ESBMC 5.0

An Industrial-Strength C Model Checker*

Mikhail R. Gadelha
University of Southampton
United Kingdom

Felipe R. Monteiro
Federal University of Amazonas
Brazil

Jeremy Morse
University of Bristol
United Kingdom

Lucas C. Cordeiro
University of Manchester
United Kingdom

Bernd Fischer
University of Stellenbosch
South Africa

Denis A. Nicole
University of Southampton
United Kingdom

ABSTRACT

ESBMC is a mature, permissively licensed open-source context-bounded model checker for the verification of single- and multi-threaded C programs. It can verify both predefined safety properties (e.g., bounds check, pointer safety, overflow) and user-defined program assertions automatically. ESBMC provides C++ and Python APIs to access internal data structures, allowing inspection and extension at any stage of the verification process. We discuss improvements over previous versions of ESBMC, including the description of new front- and back-ends, IEEE floating-point support, and an improved *k*-induction algorithm. A demonstration is available at `https://www.youtube.com/watch?v=YcJjXHlN1v8`.

CCS CONCEPTS

• **Software and its engineering** → **Formal software verification**; • **Theory of computation** → *Verification by model checking*; • **Hardware** → *Bug detection, localization and diagnosis*;

KEYWORDS

Software model checking; *k*-induction; Bug detection.

ACM Reference Format:
Mikhail R. Gadelha, Felipe R. Monteiro, Jeremy Morse, Lucas C. Cordeiro, Bernd Fischer, and Denis A. Nicole. 2018. ESBMC 5.0: An Industrial-Strength C Model Checker. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3238147.3240481

1 INTRODUCTION

ESBMC is a mature bounded model checking (BMC) tool for multi-threaded C programs. Its development started in 2008 on top of the CProver framework [1], but almost all components have been re-designed and re-implemented in subsequent years, including the basic data structures, front-end, symbolic execution, memory model, and back-end. The purpose of this paper is to describe the recent tool modifications and extensions, including *(i)* a more robust, clang-based [2] frontend; *(ii)* an improved handling of floating-point arithmetics; *(iii)* an improved *k*-induction scheme that allows ESBMC to better handle programs with unbounded loops; and *(iv)* a Python API that gives users access to ESBMC's data structures.

ESBMC primarily aims to help software developers by finding subtle bugs in their code (e.g., array bounds violations, NULL-pointer dereferences, arithmetic overflows, or deadlocks). It does not require any special annotations in the source code to find such bugs, but it does allow users to add their own assertions and also checks for violations of these. In addition, ESBMC implements *k*-induction [3] and can be used to *prove* the absence of property violations (resp. the validity of user-defined assertions). It relies on off-the-shelf satisfiability modulo theory (SMT) solvers such as Boolector, Z3, Yices, MathSAT, and CVC4 to check automatically the verification conditions corresponding to the safety properties.

ESBMC has been applied to a large number of applications including telecommunications, control systems, and medical devices [4]. It is open source (under the terms of the Apache License 2.0) and its source code and self-contained binaries for 64-bit Linux environments are available at https://github.com/esbmc/esbmc/ and www.esbmc.org, respetively.

2 COMPONENTS AND FEATURES

By default, ESBMC takes a C program and checks for array bounds violations, divisions by zero, pointer safety (incl. alignment), and all user-defined properties. It has options to check for overflows, memory leaks, deadlocks and data-races, and to choose between a fixed- or (IEEE) floating-point arithmetic. Figure 1 shows its architecture.

Front-end. ESBMC now uses clang [2], a state-of-the-art compiler suite for C/C++/ObjectiveC/ObjectiveC++ widely used in industry [5], as its front-end. As developers, we thus avoid the need to maintain a separate front-end, but this approach also brings a number of advantages for users: *(i)* ESBMC provides compilation error messages as expected from an industrial-strength tool; *(ii)* ESBMC leverages clang's powerful static analyzer to provide meaningful warnings when parsing the program; *(iii)* clang can simplify the input program (e.g., calculate `sizeof` expressions, evaluate static asserts), which simplifies the analysis of the code. Note that we use clang's API to access and traverse the program AST, without having details of the input program compiled away, which differs from other verifiers (e.g., LLBMC [6]) that rely on the LLVM bytecode.

Control-flow Graph (CFG) Generator. The CFG generator takes the program AST and transforms it into an equivalent GOTO program: a simplified representation that consists only of assignments, conditional and unconditional branches, assumes, and assertions. In particular, this step eliminates all `for`, `while`, `do-while` and `switch` statements. It also adds checks for division by zero

[*] E-mail: `esbmc@googlegroups.com`

ASE '18, September 3–7, 2018, Montpellier, France

ACM ISBN 978-1-4503-5937-5/18/09.
https://doi.org/10.1145/3238147.3240481

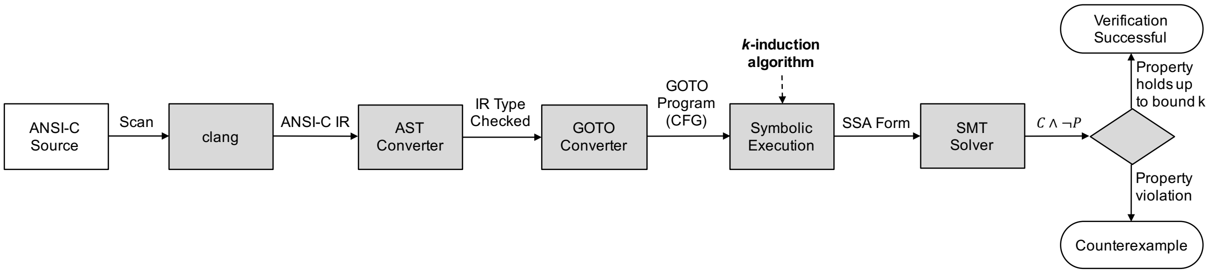

Figure 1: ESBMC architectural overview. The tool takes a C program as input. It then converts the AST generated by clang into a CFG and the symbolic execution engine unrolls the program and generates the SSA form of the program. The SSA is then converted to an SMT formula which is satisfiable only if the program contains errors.

and out-of-bounds access (and for integer and floating-point overflow, if enabled). In k-induction mode (cf. Section 3) it also analyses loop bodies and "havocs" any variable written-to inside a loop with non-deterministic values; the havocked variables are used by the inductive step to over-approximate the loop.

Symbolic Execution Engine. ESBMC then symbolically executes the GOTO program: it unrolls loops k times, generates the static single assignments (SSA) form of the unrolled program, and derives all the safety properties to be checked by the SMT solver. This step also inserts pointer safety checks for dynamically allocated memory, if they are enabled. Note that this can only be done after unrolling because the pointer analysis needs to know the maximum set of dynamically allocated structures. ESBMC aggressively simplifies the program to generate small SSA sets, using constant folding and various arithmetic (including floating-point) simplifications.

SMT back-end. ESBMC's SMT back-end supports five solvers: Boolector (default), Z3, MathSAT, CVC4 and Yices. The back-end is highly configurable and allows the encoding of quantifier-free formulas with support for bitvectors, arrays, tuple, fixed-point and floating-point arithmetic (all solvers), and linear integer and real arithmetic (all solvers but Boolector) . We use the back-end to encode the SSA form of the program into a quantifier-free formula and check satisfiability of $C \wedge \neg P$, where C is the set of constraints and P is the set of properties. If the formula is SAT, the program contains a bug: ESBMC will generate a counterexample with the set of assignments that lead to the property violation.

Python API. ESBMC now includes a Python API that reduces the difficulty of prototyping new features and makes the tool internals accessible to a wider audience, i.e., the verification process shown in Figure 1 can be intercepted and changed at any point. For example, a developer can easily add their own intrinsic function to model a new library function or to exploit a different SMT theory:

```
1 def symex_step(self, art):
2   # Boilerplate accessing instruction 'insn' omitted
3   if insn.type == gptypes.FUNCTION_CALL:
4     call = esbmc.downcast_expr(insn.code)
5     sym = esbmc.downcast_expr(call.function)
6     if sym.name.as_string() == 'c::isnan':
7       # Interpretation of call here
8       return
9   # Otherwise call through to rest of ESBMC
10  super(ThisClass, self).symex_step(art)
```

The code overrides the default instruction interpretation function (l1), and for function-call instructions (l3) aquires the call definition (l4), the name of the called function (l5), and should it be named 'isnan' (l6) applies special handling (l7, details omitted). For all other instructions the default instruction interpretation function is called (l10).

Python is well-known for its expressiveness (e.g., set comprehensions) and the language bindings eliminate the need to consider object lifetime and other low-level details. Rapid prototyping is encouraged by avoiding recompilation of the main tool; it enables new verification ideas to be quickly tested. However, the Python API also has drawbacks—it is slower than C++, and developers can operate it illegally, causing the tool to crash. In the long term, it would thus be desirable to provide ESBMC as a library of verification facilities for the development of new tools.

3 THE *K*-INDUCTION PROOF RULE

k-induction allows BMC to find a property violation or even to prove (partial) correctness without fully unwinding loops. ESBMC uses the algorithm in an iterative deepening style:

$$kind(P,k) = \begin{cases} P \text{ contains a bug,} & \text{if } B(k) \text{ is SAT} \\ P \text{ is correct,} & \text{if } B(k) \wedge [F(k) \vee I(k)] \text{ is UNSAT} \\ kind(P, k+1), & \text{otherwise.} \end{cases}$$

Here, the *base case* formula $B(k)$ is the standard BMC formula, which is satisfiable *iff* the program has a counterexample of length k or less. If all states are reachable for the current k, we know that the program must be correct without checking the inductive step. The *forward condition* $F(k)$ formalizes this; it can be derived from the program by inserting *unwinding assertions* after each loop. This step is particularly useful for proving safety in the presence of bounded loops. The *inductive step* $I(k)$ checks that, if a safety property holds in the first k steps, then it also holds for $k+1$ steps. Iterative deepening implies that ESBMC always finds the smallest k to either prove correctness or find a property violation.

ESBMC now uses scheme improved over the earlier version described by Gadelha et al. [3]. In particular, this new version no longer collects havocked variables into states, rewriting every access to these variables into state accesses. Instead, the havocked variables are directly assigned nondeterministic values in the inductive step. This is a simpler and more accurate transformation and follows the work by Donaldson et al. [7]. Note, however, that their implementation works by unwinding the program during CFG generation, thus replicating the loop k times and removing the backward jump. In ESBMC, the algorithm only adds the nondeterministic assignments before the loop and the required assumptions [3] during CFG generation, and relies on the symbolic execution to unroll the loop.

4 FLOATING-POINT ENCODING IN ESBMC

In previous versions, ESBMC verified programs using a fixed-point arithmetic [4]; this is appropriate for, e.g., programs running in a number of embedded devices, but not for programs that rely on floating-point arithmetic. The current version of ESBMC encodes floating-point arithmetic either using the SMT theory of floating-points, fully available in Z3 and, partially, in Mathsat and CVC4, or using bitvectors, which extends the floating-point arithmetic support (except for floating-point exceptions) to all solvers that are currently integrated. This is a major improvement over our prior work [8], where we model most of the C11 standard functions [9].

Currently, MathSAT does not support `fp.rem` (remainder operator) and `fp.fma` (fused multiply-add) and CVC4 does not support sort conversion functions. Our SMT backend falls back to the bitvector mode when an unsupported operation needs to be encoded; it converts the arguments from floating-points to bitvectors, encodes the operation and returns the resulting bitvector encoded as a floating-point. This is only possible by using the (non-standard) SMT-LIB functions `fp_as_ieeebv` and `fp_from_ieeebv` to convert to and from bitvectors. This process is transparent to the user and effectively means that missing operations will be correctly encoded, despite the lack of support by the underlying solver; it allows us to use SMT solvers as Boolector and Yices that do not have any built-in floating-point theory. Most of the floating-point operations in ANSI-C programs can be directly converted to SMT; only two operations needed special handling:

Cast to Boolean. The SMT standard does not define conversions between Boolean and floating-point types. In ESBMC, when casting from Booleans to floating-points, an `ite` operator is used, such that the result of the cast is 1.0 if the Boolean is true; otherwise it is 0.0. We encode casts from floating-points to Booleans as conditional assignments: the cast result is true when the floating is not 0.0; otherwise it is false.

Equality. Bitvector assignment and equality operations are encoded using the equality operator (==). However, the SMT standard defines a separate operator for floating-point equalities, the `fp.eq` operator, where "(fp.eq x y) evaluates to true if x evaluates to -zero and y to +zero, or vice versa. fp.eq and all the other comparison operators evaluate to false if one of their arguments is NaN".

The operator is defined to handle the special symbols from the IEEE floating-point standard, in particular, signaled zeros and NaNs; for this reason, ESBMC encodes all equality of floating-points using the `fp.eq` operator, while assignments remain encoded using the equality operator.

5 ILLUSTRATIVE EXAMPLE

We describe how to verify a C program with ESBMC using the code fragment shown in Fig. 2. Here, ESBMC is invoked as follows:

```
esbmc <file>.c --floatbv --k-induction
```

where `<file>.c` is the C program to be checked, `--floatbv` indicates that ESBMC will use floating-point arithmetic to represent the program's `float` and `double` variables, and `--k-induction` selects the k-induction proof rule. The user can select the SMT solver, property, and verification strategy; `esbmc --help` provides the full list of options.

ESBMC unrolls the program in Fig. 2 and converts it into SSA form, to produce verification conditions (VCs), one for each assertion that can not be statically determined. Equations (1) and (2) give C and P during the inductive step ($k = 2$).

```
 1 #include<math.h>
 2 int main() {
 3   unsigned int N = nondet_uint();
 4   double x = nondet_double();
 5   if(x <= 0 || isnan(x))
 6     return 0;
 7   unsigned int i = 0;
 8   /*i = nondet_uint();*/                ⎫
 9   /*x = nondet_double();*/              ⎬inductive step only
10   /*__ESBMC_assume(i < N);*/            ⎭
11   while(i < N) {
12     x = (2*x);
13     assert(x>0);
14     ++i;
15   }
16   /*__ESBMC_assume(!(i < N));*/}inductive step only
17   assert(x>0);
18   return 0;
19 }
```

Figure 2: C code fragment. Here `nondet_uint()` **and** `nondet_double()` **stand for non-deterministic integer and double values, respectively.** `isnan` **checks whether a given floating-point is a not-a-number (NaN) value. The commented lines 8-10 and 16 are the transformations introduced during the inductive step of the** k**-induction algorithm.**

Note that $isnan(x_1)$ in Eq. (1) checks if the symbol is a NaN, which is translated to the `fp.isNaN` SMT operator. ESBMC also adds additional literals for each clause of P to identify the respective VCs. The resulting formula $C \wedge \neg P$ is then passed to an SMT solver, where it is checked in less than one second. We can also introduce a bug by removing the $isnan(x)$ check from line 5 in Fig. 2, (effectively removing g_2 from Eq. (1)) which would lead to the counterexample in the right-hand side. Here, state S_3 leads to an assertion failure in line 13 (*i.e.*, if $x = -NaN$, then $x > 0.0f$ evaluates to false); ESBMC is also able to detect this violation in less than one second.

$$\begin{array}{lcl} S_1 & \mapsto & N = 0 \\ S_2 & \mapsto & x = -NaN \\ S_3 & \mapsto & x > 0.0f \end{array}$$

$$C := \left[\begin{array}{l} N_1 = nondet\_uint_1 \\ \wedge\ x_1 = nondet\_double_1 \\ \wedge\ g_1 = (x_1 \leq 0.0f) \\ \wedge\ g_2 = ite\,(g_1, \text{TRUE}, isnan(x_1)) \\ \wedge\ i_1 = nondet\_uint_2 \\ \wedge\ x_2 = nondet\_double_2 \\ \wedge\ \neg g_2 \implies i_1 < N_1 \\ \wedge\ g_3 = (i_1 < N_1) \\ \wedge\ x_3 = 2.0f * x_2 \\ \wedge\ \neg g_2 \wedge g_3 \implies x_3 > 0.0f \\ \wedge\ i_2 = i_1 + 1 \\ \wedge\ g_4 = (i_2 < N_1) \\ \wedge\ x_4 = 2.0f * x_3 \\ \wedge\ x_5 = ite\,(\neg g_4, x_3, x_4) \end{array} \right] \quad (1) \qquad P := \left[\begin{array}{l} \neg g_2 \wedge g_3 \wedge g_4 \\ \qquad \implies x_4 > 0.0f \\ \wedge\ \neg g_2 \\ \wedge\ (g_3 \wedge \neg g_4 \vee \neg g_3) \\ \qquad \implies x_5 > 0.0f \end{array} \right] \quad (2)$$

Fig. 2 also shows the transformations introduced by the inductive step, during k-induction; the transformations are commented out during the base case and the forward condition. The transformations aim to prove that no property violation is reachable, regardless of loop unwinding; this is translated to assuming nondeterministic values to the loop variables (lines 8-9), assuming that the loop body is evaluated (line 10, the assumption removes every state that does not satisfy the loop entry condition) and terminates (line 16, the assumption removes every state that does not satisfy the loop termination condition). Note that, since the inductive step is an overapproximation, only correctness can be proved, i.e., it might find spurious counterexamples.

	2LS	CBMC	CPA-Seq	DepthK	ESBMC v1.25.2	ESBMC v5.0	Symbiotic	UAutomizer	UKojak	UTaipan
Correct true	1898	1438	3790	1184	1957	2822	1418	3902	1725	2292
Correct false	1426	1856	2598	1516	1476	1494	1209	1278	514	563
Incorrect true	2	2	0	19	336	14	1	2	0	3
Incorrect false	5	3	4	37	92	10	0	0	0	3
Total correct results	3324	3294	6388	2694	3433	4316	2627	5180	2239	2855
Total incorrect results	7	5	4	58	428	24	1	2	0	6

Table 1: Results from SV-COMP 2018.

6 EXPERIMENTAL EVALUATION

We evaluated ESBMC over the SV-COMP [10] benchmarks, which comprises 9523 verification tasks that check for property reachability (2941 tasks), memory safety (326 tasks), reachability in concurrent programs (1047 tasks), overflow (358 tasks), termination (2009 tasks) and reachability in Linux device drivers (2842 tasks). Table 1 shows our experimental results; a detailed description of the different tools and the experimental setup can be found in [10]. Here, a task counts as *correct true* if it does not contain any reachable error location or assertion violation, and the tool reports "safe"; however, if the tool reports "unsafe", it counts as *incorrect true*. Similarly, a task counts as *correct false* if it does contain a reachable violation, and the tool reports "unsafe", together with a confirmed witness (path to failure); otherwise, it counts as *incorrect false* accordingly. The difference between the grand total (9523) and the sum of the two sub-totals gives the number of tasks for which the tool exhausted time or memory, or failed otherwise.

***k*-induction.** Overall, ESBMC ranked third, behind CPA-Seq and UAutomizer, with 14 incorrect false results (10 due to inaccuracies in our concurrency and memory models, plus 4 due to bugs in the simplifier), and 10 incorrect true results. However, none of the incorrect results are related to the k-induction algorithm, and the results show that ESBMC is currently the best k-induction tool. ESBMC outperformed CPA-Seq and UAutomizer in the verification of reachability properties for arrays and memory safety issues.

CBMC also implements k-induction, requiring three different calls: to generate the CFG, to annotate the program and to verify it, whereas ESBMC handles the whole process in a single call. Additionally, CBMC does not have a forward condition to check if all states were reached and relies on a limited loop unwinding [7].

CPA-Seq applies a number of different techniques when verifying a program, so a direct comparison to their k-induction is not possible; however, a "pure k-induction" version (CPA-kind [11]) showed poor results in a previous competition.

DepthK uses an invariant generator to instrument the code with invariants and uses k-induction to verify the program [12]. Although one would expect better results, DepthK uses an old version of ESBMC to verify the programs; this explains the poor results.

2LS integrates an abstract interpretation invariant generation between the base case and the inductive step; in contrast to our k-induction, their version has no forward condition. Overall 2LS verifies 22% fewer benchmarks than ESBMC (2LS returns 33% fewer correct results), although it also returns fewer incorrect results.

Comparison with old *k*-induction schema. In order to compare with the old k-induction schema used in ESBMC v1.25.2 [3], we re-ran it over the current SV-COMP benchmark set. Table 1 shows a 25% increase in correct results and a 95% decrease in incorrect results, but this is slightly misleading: 168 of the incorrect results produced by the old scheme are from the concurrency category and are caused by the concurrent model used back then. However, the old scheme also produces incorrect results in the ReachSafety-ECA (95) and ReachSafety-Recursive (28) categories, which are related to its incorrect approximation of loop termination conditions.

Floating-point verification. ESBMC uses MathSAT for tasks that involve floating-point arithmetics. This combination not only outperforms a Z3-based ESBMC, but also all other tools in SV-COMP. ESBMC achieved the highest score in the ReachSafety-Floats subcategory where it can verify 84% of the tasks (145 out of 172) within the time and memory restrictions.

7 CONCLUSION AND FUTURE WORK

We presented ESBMC, the first open-source SMT-based context-bounded model checker to support full C programs [4, 13]. ESBMC is a mature tool; here, we focussed on three novel features of ESBMC v5.0: the new clang front-end, the new floating-point back-end and, in particular, our new implementation of the k-induction proof rule. Results over the SV-COMP 2018 benchmark suite show that ESBMC is the strongest k-induction tool currently available. We are extending the k-induction proof rule to use information from the inductive step, to make bug finding more efficiently [14].

REFERENCES

[1] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," *TACAS*, LNCS 2988, 2004, pp. 168–176.

[2] B. C. Lopes and R. Auler, *Getting Started with LLVM Core Libraries.* Packt Publishing, 2014.

[3] M. Y. R. Gadelha, H. I. Ismail, and L. C. Cordeiro., "Handling loops in bounded model checking of C programs via k-induction," *STTT*, vol. 19, no. 1, pp. 97–114, 2017.

[4] L. C. Cordeiro, B. Fischer, and J. Marques-Silva, "SMT-based bounded model checking for embedded ANSI-C software," *IEEE Transactions on Software Engineering*, vol. 38, no. 4, pp. 957–974, 2012.

[5] C. Metz, "Why Apple's swift language will instantly remake computer programming," 204, http://www.wired.com/2014/07/apple-swift/.

[6] F. Merz, S. Falke, and C. Sinz, "LLBMC: Bounded model checking of C and C++ programs using a compiler IR," *VSTTE*, LNCS 7152, 2012, pp. 146–161.

[7] A. Donaldson, L. Haller, D. Kroening, and P. Rümmer, "Software verification using k-induction," *SAS*, 2011, pp. 351–368.

[8] M. Y. R. Gadelha, L. C. Cordeiro, and D. A. Nicole, "Encoding floating-point numbers using the SMT theory in ESBMC: An empirical evaluation over the SV-COMP benchmarks," *SBMF*, 2017, pp. 91–106.

[9] R. Smith, *Working Draft, Standard for Programming Language C++*, 2016, [Online; accessed January-2017].

[10] SoSy-Lab, "SV-Comp 2018," https://sv-comp.sosy-lab.org/2018/, 2018, [Online; accessed January-2018].

[11] D. Beyer, M. Dangl, and P. Wendler, "Boosting k-induction with continuously-refined invariants," *CAV*, LNCS 9206, 2015, pp. 622–640.

[12] W. Rocha, H. Rocha, B. Fischer, and L. C. Cordeiro, "Depthk: A k-induction verifier based on invariant inference for C programs - (competition contribution)," *TACAS*, 2017, pp. 360–364.

[13] L. C. Cordeiro and B. Fischer, "Verifying multi-threaded software using SMT-based context-bounded model checking," *ICSE*, 2011, pp. 331–340.

[14] M. Y. R. Gadelha, L. C. Cordeiro, and D. A. Nicole, "Counterexample-guided k-induction verification for fast bug detection," *CoRR*, vol. abs/1706.02136, 2017.