ESBMC v6.0

Mikhail R. Gadelha
Felipe R. Monteiro
Lucas C. Cordeiro
Denis A. Nicole
ESBMC v6.0
Verifying C Programs Using \( k \)-Induction and Invariant Inference
(Competition Contribution)

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

• 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

Gadelha et al., ASE’18
ESBMC

- SMT-based bounded model checker of single- and multi-threaded C/C++ programs
  - arithmetic under- and overflow
  - pointer safety
  - array bounds
  - division by zero
  - memory leaks
  - atomicity and order violations
  - deadlock
  - data race
  - user-specified assertions

Gadelha et al., ASE’18
ESBMC Architecture

- ESBMC uses a $k$-induction proof rule to verify and falsify properties over C programs
ESBMC Architecture

- ESBMC uses a $k$-induction proof rule to verify and falsify properties over C programs

ESBMC uses clang’s API to access and traverse the program AST, not LLVM bytecode.
ESBMC Architecture

- ESBMC uses a $k$-induction proof rule to verify and falsify properties over C programs

ESBMC Verification Approach

- Strengths & Weaknesses
- Demo
- Contribute!

ESBMC uses a $k$-induction proof rule to verify and falsify properties over C programs. The tool can simplify the input program, for example, calculating `sizeof` expressions and evaluating static asserts.
ESBMC Architecture

- ESBMC uses a $k$-induction proof rule to verify and falsify properties over C programs
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., $a \leq x \leq b$)

- Abstract-interpretation component from CPROVER
- Only for integer variables
ESBMC Architecture

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

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

• **Strengths**

  Reach-Safety
  
  - ECA (score: 1113)
  - Floats (score: 790)
  - Heap (score: 300)
  - Product Lines (score: 787)

  Falsification (score: 1916)
  
  - Arithmetic, floating point arithmetic
  - User-specified assertions
  
  the use of invariants increases the number of correct proofs in ESBMC by about 7%

• **Weaknesses**

  need relational analysis that can keep track of relationship between variables
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) { // unsatisfiable
            __VERIFIER_error(); // property violation
        }
    }
}

Simplified safe 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) { // unsatisfiable
            __VERIFIER_error(); // property violation
        }
    }
}
```

Program under verification
Enable k-induction instead of plain BMC
Enable interval analysis

```
esbmc main.c --k-induction --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) { // unsatisfiable
            __VERIFIER_error(); // property violation
        }
    }
}
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) { // unsatisfiable
            __VERIFIER_error(); // property violation
        }
    }
}
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) { // unsatisfiable
            __VERIFIER_error(); // property violation
        }
    }
}
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) { // unsatisfiable
            __VERIFIER_error(); // property violation
        }
    }
}
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) { // unsatisfiable
            __VERIFIER_error(); // property violation
        }
    }
}
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) { // unsatisfiable
            __VERIFIER_error(); // property violation
        }
    }
}

ASSUME s <= 5 && 1 <= s
ASSUME s <= 5 && 1 <= s
ASSUME s <= 5 && 1 <= s && 6 <= input
ASSUME s == 1 && input == 1
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) { // unsatisfiable
            __VERIFIER_error(); // property violation
        }
    }
}
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) { // unsatisfiable
            __VERIFIER_error(); // property violation
        }
    }
}

ASSUME s <= 5 && 1 <= s
ASSUME s <= 5 && 1 <= s
ASSUME s <= 5 && 1 <= s && 6 <= input
ASSUME s == 1 && input == 1
ASSUME s == 2 && input == 2
ASSUME s == 3 && input == 3
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) {
            // unsatisfiable
            __VERIFIER_error(); // property violation
        }
    }
}
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
        }
    }
}

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

*** Checking inductive step
Starting Bounded Model Checking
Unwinding loop 1 iteration 1 file svcomp2018.c line 17 function main
Not unwinding loop 1 iteration 2 file svcomp2018.c line 17 function main
Symex completed in: 0.001s (82 assignments)
Slicing time: 0.000s (removed 22 assignments)
Generated 1 VCC(s), 1 remaining after simplification (60 assignments)
No solver specified; defaulting to Boolector
Encoding remaining VCC(s) using bit-vector arithmetic
Encoding to solver time: 0.000s
Solving with solver Boolector 3.0.0
Encoding to solver time: 0.000s
Runtime decision procedure: 0.001s
BMC program time: 0.003s

VERIFICATION SUCCESSFUL

Solution found by the inductive step (k = 2)
Falsification

![Graph showing the comparison of different verification tools using accumulated score and time in seconds. The graph includes various tools such as 2LS, CBMC-Path, CBMC, CPA-Seq, DepthK, DIVINE-explicit, DIVINE-SMT, ESBMC-kind, PeSCo, Symbiotic, UAutomizer, UKojak, UTaipan. The x-axis represents the accumulated score, and the y-axis represents the time in seconds. The graph illustrates the performance and speed of each tool in solving falsification problems.]
Thank you!

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