	X /✓	6	1/1	1/1	1 (1)	yes
triplehalve	X /✓	5	4/4	1/4	2 (1)	yes
unreach-branch	X /✓	2	2 /1	1/1	2 (1)	yes

Results

verification result

number of functions in program

max number of unfoldings

min number of unfoldings

number of blamed functions

maximal number of blamed functions

correctness of blames

Table 1: Results of blame computation for incorrect (i) and correct (c) versions of all programs

program	verifier i/c	#funs	max unf. i/c	min unf. i/c	blame i (max)	correct?
arithmetic	X /✓	3	2/1	1/1	2 (1)	yes
armstrong	X /✓	5	3/2	3/2	2 (2)	yes
array3	X /✓	2	2 /1	1/1	2 (1)	yes
average	X /✓	4	2/1	1/1	2 (1)	yes
bm01-conjunctive	X /✓	2	2 /1	1/1	2 (1)	yes
bm02_linear	X /✓	2	1/1	1/1	1 (1)	yes
bm02_linear-2	X /✓	3	2/1	1/1	2 (1)	yes
calcexample	X /✓	3	2/1	1/1	2 (1)	yes
conflict-branch	X /✓	2	2 /2	1/2	2 (1)	yes
controller-int	X /✓	4	2/1	1/1	2 (1)	yes
count up down	X /✓	5	2/1	1/1	2 (1)	yes
decorate	X /✓	6	3/1	1/1	2 (1)	yes
diamond_2	X /✓	3	2/2	1/2	2 (1)	yes
double-inc	X /✓	6	6 /3	1/3	4 (1)	yes
final-score	X /✓	5	3/2	1/2	2 (1)	yes
force_two_inc	X /✓	3	2/1	1/1	3 (2)	yes
fourfold	X /✓	6	5/1	1/1	3 (1)	yes
function	X /✓	3	3 /2	1/2	3 (1)	yes
mixnumbers	X /✓	11	8/2	1/2	5 (1)	yes
overflowmod	X /✓	3	3 /1	1/1	2 (1)	yes
sum-diff-loop	X /✓	5	3/1	1/1	3 (1)	yes
sum-diff-simple	X /✓	4	4 /1	1/1	4 (1)	yes
transform-a	X /✓	5	4/3	1/3	3 (1)	yes
transform-b	X /✓	6	1/1	1/1	1 (1)	yes
triplehalve	X /✓	5	4/4	1/4	2 (1)	yes
unreach-branch	X /✓	2	2 /1	1/1	2 (1)	yes

Results

verification result

number of functions in program

max number of unfoldings

min number of unfoldings

number of blamed functions

maximal number of blamed functions

correctness of blames

Table 1: Results of blame computation for incorrect (i) and correct (c) versions of all programs

program	verifier i/c	#funs	max unf. i/c	min unf. i/c	blame i (max)	correct?
arithmetic	X /✓	3	2/1	1/1	2 (1)	yes
armstrong	X /✓	5	3/2	3/2	2 (2)	yes
array3	X /✓	2	2 /1	1/1	2 (1)	yes
average	X /✓	4	2/1	1/1	2 (1)	yes
bm01-conjunctive	X /✓	2	2 /1	1/1	2 (1)	yes
bm02_linear	X /✓	2	1/1	1/1	1 (1)	yes
bm02_linear-2	X /✓	3	2/1	1/1	2 (1)	yes
calcexample	X /✓	3	2/1	1/1	2 (1)	yes
conflict-branch	X /✓	2	2 / 2	1/2	2 (1)	yes
controller-int	X /✓	4	2/1	1/1	2 (1)	yes
count up down	X /✓	5	2/1	1/1	2 (1)	yes
decorate	X /✓	6	3/1	1/1	2 (1)	yes
diamond_2	X /✓	3	2/2	1/2	2 (1)	yes
double-inc	X /✓	6	6 /3	1/3	4 (1)	yes
final-score	X /✓	5	3/2	1/2	2 (1)	yes
force_two_inc	X /✓	3	2/1	1/1	3 (2)	yes
fourfold	X /✓	6	5/1	1/1	3 (1)	yes
function	X /✓	3	3 /2	1/2	3 (1)	yes
mixnumbers	X /✓	11	8/2	1/2	5 (1)	yes
overflowmod	X /✓	3	3 /1	1/1	2 (1)	yes
sum-diff-loop	X /✓	5	3/1	1/1	3 (1)	yes
sum-diff-simple	X /✓	4	4 /1	1/1	4 (1)	yes
transform-a	X /✓	5	4/3	1/3	3 (1)	yes
transform-b	X /✓	6	1/1	1/1	1 (1)	yes
triplehalve	X /✓	5	4/4	1/4	2 (1)	yes
unreach-branch	X /✓	2	2 /1	1/1	2 (1)	yes

Results

verification result

number of functions in program

max number of unfoldings

min number of unfoldings

number of blamed functions

maximal number of blamed functions

correctness of blames

Table 1: Results of blame computation for incorrect (i) and correct (c) versions of all programs

program	verifier i/c	#funs	max unf. i/c	min unf. i/c	blame i (max)	correct?
arithmetic	X /✓	3	2/1	1/1	2 (1)	yes
armstrong	X /✓	5	3/2	3/2	2 (2)	yes
array3	X /✓	2	2 /1	1/1	2 (1)	yes
average	X /✓	4	2/1	1/1	2 (1)	yes
bm01-conjunctive	X /✓	2	2 /1	1/1	2 (1)	yes
bm02_linear	X /✓	2	1/1	1/1	1 (1)	yes
bm02_linear-2	X /✓	3	2/1	1/1	2 (1)	yes
calcexample	X /✓	3	2/1	1/1	2 (1)	yes
conflict-branch	X /✓	2	2 /2	1/2	2 (1)	yes
controller-int	X /✓	4	2/1	1/1	2 (1)	yes
count up down	X /✓	5	2/1	1/1	2 (1)	yes
decorate	X /✓	6	3/1	1/1	2 (1)	yes
diamond_2	X /✓	3	2/2	1/2	2 (1)	yes
double-inc	X /✓	6	6 /3	1/3	4 (1)	yes
final-score	X /✓	5	3/2	1/2	2 (1)	yes
force_two_inc	X /✓	3	2/1	1/1	3 (2)	yes
fourfold	X /✓	6	5/1	1/1	3 (1)	yes
function	X /✓	3	3 /2	1/2	3 (1)	yes
mixnumbers	X /✓	11	8/2	1/2	5 (1)	yes
overflowmod	X /✓	3	3 /1	1/1	2 (1)	yes
sum-diff-loop	X /✓	5	3/1	1/1	3 (1)	yes
sum-diff-simple	X /✓	4	4 /1	1/1	4 (1)	yes
transform-a	X /✓	5	4/3	1/3	3 (1)	yes
transform-b	X /✓	6	1/1	1/1	1 (1)	yes
triplehalve	X /✓	5	4/4	1/4	2 (1)	yes
unreach-branch	X /✓	2	2 /1	1/1	2 (1)	yes

Results

verification result

number of functions in program

max number of unfoldings

min number of unfoldings

number of blamed functions

maximal number of blamed functions

correctness of blames

Table 1: Results of blame computation for incorrect (i) and correct (c) versions of all programs

program	verifier i/c	#funs	max unf. i/c	min unf. i/c	blame i (max)	correct?
arithmetic	X / ✓	3	2/1	1/1	2 (1)	yes
armstrong	X / ✓	5	3/2	3/2	2 (2)	yes
array3	X / ✓	2	2 /1	1/1	2 (1)	yes
average	X / ✓	4	2/1	1/1	2 (1)	yes
bm01-conjunctive	X / ✓	2	2 /1	1/1	2 (1)	yes
bm02_linear	X / ✓	2	1/1	1/1	1 (1)	yes
bm02_linear-2	X / ✓	3	2/1	1/1	2 (1)	yes
calcexample	X / ✓	3	2/1	1/1	2 (1)	yes
conflict-branch	X / ✓	2	2 / 2	1/2	2 (1)	yes
controller-int	X / ✓	4	2/1	1/1	2 (1)	yes
count up down	X / ✓	5	2/1	1/1	2 (1)	yes
decorate	X / ✓	6	3/1	1/1	2 (1)	yes
diamond_2	X / ✓	3	2/2	1/2	2 (1)	yes
double-inc	X / ✓	6	6 /3	1/3	4 (1)	yes
final-score	X / ✓	5	3/2	1/2	2 (1)	yes
force_two_inc	X / ✓	3	2/1	1/1	3 (2)	yes
fourfold	X / ✓	6	5/1	1/1	3 (1)	yes
function	X / ✓	3	3 /2	1/2	3 (1)	yes
mixnumbers	X / ✓	11	8/2	1/2	5 (1)	yes
overflowmod	X / ✓	3	3 /1	1/1	2 (1)	yes
sum-diff-loop	X / ✓	5	3/1	1/1	3 (1)	yes
sum-diff-simple	X / ✓	4	4 /1	1/1	4 (1)	yes
transform-a	X / ✓	5	4/3	1/3	3 (1)	yes
transform-b	X / ✓	6	1/1	1/1	1 (1)	yes
triplehalve	X / ✓	5	4/4	1/4	2 (1)	yes
unreach-branch	X / ✓	2	2 /1	1/1	2 (1)	yes

Results

verification result

number of functions in program

max number of unfoldings

min number of unfoldings

number of blamed functions

maximal number of blamed functions

correctness of blames

Table 1: Results of blame computation for incorrect (i) and correct (c) versions of all programs

program	verifier i/c	#funs	max unf. i/c	min unf. i/c	blame i (max)	correct?
arithmetic	X / ✓	3	2/1	1/1	2 (1)	yes
armstrong	X / ✓	5	3/2	3/2	2 (2)	yes
array3	X / ✓	2	2 /1	1/1	2 (1)	yes
average	X / ✓	4	2/1	1/1	2 (1)	yes
bm01-conjunctive	X / ✓	2	2 /1	1/1	2 (1)	yes
bm02_linear	X / ✓	2	1/1	1/1	1 (1)	yes
bm02_linear-2	X / ✓	3	2/1	1/1	2 (1)	yes
calcexample	X / ✓	3	2/1	1/1	2 (1)	yes
conflict-branch	X / ✓	2	2 / 2	1/2	2 (1)	yes
controller-int	X / ✓	4	2/1	1/1	2 (1)	yes
count up down	X / ✓	5	2/1	1/1	2 (1)	yes
decorate	X / ✓	6	3/1	1/1	2 (1)	yes
diamond_2	X / ✓	3	2/2	1/2	2 (1)	yes
double-inc	X / ✓	6	6 /3	1/3	4 (1)	yes
final-score	X / ✓	5	3/2	1/2	2 (1)	yes
force_two_inc	X / ✓	3	2/1	1/1	3 (2)	yes
fourfold	X / ✓	6	5/1	1/1	3 (1)	yes
function	X / ✓	3	3 /2	1/2	3 (1)	yes
mixnumbers	X / ✓	11	8/2	1/2	5 (1)	yes
overflowmod	X / ✓	3	3 /1	1/1	2 (1)	yes
sum-diff-loop	X / ✓	5	3/1	1/1	3 (1)	yes
sum-diff-simple	X / ✓	4	4 /1	1/1	4 (1)	yes
transform-a	X / ✓	5	4/3	1/3	3 (1)	yes
transform-b	X / ✓	6	1/1	1/1	1 (1)	yes
triplehalve	X / ✓	5	4/4	1/4	2 (1)	yes
unreach-branch	X / ✓	2	2 /1	1/1	2 (1)	yes

Results: RQ1

RQ1

Can our implementation of blame computation identify functions whose modification would correct the bug?

Table 1: Results of blame computation for incorrect (i) and correct (c) versions of all programs

program	verifier i/c	#funs	max unf. i/c	min unf. i/c	blame i (max)	correct?
arithmetic	X /✓	3	2/1	1/1	2 (1)	yes
armstrong	X /✓	5	3/2	3/2	2 (2)	yes
array3	X /✓	2	2 /1	1/1	2 (1)	yes
average	X /✓	4	2/1	1/1	2 (1)	yes
bm01-conjunctive	X /✓	2	2 /1	1/1	2 (1)	yes
bm02_linear	X /✓	2	1/1	1/1	1 (1)	yes
bm02_linear-2	X /✓	3	2/1	1/1	2 (1)	yes
calcexample	X /✓	3	2/1	1/1	2 (1)	yes
conflict-branch	X /✓	2	2 /2	1/2	2 (1)	yes
controller-int	X /✓	4	2/1	1/1	2 (1)	yes
count up down	X /✓	5	2/1	1/1	2 (1)	yes
decorate	X /✓	6	3/1	1/1	2 (1)	yes
diamond_2	X /✓	3	2/2	1/2	2 (1)	yes
double-inc	X /✓	6	6 /3	1/3	4 (1)	yes
final-score	X /✓	5	3/2	1/2	2 (1)	yes
force_two_inc	X /✓	3	2/1	1/1	3 (2)	yes
fourfold	X /✓	6	5/1	1/1	3 (1)	yes
function	X /✓	3	3 /2	1/2	3 (1)	yes
mixnumbers	X /✓	11	8/2	1/2	5 (1)	yes
overflowmod	X /✓	3	3 /1	1/1	2 (1)	yes
sum-diff-loop	X /✓	5	3/1	1/1	3 (1)	yes
sum-diff-simple	X /✓	4	4 /1	1/1	4 (1)	yes
transform-a	X /✓	5	4/3	1/3	3 (1)	yes
transform-b	X /✓	6	1/1	1/1	1 (1)	yes
triplehalve	X /✓	5	4/4	1/4	2 (1)	yes
unreach-branch	X /✓	2	2 /1	1/1	2 (1)	yes

Results: RQ2

RQ2

To what extent can blame computation derive a result without unfolding all functions, i.e., on a program of reduced size?

Table 1: Results of blame computation for incorrect (i) and correct (c) versions of all programs

program	verifier i/c	#funs	max unf. i/c	min unf. i/c	blame i (max)	correct?
arithmetic	X/✓	3	2/1	1/1	2 (1)	yes
armstrong	X/✓	5	3/2	3/2	2 (2)	yes
array3	X/✓	2	2 /1	1/1	2 (1)	yes
average	X/✓	4	2/1	1/1	2 (1)	yes
bm01-conjunctive	X/✓	2	2 /1	1/1	2 (1)	yes
bm02_linear	X/✓	2	1/1	1/1	1 (1)	yes
bm02_linear-2	X/✓	3	2/1	1/1	2 (1)	yes
calcexample	X/✓	3	2/1	1/1	2 (1)	yes
conflict-branch	X/✓	2	2 /2	1/2	2 (1)	yes
controller-int	X/✓	4	2/1	1/1	2 (1)	yes
count up down	X/✓	5	2/1	1/1	2 (1)	yes
decorate	X/✓	6	3/1	1/1	2 (1)	yes
diamond_2	X/✓	3	2/2	1/2	2 (1)	yes
double-inc	X/✓	6	6 /3	1/3	4 (1)	yes
final-score	X/✓	5	3/2	1/2	2 (1)	yes
force_two_inc	X/✓	3	2/1	1/1	3 (2)	yes
fourfold	X/✓	6	5/1	1/1	3 (1)	yes
function	X/✓	3	3 /2	1/2	3 (1)	yes
mixnumbers	X/✓	11	8/2	1/2	5 (1)	yes
overflowmod	X/✓	3	3 /1	1/1	2 (1)	yes
sum-diff-loop	X/✓	5	3/1	1/1	3 (1)	yes
sum-diff-simple	X/✓	4	4 /1	1/1	4 (1)	yes
transform-a	X/✓	5	4/3	1/3	3 (1)	yes
transform-b	X/✓	6	1/1	1/1	1 (1)	yes
triplehalve	X/✓	5	4/4	1/4	2 (1)	yes
unreach-branch	X/✓	2	2 /1	1/1	2 (1)	yes

Threats to Validity

- limited benchmark size and program complexity
- dependence on verifier
- dependence on solver behavior
- contracts may be incomplete or imprecise
- currently restricted to (limited) C and ACSL

Future Work

- blame localization within function bodies
- contract-based explanation witnesses
- evaluation on larger and more realistic programs
- combine with other known techniques

Conclusion

- contract-based approach to blame analysis
- localization at the function level
- iterative folding and unfolding guided by MaxSMT
- faulty functions can be identified without full program unfolding