

ESBMC v6.1

Mikhail R. Gadelha
Felipe R. Monteiro
Lucas C. Cordeiro
Denis A. Nicole

25th Intl. Conference on Tools and Algorithms for the Construction and Analysis of Systems

8th Intl. Competition on Software Verification

ESBMC v6.1

Mikhail R. Gadelha, Felipe R. Monteiro, Lucas C. Cordeiro, and Denis A. Nicole



UNIVERSITY OF
Southampton



MANCHESTER
1824

ESBMC

Gadelha *et al.*, ASE'18

- SMT-based bounded model checker of single- and multi-threaded C/C++ programs

turned 10 years old in 2018

- Combines BMC, k -induction and abstract interpretation:

path towards correctness proof

bug hunting

- Exploits SMT solvers and their background theories

optimized encodings for pointers, bit operations, unions, arithmetic over- and underflow, and floating-points

ESBMC

Gadelha *et al.*, ASE'18

- SMT-based bounded model checker of single- and multi-threaded C/C++ programs

- pointer safety
- array bounds
- division by zero
- user-specified assertions
- memory leaks
- arithmetic under- and overflow
- atomicity and order violations
- deadlock
- data race

**Properties checked
by ESBMC**



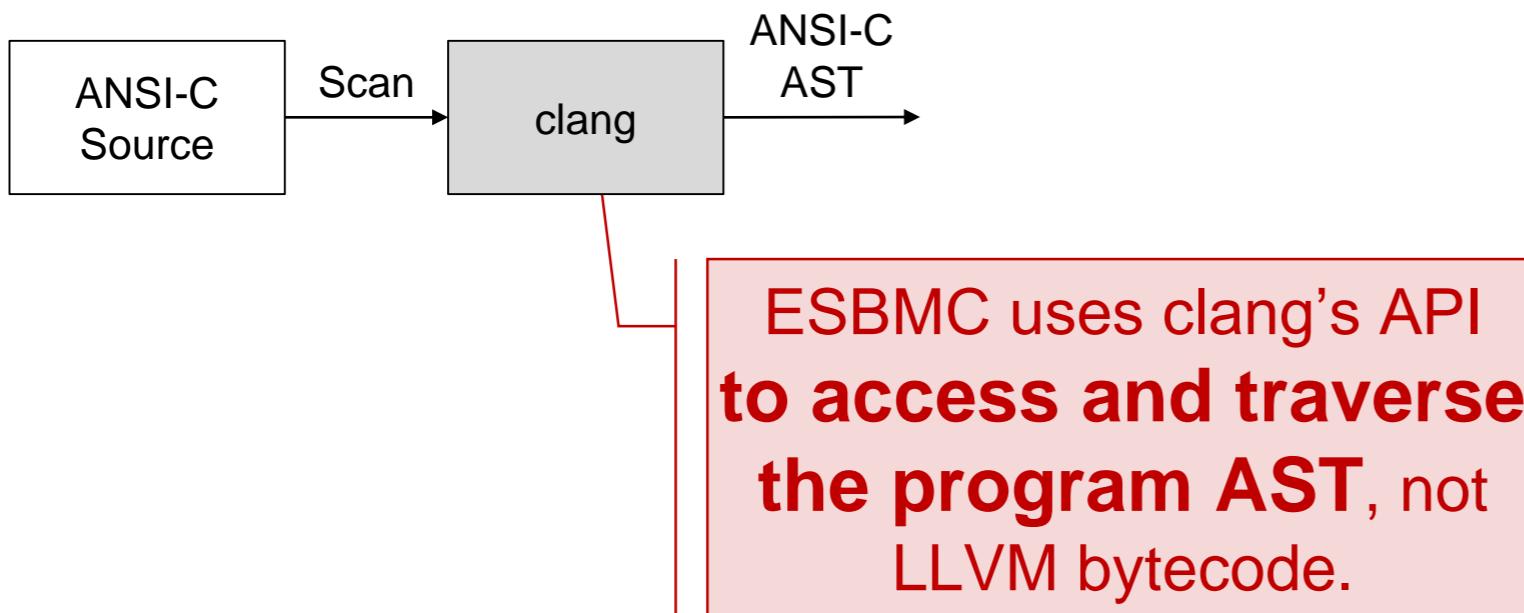
ESBMC Architecture

- ESBMC-falsif uses an incremental BMC approach while ESBMC-kind uses a bidirectional k -induction to falsify properties

ANSI-C
Source

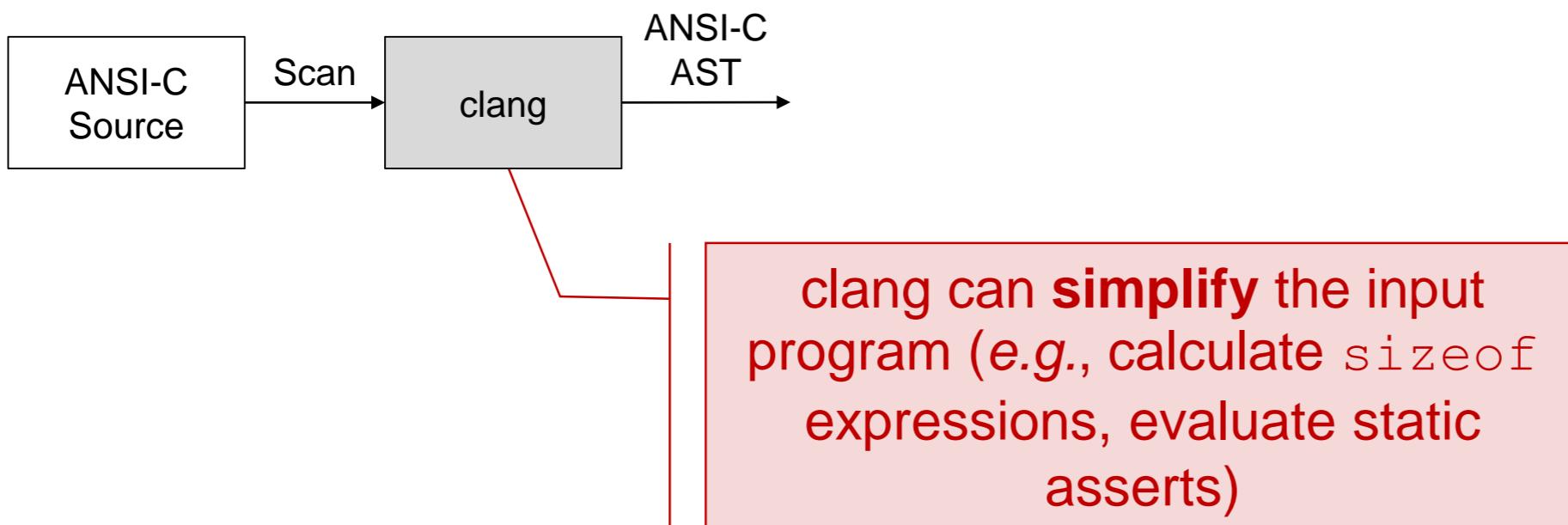
ESBMC Architecture

- ESBMC-falsif uses an incremental BMC approach while ESBMC-kind uses a bidirectional k -induction to falsify properties



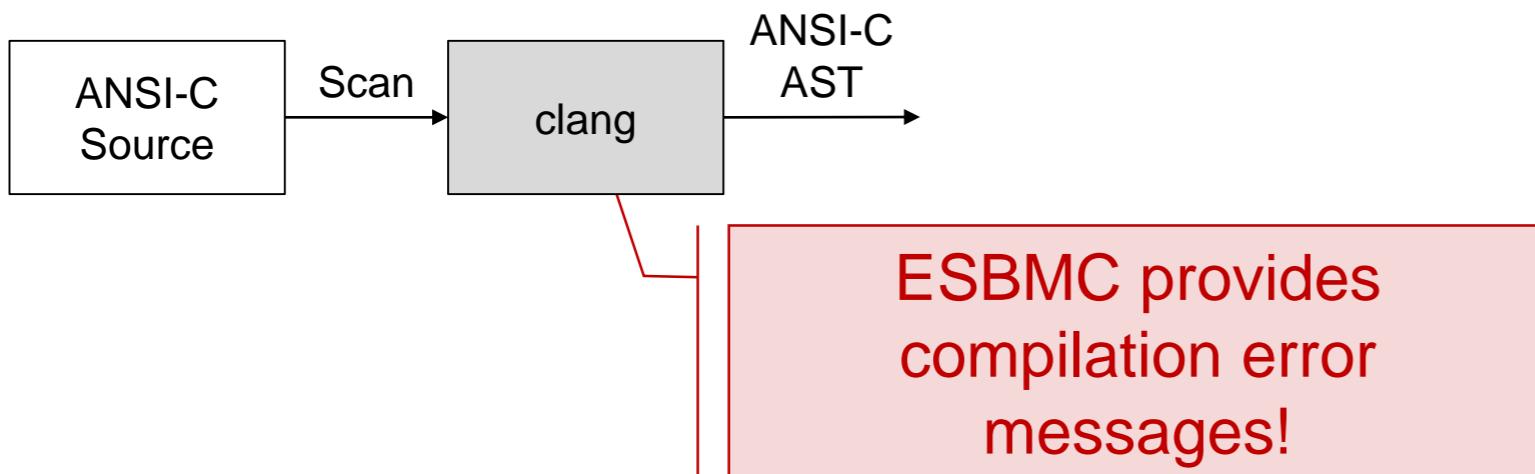
ESBMC Architecture

- ESBMC-falsif uses an incremental BMC approach while ESBMC-kind uses a bidirectional k -induction to falsify properties



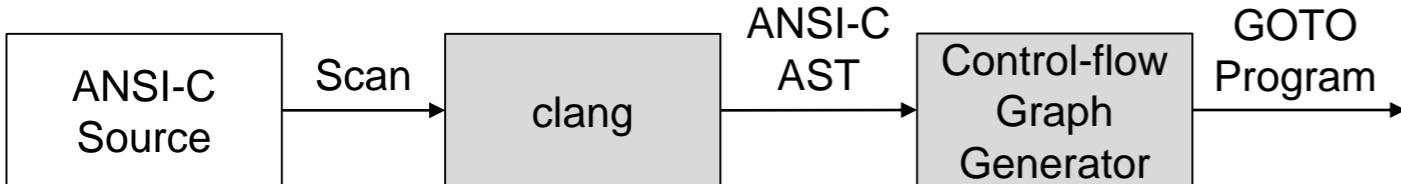
ESBMC Architecture

- ESBMC-falsif uses an incremental BMC approach while ESBMC-kind uses a bidirectional k -induction to falsify properties



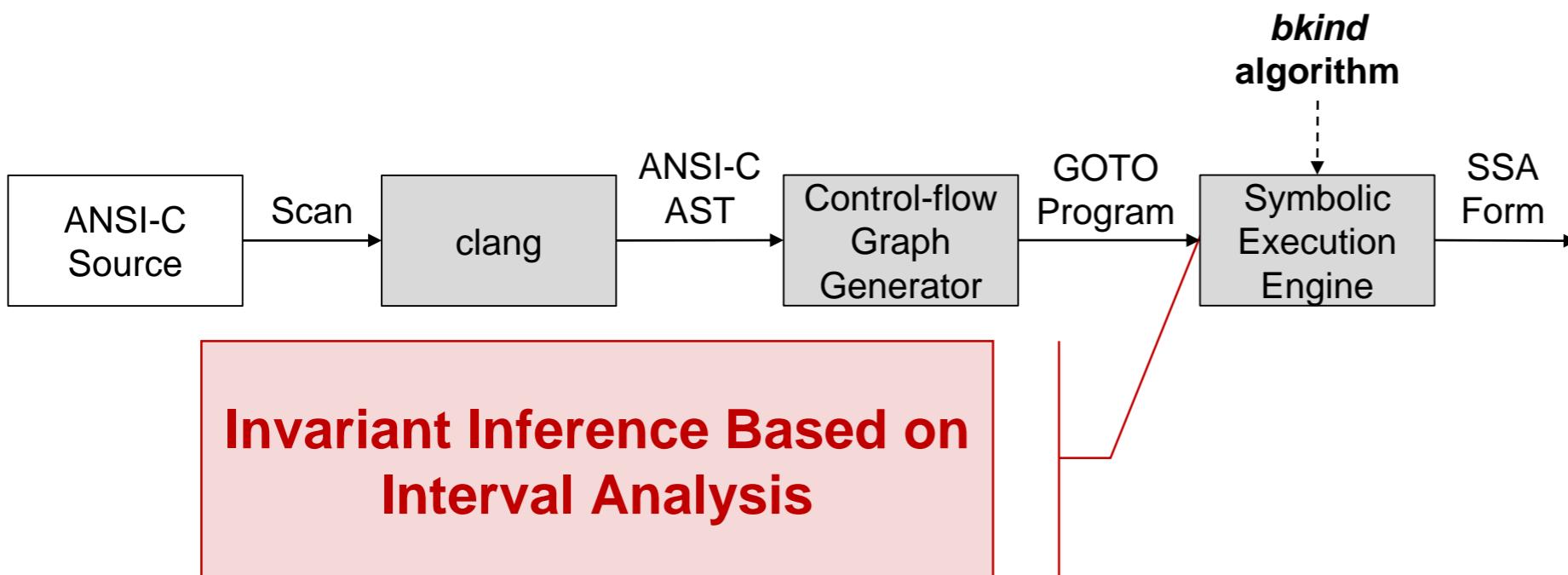
ESBMC Architecture

- The CFG generator takes the program AST and transforms it into an equivalent GOTO program
 - only of assignments, conditional and unconditional branches, assumes, and assertions.



ESBMC Architecture

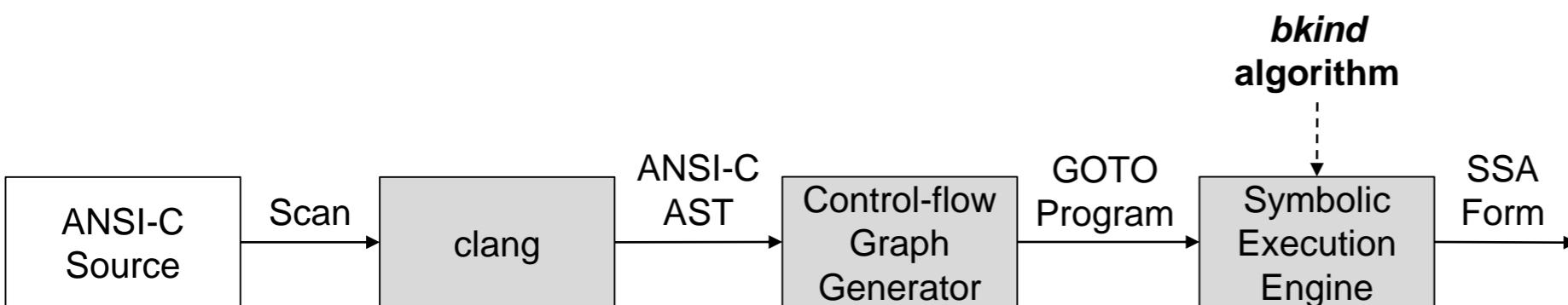
- ESBMC perform a static analysis prior to loop unwinding and overestimate the range that a variable can assume
 - “rectangular” invariant generation based on interval analysis (e.g., $a \leq x \leq b$)



- Abstract-interpretation component from CPROVER
- Only for **integer** variables

ESBMC Architecture

- ESBMC perform a static analysis prior to loop unwinding and overestimate the range that a variable can assume
 - “rectangular” invariant generation based on interval analysis (e.g., $a \leq x \leq b$)



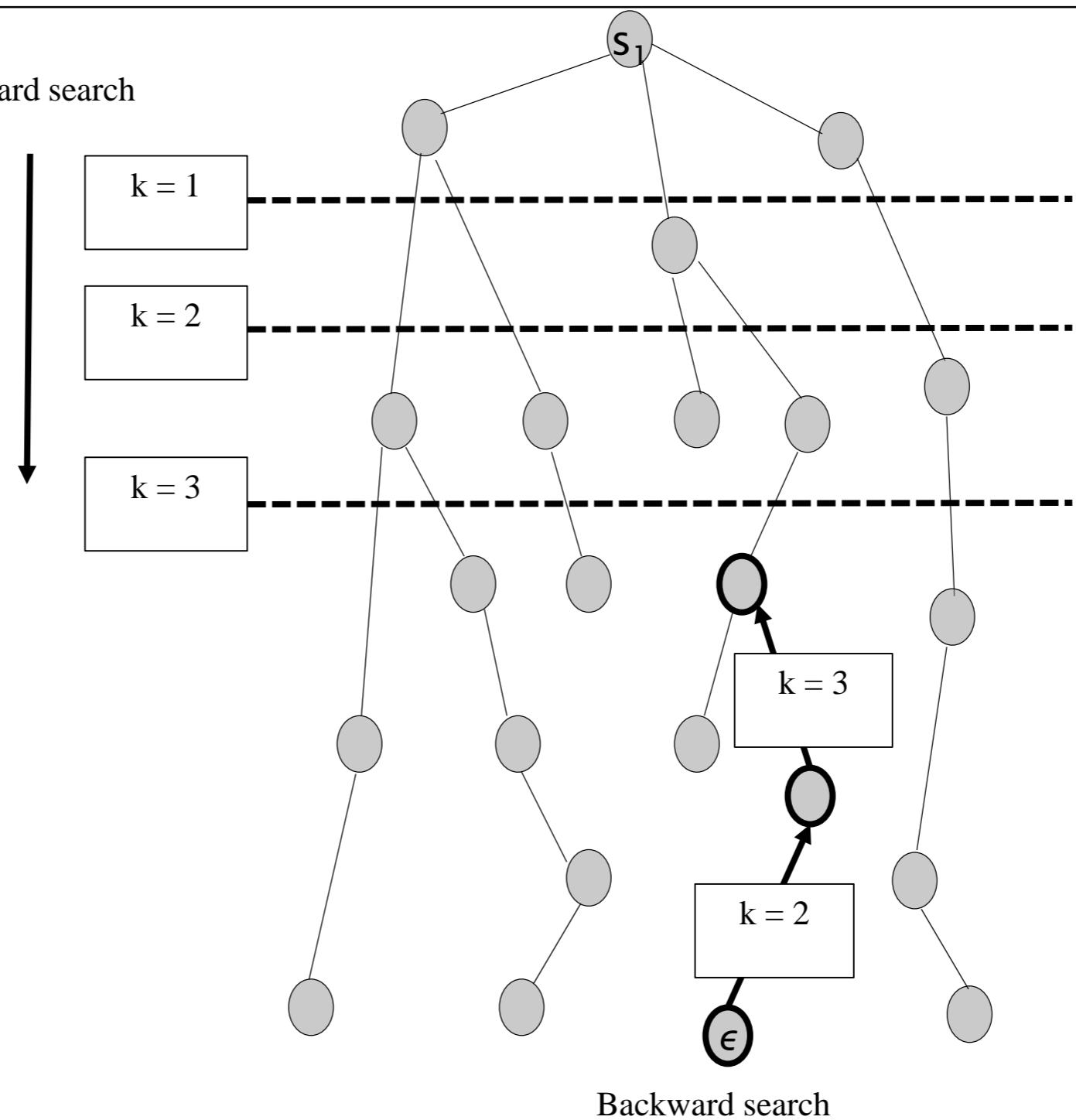
Cornell University arXiv.org > cs > arXiv:1904.02501 Computer Science > Logic in Computer Science Beyond k-induction: Learning from Counterexamples to Bidirectionally Explore the State Space Mikhail R. Gadelha, Felipe R. Monteiro, Enrico Steffinlongo, Lucas C. Cordeiro, Denis A. Nicole

A screenshot of an arXiv.org article page. The header includes the Cornell University logo, the arXiv.org URL, and the article's subject categories: Computer Science > Logic in Computer Science. The title of the paper is "Beyond k-induction: Learning from Counterexamples to Bidirectionally Explore the State Space". The authors listed are Mikhail R. Gadelha, Felipe R. Monteiro, Enrico Steffinlongo, Lucas C. Cordeiro, and Denis A. Nicole.

ESBMC Architecture

- ESBMC performs a forward search to estimate the “rectangle” of the state space

Forward search



and over-

$\leq x \leq b$)

ANSI-C
Source



Cornell Uni

arXiv.org > cs > arX

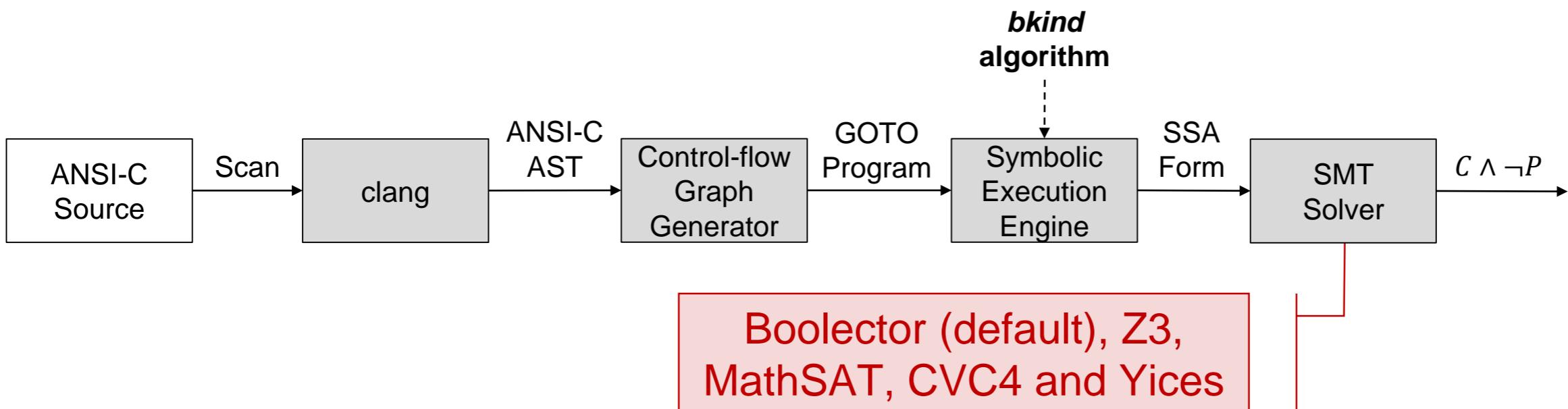
Computer Science

Beyond k-in-
to Bidirec-

Mikhail R. Gadelha
A. Nicole

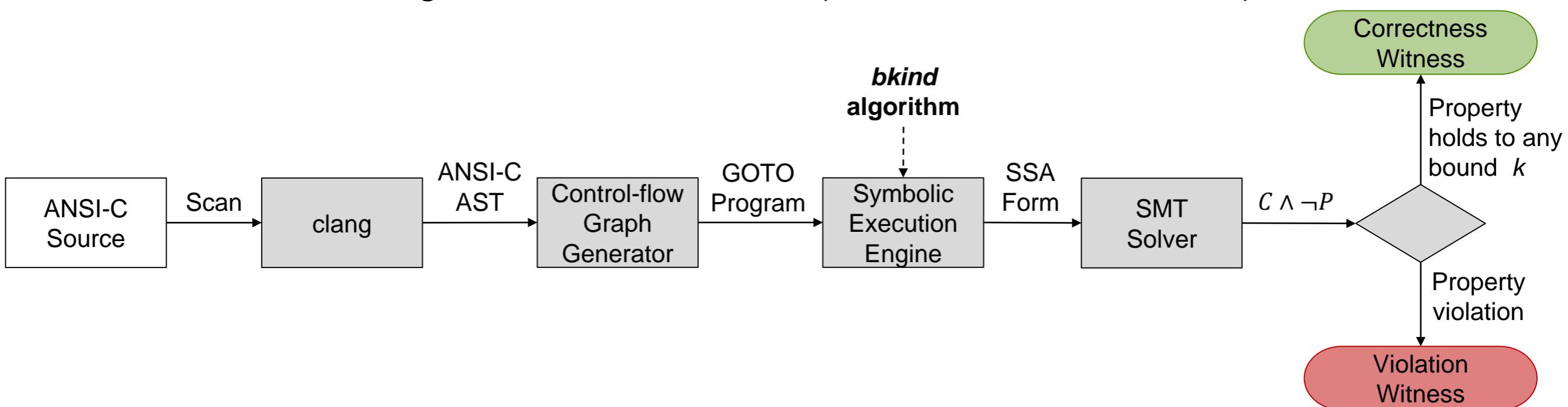
ESBMC Architecture

- The back-end is highly configurable and allows the encoding of quantifier-free formulas
bitvectors, arrays, tuple, fixed-point and floating-point arithmetic (all solvers),
and linear integer and real arithmetic (all solvers but Boolector).



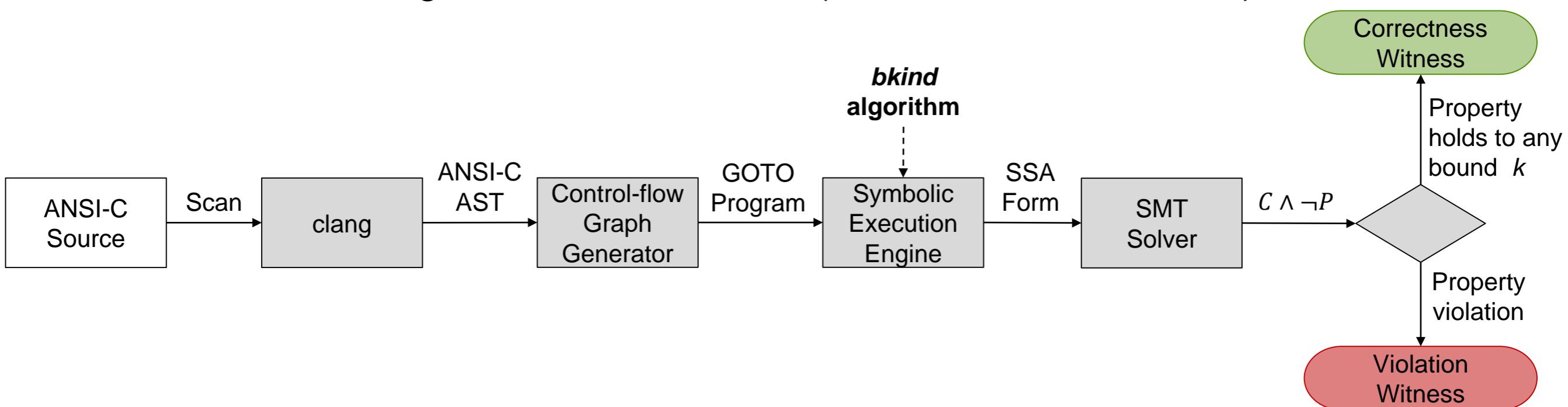
ESBMC Architecture

- The back-end is highly configurable and allows the encoding of quantifier-free formulas
bitvectors, arrays, tuple, fixed-point and floating-point arithmetic (all solvers),
and linear integer and real arithmetic (all solvers but Boolector).



ESBMC Architecture

- The back-end is highly configurable and allows the encoding of quantifier-free formulas
bitvectors, arrays, tuple, fixed-point and floating-point arithmetic (all solvers),
and linear integer and real arithmetic (all solvers but Boolector).



- A test specification is then derived from the violation witness.

ESBMC Architecture

The back end is highly configurable and allows the encoding of

```
1  □<?xml version="1.0" encoding="UTF-8" standalone="no"?><!DOCTYPE testcase PUBLIC "+//IDN
sosy-lab.org//DTD test-format testcase 1.0//EN" "https://sosy-lab.org/test-format/testcase-1.0.dtd
```

- A test specification is then derived from the violation witness.

Strengths & Weaknesses

- ESBMC-falsif uses a naïve approach: it unrolls the program incrementally starting from $k=1$ until it finds a property violation or exhausts time/memory limit.
- ESBMC-kind uses the bkind algorithm to perform a bidirectional search in the state-space, cutting in half the number of steps to find a property violation.
- We do not support coverage test generation.

```
unsigned int s = 1;
int main() {
    while (1) {
        unsigned int input = __VERIFIER_nondet_int();
        if (input > 5) {
            return 0;
        } else if (input == 1 && s == 1) {
            s = 2;
        } else if (input == 2 && s == 2) {
            s = 3;
        } else if (input == 3 && s == 3) {
            s = 4;
        } else if (input == 4 && s == 4) {
            s = 5;
        } else if (input == 5 && s >= 5) { // satisfiable
            __VERIFIER_error(); // property violation
        }
    }
}
```

Simplified unsafe program extracted from SV-COMP 2018

```
unsigned int s = 1;
int main() {
    while (1) {
        unsigned int input = __Verifier_nondet_int();
        if (input > 5) {
            return 0;
        } else if (input == 1 && s == 1) {
            s = 2;
        } else if (input == 2 && s == 2) {
            s = 3;
        } else if (input == 3 && s == 3) {
            s = 4;
        } else if (input == 4 && s == 4) {
            s = 5;
        } else if (input == 5 && s >= 5) { // satisfiable
            __Verifier_error(); // property violation
        }
    }
}
```

Program under
verification

Enables *bkind*
instead of plain BMC

Enables interval
analysis

esbmc main.c --bkind --interval-analysis

with interval analysis

```
unsigned int s = 1;
int main() {
    while (1) {
        unsigned int input = __VERIFIER_nondet_int();
        if (input > 5) {
            return 0;
        } else if (input == 1 && s == 1) {
            s = 2;
        } else if (input == 2 && s == 2) {
            s = 3;
        } else if (input == 3 && s == 3) {
            s = 4;
        } else if (input == 4 && s == 4) {
            s = 5;
        } else if (input == 5 && s >= 5) { // satisfiable
            __VERIFIER_error(); // property violation
        }
    }
}
```

1

ASSUME $s \leq 5 \text{ && } 1 \leq s$

with interval analysis

```
unsigned int s = 1;
int main() {
    while (1) {
        unsigned int input = __VERIFIER_nondet_int();
        if (input > 5) {
            return 0;
        } else if (input == 1 && s == 1) {
            s = 2;
        } else if (input == 2 && s == 2) {
            s = 3;
        } else if (input == 3 && s == 3) {
            s = 4;
        } else if (input == 4 && s == 4) {
            s = 5;
        } else if (input == 5 && s >= 5) { // satisfiable
            __VERIFIER_error(); // property violation
        }
    }
}
```

1

ASSUME $s \leq 5 \&& 1 \leq s$

2

ASSUME $s \leq 5 \&& 1 \leq s$

with interval analysis

```
unsigned int s = 1;
int main() {
    while (1) {
        unsigned int input = __VERIFIER_nondet_int();
        if (input > 5) {
            return 0;
        } else if (input == 1 && s == 1) {
            s = 2;
        } else if (input == 2 && s == 2) {
            s = 3;
        } else if (input == 3 && s == 3) {
            s = 4;
        } else if (input == 4 && s == 4) {
            s = 5;
        } else if (input == 5 && s >= 5) { // satisfiable
            __VERIFIER_error(); // property violation
        }
    }
}
```

The diagram illustrates the flow of interval analysis through the code. Three green circles labeled 1, 2, and 3 are connected by arrows to three green boxes containing ASSUME statements. Circle 1 points to the first ASSUME box. Circle 2 points to the second ASSUME box. Circle 3 points to the third ASSUME box. The ASSUME statements are:

- ASSUME $s \leq 5 \text{ \&\& } 1 \leq s$
- ASSUME $s \leq 5 \text{ \&\& } 1 \leq s$
- ASSUME $s \leq 5 \text{ \&\& } 1 \leq s \text{ \&\& } 6 \leq \text{input}$

with interval analysis

```
unsigned int s = 1;
int main() {
    while (1) {
        unsigned int input = __VERIFIER_nondet_int();
        if (input > 5) {
            return 0;
        } else if (input == 1 && s == 1) {
            s = 2;
        } else if (input == 2 && s == 2) {
            s = 3;
        } else if (input == 3 && s == 3) {
            s = 4;
        } else if (input == 4 && s == 4) {
            s = 5;
        } else if (input == 5 && s >= 5) { // satisfiable
            __VERIFIER_error(); // property violation
        }
    }
}
```

The diagram illustrates the flow of interval analysis assumptions through the code. It shows four points of analysis (labeled 1, 2, 3, and 4) and the corresponding assumptions made at each point:

- Point 1: ASSUME $s \leq 5 \text{ \&\& } 1 \leq s$
- Point 2: ASSUME $s \leq 5 \text{ \&\& } 1 \leq s$
- Point 3: ASSUME $s \leq 5 \text{ \&\& } 1 \leq s \text{ \&\& } 6 \leq \text{input}$
- Point 4: ASSUME $s == 1 \text{ \&\& } \text{input} == 1$

with interval analysis

```
unsigned int s = 1;
int main() {
    while (1) {
        unsigned int input = __VERIFIER_nondet_int();
        if (input > 5) {
            return 0;
        } else if (input == 1 && s == 1) {
            s = 2;
        } else if (input == 2 && s == 2) {
            s = 3;
        } else if (input == 3 && s == 3) {
            s = 4;
        } else if (input == 4 && s == 4) {
            s = 5;
        } else if (input == 5 && s >= 5) { // satisfiable
            __VERIFIER_error(); // property violation
        }
    }
}
```

The diagram illustrates the flow of interval analysis assumptions through the execution of the provided C code. The code initializes `s` to 1 and enters a loop. Inside the loop, it reads an unsigned integer `input` from a verifier. It then checks if `input` is greater than 5, in which case it returns 0. If `input` is 1 and `s` is 1, it sets `s` to 2. If `input` is 2 and `s` is 2, it sets `s` to 3. If `input` is 3 and `s` is 3, it sets `s` to 4. If `input` is 4 and `s` is 4, it sets `s` to 5. If `input` is 5 and `s` is 5 or greater, it calls `__VERIFIER_error()`, indicating a property violation.

- Assumption 1: `ASSUME s <= 5 && 1 <= s`
- Assumption 2: `ASSUME s <= 5 && 1 <= s`
- Assumption 3: `ASSUME s <= 5 && 1 <= s && 6 <= input`
- Assumption 4: `ASSUME s == 1 && input == 1`
- Assumption 5: `ASSUME s == 2 && input == 2`

with interval analysis

```
unsigned int s = 1;
int main() {
    while (1) {
        unsigned int input = __VERIFIER_nondet_int();
        if (input > 5) {
            return 0;
        } else if (input == 1 && s == 1) {
            s = 2;
        } else if (input == 2 && s == 2) {
            s = 3;
        } else if (input == 3 && s == 3) {
            s = 4;
        } else if (input == 4 && s == 4) {
            s = 5;
        } else if (input == 5 && s >= 5) { // satisfiable
            __VERIFIER_error(); // property violation
        }
    }
}
```

The diagram illustrates the flow of interval analysis assumptions through the code. Six green circles, labeled 1 through 6, are connected by arrows to corresponding `ASSUME` statements in the code. Circle 1 points to the first `ASSUME` statement. Circle 2 points to the second. Circle 3 points to the third. Circle 4 points to the fourth. Circle 5 points to the fifth. Circle 6 points to the sixth.

- 1: ASSUME `s <= 5 && 1 <= s`
- 2: ASSUME `s <= 5 && 1 <= s`
- 3: ASSUME `s <= 5 && 1 <= s && 6 <= input`
- 4: ASSUME `s == 1 && input == 1`
- 5: ASSUME `s == 2 && input == 2`
- 6: ASSUME `s == 3 && input == 3`

with interval analysis

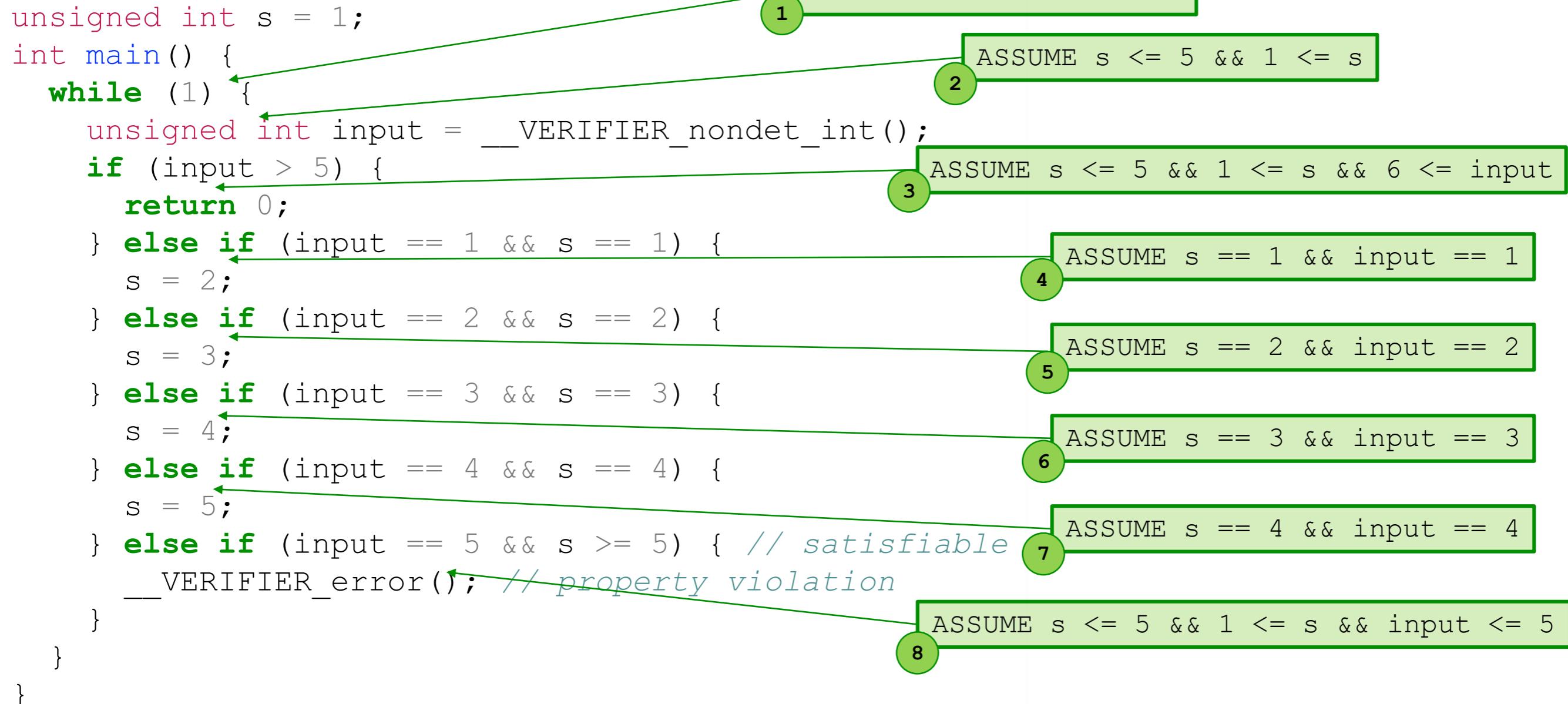
```

unsigned int s = 1;
int main() {
    while (1) {
        unsigned int input = __VERIFIER_nondet_int();
        if (input > 5) {
            return 0;
        } else if (input == 1 && s == 1) {
            s = 2;
        } else if (input == 2 && s == 2) {
            s = 3;
        } else if (input == 3 && s == 3) {
            s = 4;
        } else if (input == 4 && s == 4) {
            s = 5;
        } else if (input == 5 && s >= 5) { // satisfiable
            __VERIFIER_error(); // property violation
        }
    }
}

```

The diagram illustrates the flow of interval analysis assumptions through the code. Seven green circles, labeled 1 through 7, are connected by arrows to seven corresponding green boxes containing assumptions. The assumptions are as follows:

- Circle 1: ASSUME $s \leq 5 \text{ && } 1 \leq s$
- Circle 2: ASSUME $s \leq 5 \text{ && } 1 \leq s$
- Circle 3: ASSUME $s \leq 5 \text{ && } 1 \leq s \text{ && } 6 \leq \text{input}$
- Circle 4: ASSUME $s == 1 \text{ && } \text{input} == 1$
- Circle 5: ASSUME $s == 2 \text{ && } \text{input} == 2$
- Circle 6: ASSUME $s == 3 \text{ && } \text{input} == 3$
- Circle 7: ASSUME $s == 4 \text{ && } \text{input} == 4$

with interval analysis

with interval analysis

```

unsigned int s = 1;
int main() {
    while (1) {
        unsigned int input = __VERIFIER_nondet_int();
        if (input > 5) {
            return 0;
        } else if (input == 1 && s == 1) {
            s = 2;
        } else if (input == 2 && s == 2) {
            s = 3;
        } else if (input == 3 && s == 3) {
            s = 4;
        } else if (input == 4 && s == 4) {
            s = 5;
        } else if (input == 5 && s >= 5) { // satisfiable
            __VERIFIER_error(); // property violation
        }
    }
}

```

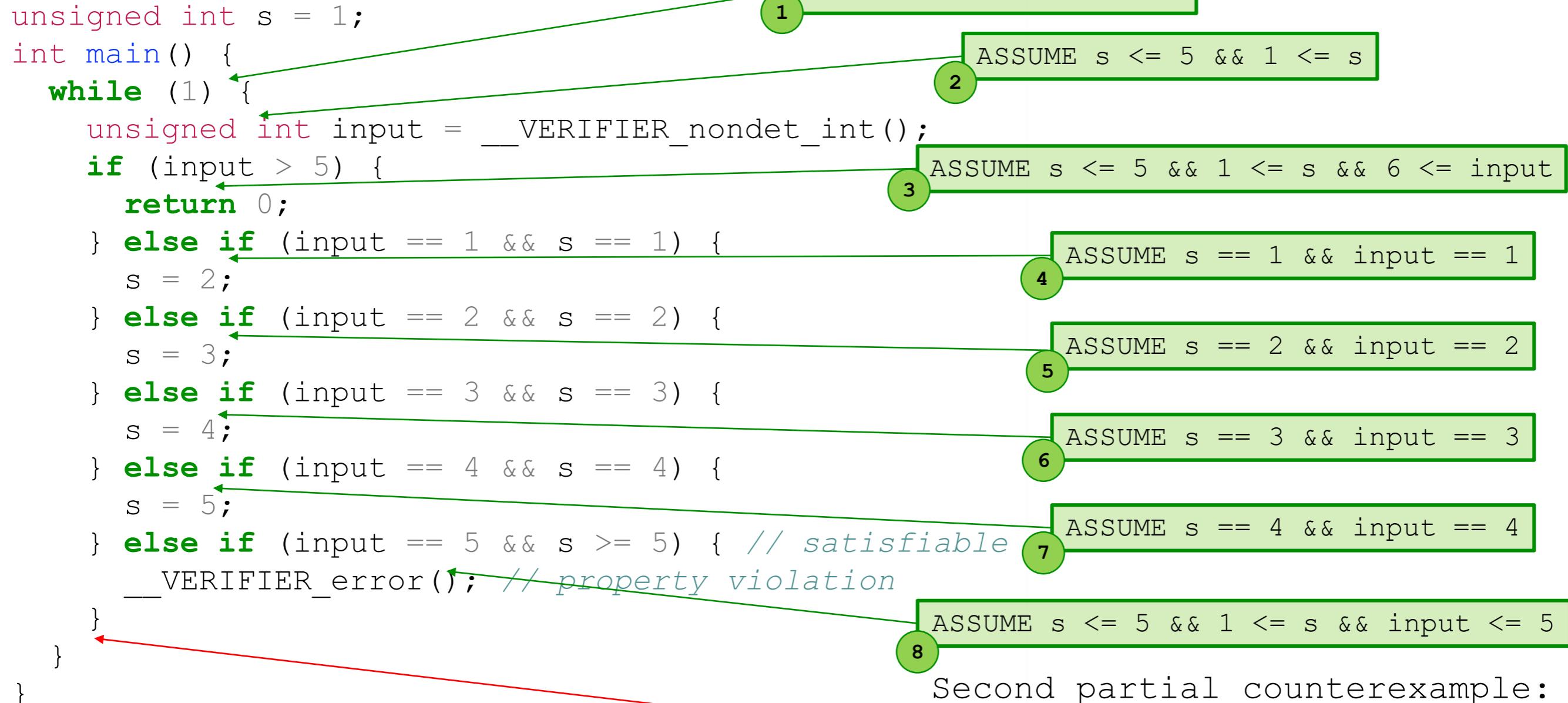
The diagram illustrates the verification process using interval analysis. It shows the state of variables `s` and `input` at different points in the loop:

- Point 1:** ASSUME $s \leq 5 \&& 1 \leq s$
- Point 2:** ASSUME $s \leq 5 \&& 1 \leq s$
- Point 3:** ASSUME $s \leq 5 \&& 1 \leq s \&& 6 \leq input$
- Point 4:** ASSUME $s == 1 \&& input == 1$
- Point 5:** ASSUME $s == 2 \&& input == 2$
- Point 6:** ASSUME $s == 3 \&& input == 3$
- Point 7:** ASSUME $s == 4 \&& input == 4$
- Point 8:** ASSUME $s \leq 5 \&& 1 \leq s \&& input \leq 5$

A red arrow points from Point 8 to a red box containing the assertion `ASSERT s == 5 && input == 5`, indicating a partial counterexample.

First partial counterexample:

`ASSERT s == 5 && input == 5`

with interval analysis

Thank you!

More information available at <http://esbmc.org/>

