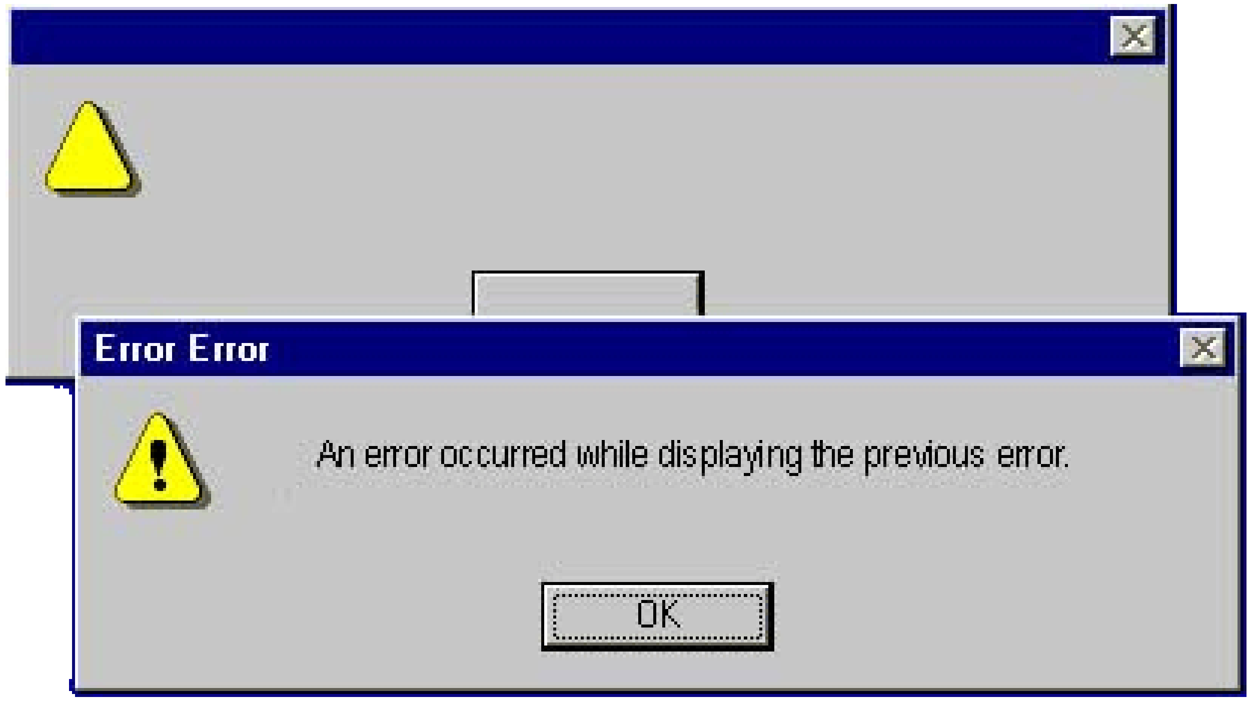


ESBMC v6.1

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
UNIVERSIDADE FEDERAL DO AMAZONAS
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

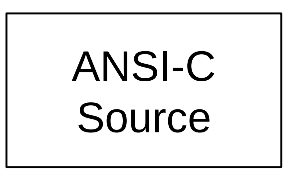
- SMT-based bounded model checker of single- and multi-threaded C/C++ programs

  - 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

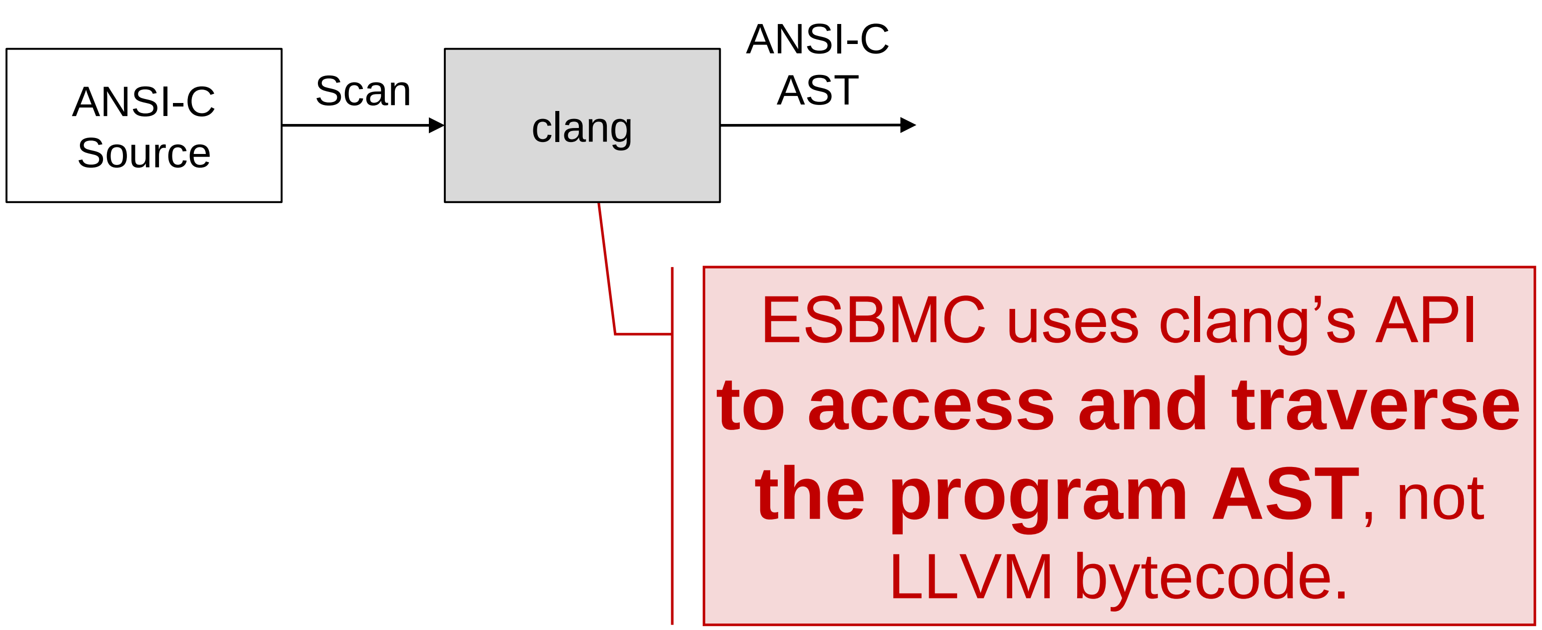
ESBMC Architecture

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



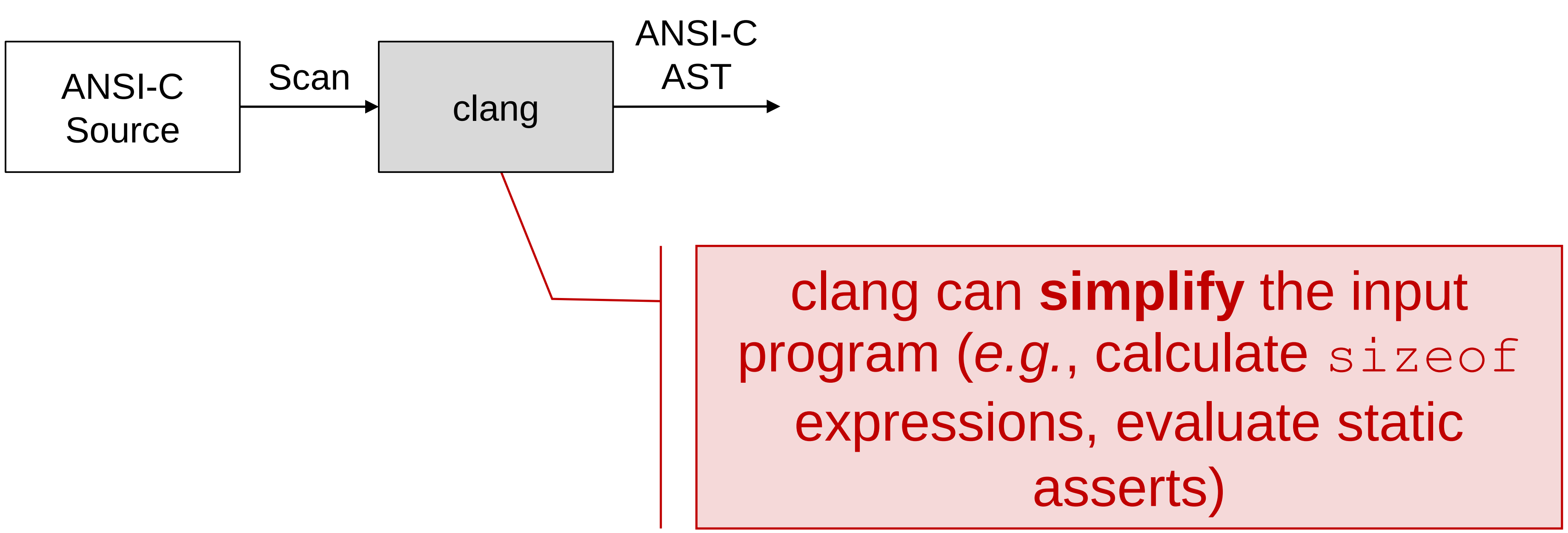
ESBMC Architecture

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



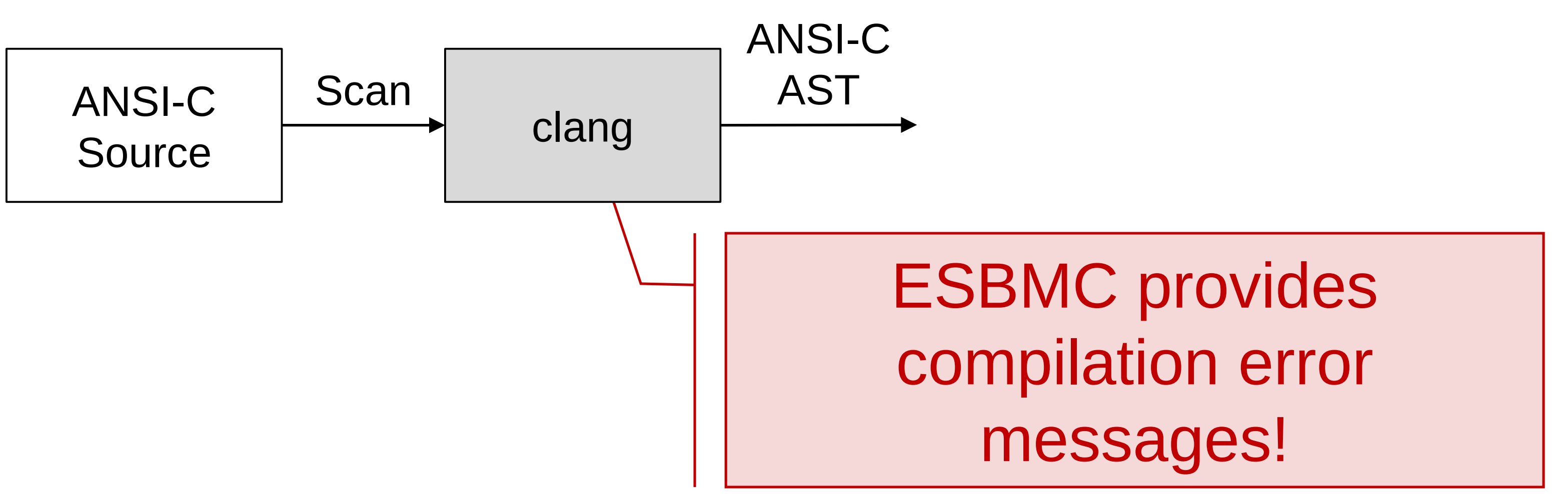
ESBMC Architecture

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



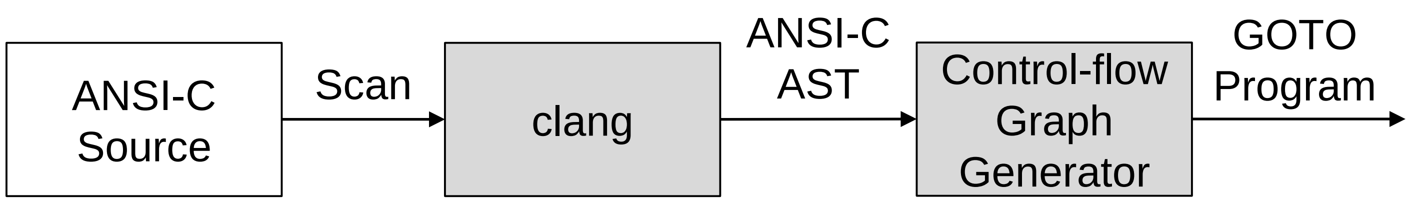
ESBMC Architecture

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



ESBMC Architecture

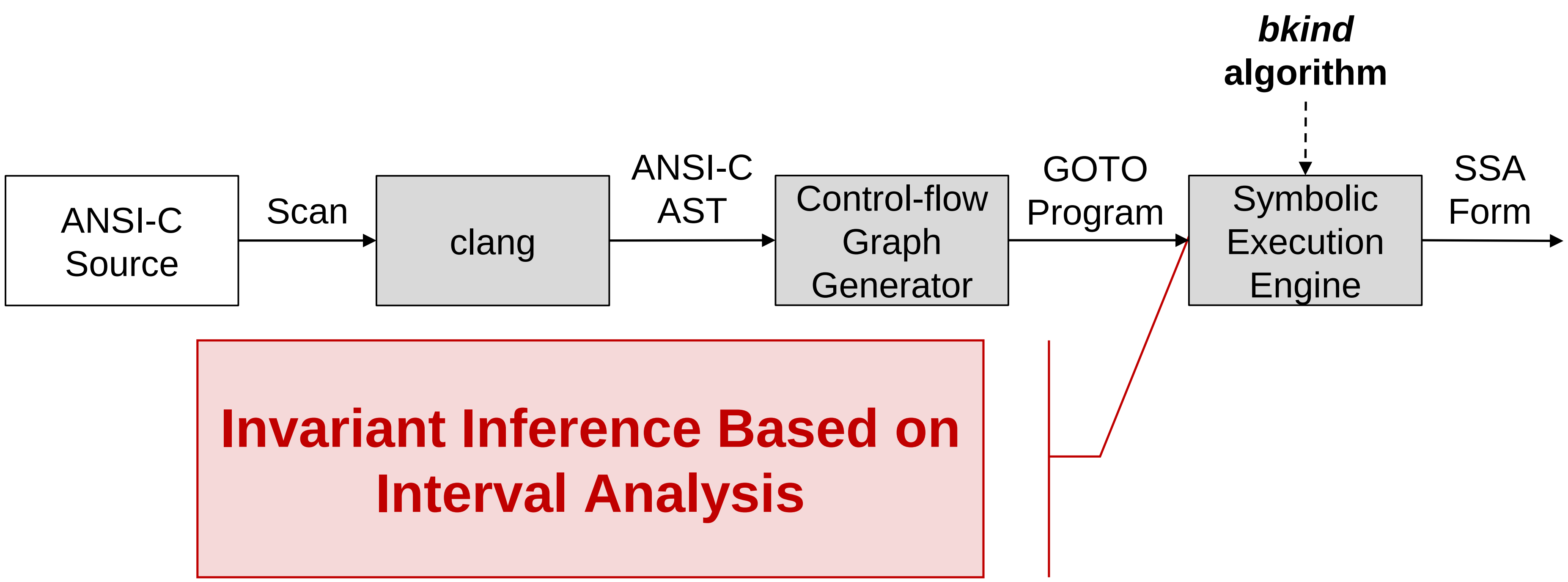
- The CFG generator takes the program AST and transforms it into an equivalent GOTO program
  - only of assignments, conditional and unconditional branches, assumes, and assertions.



ESBMC Architecture

- ESBMC perform a static analysis prior to loop unwinding and over-estimate the range that a variable can assume

“rectangular” invariant generation based on interval analysis (*e.g.*, a ≤ x ≤ b)

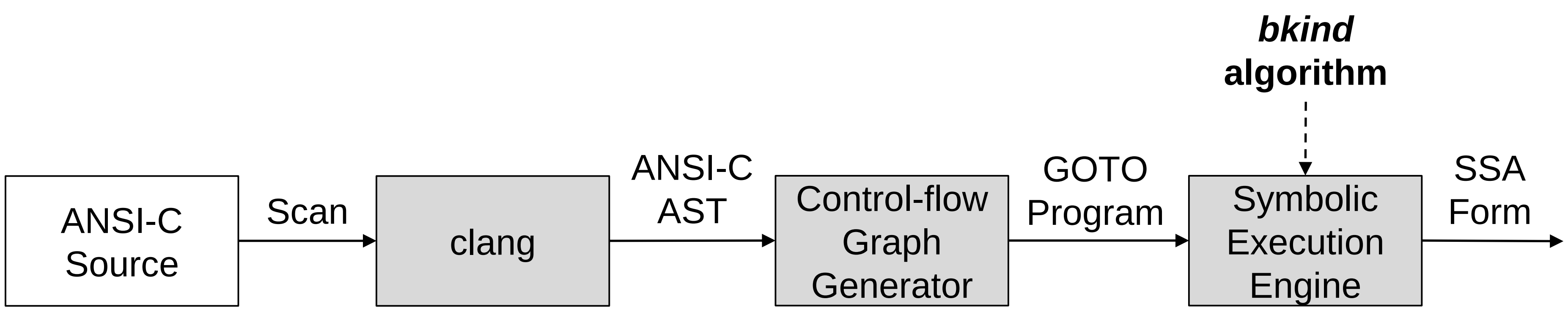


- Abstract-interpretation component from CPROVER
- Only for **integer** variables

ESBMC Architecture

- ESBMC perform a static analysis prior to loop unwinding and over-estimate the range that a variable can assume

  “rectangular” invariant generation based on interval analysis (*e.g.*, $a \leq x \leq b$)




Cornell University
the Sim
arXiv.org > cs > arXiv:1904.02501
Search or Article ID
(Help | Advanced search)
Computer Science > Logic in Computer Science

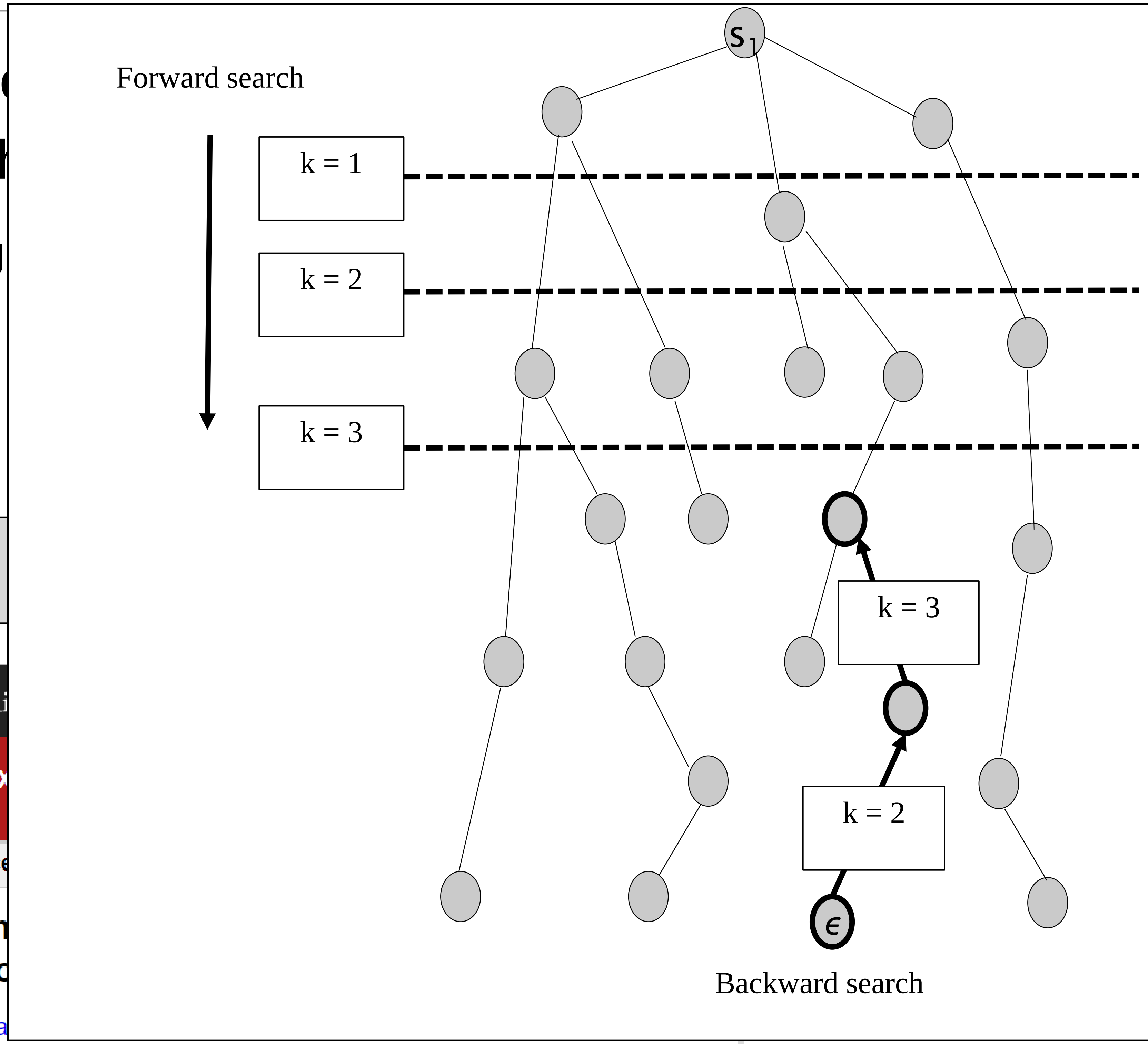
Beyond k-induction: Learning from Counterexamples to Bidirectionally Explore the State Space

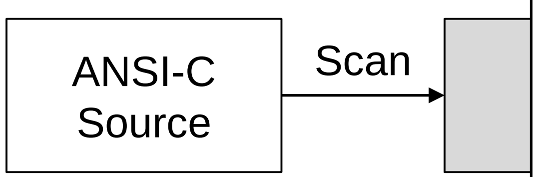
Mikhail R. Gadelha, Felipe R. Monteiro, Enrico Steffinlongo, Lucas C. Cordeiro, Denis A. Nicole

ESBMC Architecture

- ESBMC pe… and over-estimate th…

  "rectang… ≤ x ≤ b)





Cornell Uni…

arXiv.org > cs > arX…

Computer Science…

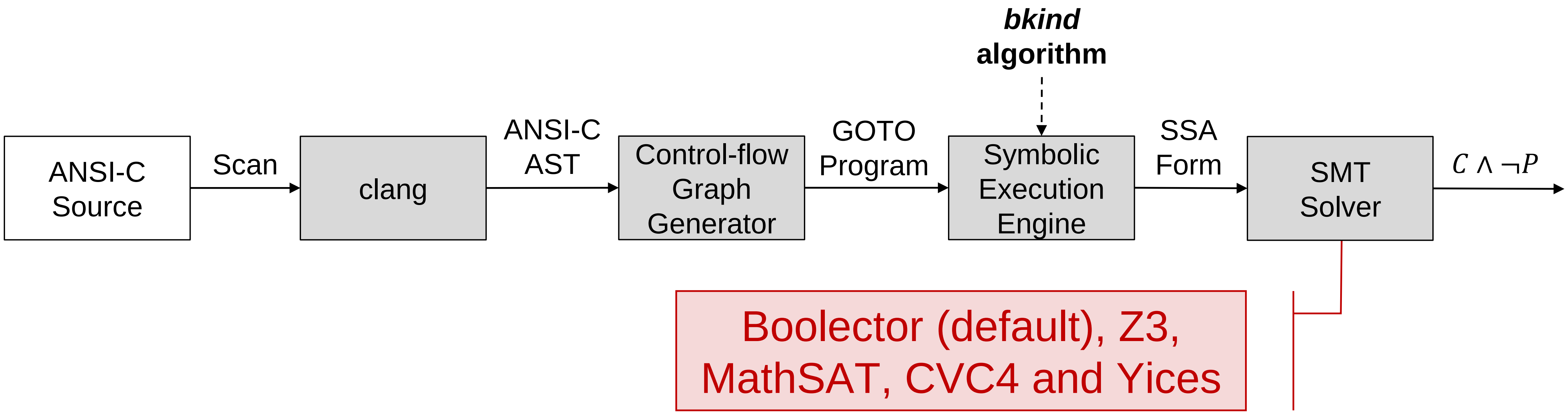
Beyond k-in… to Bidirectio…

Mikhail R. Gadelha… A. Nicole

ESBMC Architecture

- The back-end is highly configurable and allows the encoding of quantifier-free formulas

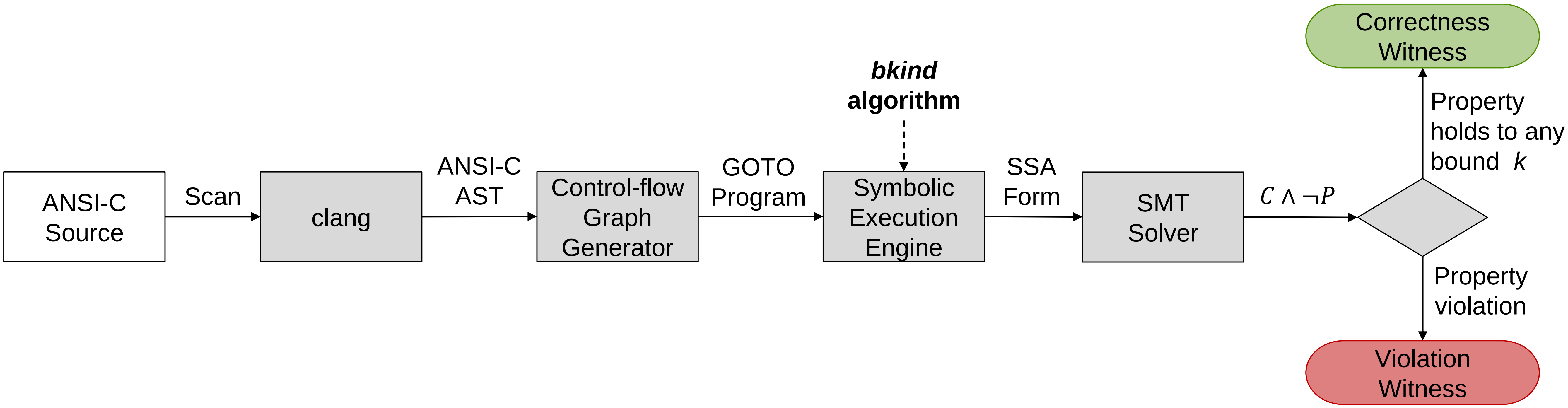
  bitvectors, arrays, tuple, fixed-point and floating-point arithmetic (all solvers), and linear integer and real arithmetic (all solvers but Boolector).



ESBMC Architecture

- The back-end is highly configurable and allows the encoding of quantifier-free formulas

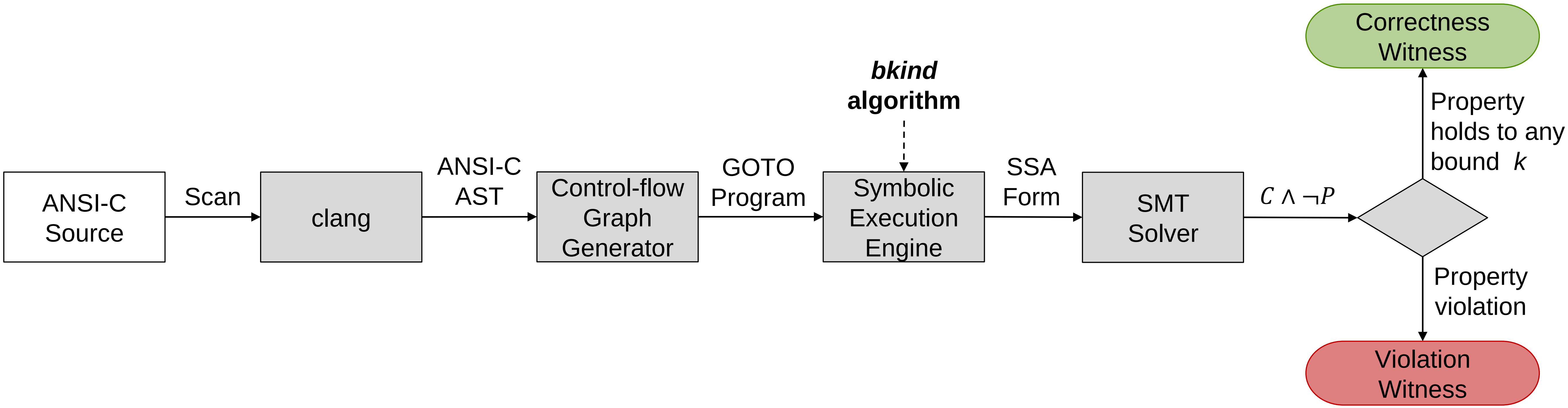
    bitvectors, arrays, tuple, fixed-point and floating-point arithmetic (all solvers), and linear integer and real arithmetic (all solvers but Boolector).



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


```xml
<?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">
<testcase>
  <input>0</input>
  <input>1325171125</input>
  <input>472399531</input>
  <input>1907900516</input>
  <input>-1309687790</input>
  <input>-1111228414</input>
  <input>1325866080</input>
  <input>845879320</input>
  <input>2022020726</input>
  <input>377524192</input>
  <input>-130150474</input>
  <input>1400296004</input>
  <input>140741300</input>
  <input>201440847</input>
  <input>1979366910</input>
  <input>1992568592</input>
  <input>1524518093</input>
</testcase>
```


- 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.

```c
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

```c
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 → `main.c`

Enables *bkind* instead of plain BMC → `--bkind`

Enables interval analysis → `--interval-analysis`

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

with interval analysis

ASSUME s <= 5 && 1 <= s

1

```
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
    }
  }
}
```

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 <= 5 && 1 <= s`

2 `ASSUME s <= 5 && 1 <= 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 <= 5 && 1 <= s

2 ASSUME s <= 5 && 1 <= s

3 ASSUME s <= 5 && 1 <= s && 6 <= 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
    }
  }
}
```

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

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 <= 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

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 <= 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

```c
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 <= 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
7. ASSUME s == 4 && input == 4

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 <= 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
7 ASSUME s == 4 && input == 4
8 ASSUME s <= 5 && 1 <= s && input <= 5
```

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 <= 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
7. ASSUME s == 4 && input == 4
8. ASSUME s <= 5 && 1 <= s && input <= 5

First partial counterexample:

ASSERT s == 5 && input == 5

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 <= 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`

7 `ASSUME s == 4 && input == 4`

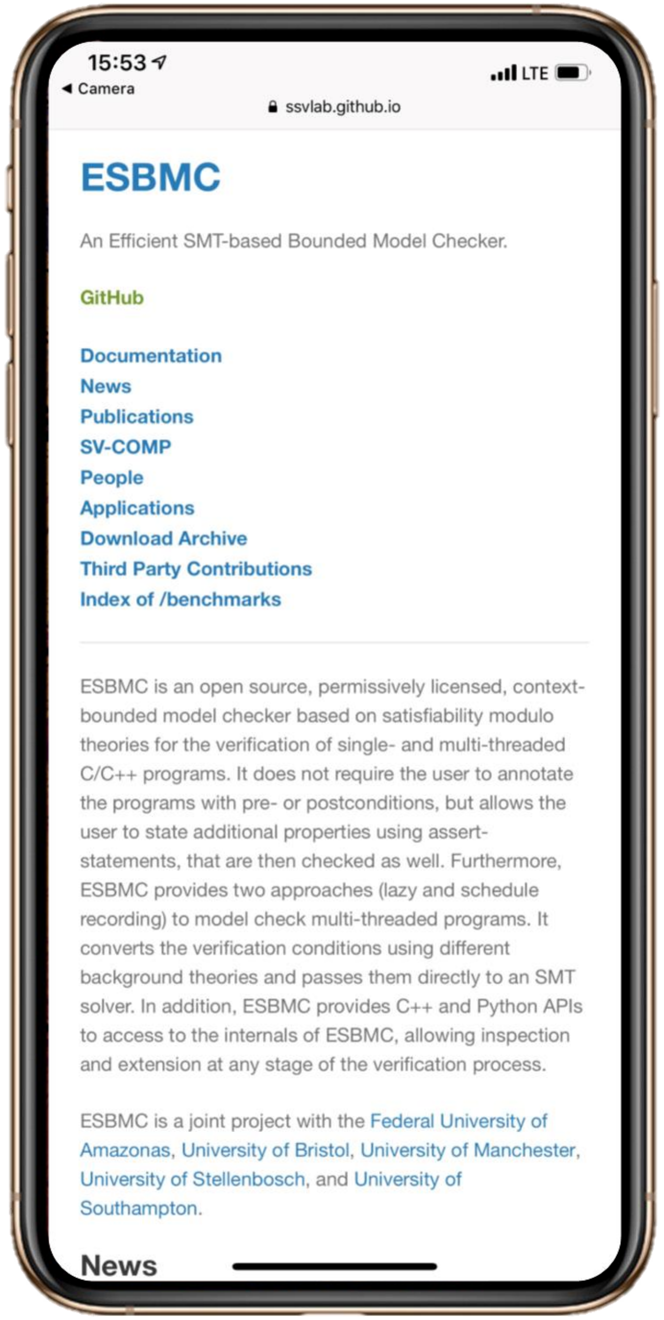
8 `ASSUME s <= 5 && 1 <= s && input <= 5`

Second partial counterexample:

`ASSERT s == 4 && input == 4`

Thank you!

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



Scan me