ESBMC 5.0: An industrial-strength C model checker

ESBMC 5.0: An industrial-strength C model checker

Mikhail Ramalho*, Felipe R. Monteiro†, Jeremy Morse‡, Lucas Cordeiro§, Bernd Fischer¶ and Denis Nicole*
*University of Southampton, ‡University of Bristol, §University of Manchester, United Kingdom
†Federal University of Amazonas, Brazil, ¶University of Stellenbosch, South Africa
esbmc@googlegroups.com

***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 safety (e.g., bounds check, pointer safety, overflow) and user-defined (as asserts in the program) properties 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=N_qmN-cazvM`**.**

I. 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 The purpose of this paper is to describe the recent (but yet unpublished) tool modifications and extensions, including:

- a more robust, clang-based [2] frontend;
- an improved handling of floating-point arithmetics;
- an improved k-induction scheme that allows ESBMC to better handle programs with unbounded loops; and
- a Python API that gives users easy access to ESBMC's internal data structures.

ESBMC primarily aims to help software developers by finding subtle bugs in their code (e.g., array bounds violations, nil-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 therefore 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 fully automatically check the verification conditions corresponding to the safety properties.

ESBMC has been applied to a large number of applications from 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.

II. 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 also 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 the tool architecture.

1) 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. This avoids the need to maintain a separate front-end and allows us to focus on our main objective, software verification. Specifically, ESBMC uses clang's API to convert clang's AST into its own AST. ESBMC thus differs substantially from other software verifiers such as LLBMC [6] that use the LLVM bytecode. Note that the C++ elements of the clang AST are not yet fully integrated with ESBMC; it does not yet support polymorphism and some other minor features. This requires some bug fixes in clang, for which we already submitted patches.

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

3) 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 objects. ESBMC aggressively simplifies the program to generate small SSA sets, using constant folding and various arithmetic (including floating-point) simplifications.

4) 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 encoding quantifier-free formulas with support for bitvectors, arrays, tuple, and fixed-point arithmetic (all solvers), linear integer and real arithmetic (all solvers but Boolector) and floating-point arithmetic (all solvers).We use the back-end to encode the SSA form of the program into a quantifier-free formula and check satisfiability of $\mathcal{C} \wedge \neg \mathcal{P}$, where $\mathcal{C}$ is the set of constraints

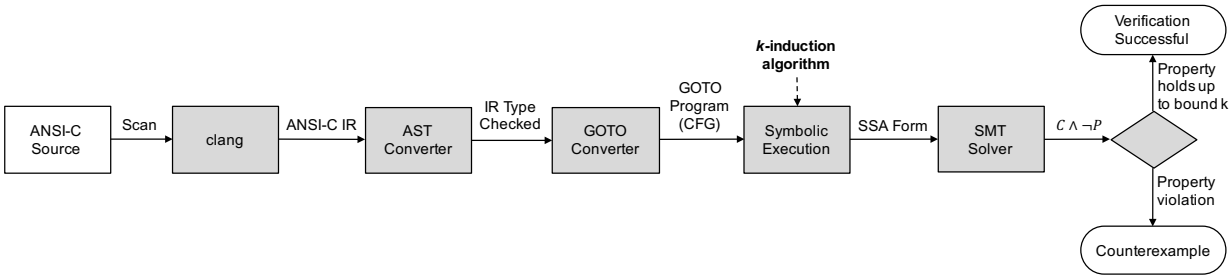

Fig. 1: ESBMC architectural overview. The tool takes a C program as input. It then converts the AST generated by clang into a CFG and the symbolic execution engine unrolls the program and generates the SSA form of the program. The SSA is then converted to an SMT formula which is satisfiable only if the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

and $\mathcal{P}$ is the set of properties. If the formula is satisfiable, then the program contains a bug: ESBMC will generate a counterexample with the set of assignments that lead to the property violation.

5) 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. 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 rapidly tested. For example, a developer can easily add their own intrinsic function to model a new library function or to exploit a different SMT theory:

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

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.

III. The k-induction Algorithm in ESBMC

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

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

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. The *forward condition* $F(k)$ checks whether all program states were reachable for the current k. 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 an improved scheme of 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 the implementation by Donaldson et al. works from the outermost loop inwards and downwards, so that the creation of copies of the loop body during unrolling will replicate nested loops, requiring further loop unrolling of the nested loops. The implementation should keep track of the newly created loops to correctly replicate them, but does not do that. Our implementation avoids this problem by working in the opposite direction (from the inner nested loops outwards).

Property-directed reachability (PDR), which allows refining invariants from the counterexamples, has been proposed as a better alternative to k-induction. However, PDR is difficult to implement robustly, and there are only few PDR-based verifiers for C programs (e.g., SeaHorn [8]), which in practice do not perform as well as expected.

IV. Floating-point Encoding in ESBMC

In previous versions, ESBMC verified programs using a fixed-point arithmetic; 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 [9], where we model most of the C11 standard functions [10].

Currently, MathSAT does not support `fp.rem` (remainder operator) and `fp.fma` (fused multiply-add) and CVC4 does not support the conversion to other sorts operators; 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 like Boolector 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:

1) 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.

2) Equality: Bitvector assignments and equalities 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 being encoded using the equality operator.

V. Illustrative Example

Here, we describe how to verify a C program with ESBMC using the code fragment shown in Fig. 2. For this particular program, 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 variables, and `--k-induction` selects the k-induction proof rule strategy. ESBMC has many options to customize the verification process (e.g., SMT solver, property, verification strategy); `esbmc --help` provides the full list.

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. The resulting formula $\mathcal{C} \wedge \neg \mathcal{P}$ is then passed to an SMT solver to check for satisfiability; in our running example, the resulting formula is correctly checked in less than one second. We can also introduce a bug by removing the $isnan(x)$ check from line 5 in Fig. 2, which would lead to this counterexample:

$$\begin{aligned} S_1 &\mapsto N = 2147483648 \\ S_2 &\mapsto x = +NaN \\ S_3 &\mapsto x = +NaN \\ S_4 &\mapsto x > 0.0f \end{aligned}$$

Here, state S_4 leads to an assertion failure in line 9 (*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.

```
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   for(unsigned int i = 0; i < N; ++i) {
8     x = (2*x);
9     assert(x>0);
10  }
11  assert(x>0);
12  return 0;
13 }
```

Fig. 2: Illustrative 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.

VI. Experimental Evaluation

1) Setup: We evaluated ESBMC over the benchmarks from the latest SV-COMP [11], which contains 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).

The experiments were conducted on a computer with an Intel Xeon E3-1230 v5 CPU running at 3.40GHz and 33GB of RAM under an x86_64-linux operating system (Ubuntu 16.04, Linux kernel 4.4). The verification time limit was set to 900s (CPU) and memory usage limited to 15 GB.

Table I shows our experimental results; a detailed description of the different tools can be found in [11]. A task counts as *correct true* (resp. *false*) if it does not (resp. does) contain any reachable error location or assertion violation, and the tool reports "safe" (resp. "unsafe"), together with an appropriate witness (proof resp. counterexample); otherwise, it counts as *incorrect true* (resp. *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.

2) 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 respectively, and 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.

CBMC also implements k-induction, but the process is more onerous on the user, requiring three different calls to CBMC: 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 technique [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-indcution" version (CPA-kind, [12]) showed poor results in a previous competition.

	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 I: Results from SV-COMP 2018.

DepthK uses an invariant generator to instrument the code with invariants and uses k-induction to verify the program [13]. Although one would expect better results, DepthK uses an old version of ESBMC to verify the programs; this explains the poor results.

2LS implements an algorithm called kIkI (k-invariants and k-induction), which 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 (in particular, 2LS returns 33% fewer correct results), although it also returns fewer incorrect results.

3) Comparison with old k-induction schema: In order to compare with 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 I 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 the k-induction algorithm, specifically its incorrect approximation of loop termination conditions.

4) 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.

Although CBMC was the first verifier to support bit-precise verification of C programs that use floating-point arithmetic, it is less efficient than ESBMC. In particular, it can sometimes take a very long time to produce an SMT formula; for example, the SV-COMP benchmarks *floats-esbmc-regression/*nondet*.c* are less than 30 lines long, but the latest CBMC version (v5.8) takes hours to convert the derived VC into SMT format,[1] while ESBMC can verify them in a few seconds. Regarding the SMT backend, ESBMC provides a superior alternative to CBMC, which generates the SMT formula in a file and externally calls the solvers, whereas ESBMC uses the solvers' native APIs. In [4], we explain the difference in performance using both approaches (using API and file interfaces). Additionally, the SMT backend of CBMC is unable to support full ANSI-C, as recently reported in [14].

[1]https://github.com/diffblue/cbmc/issues/1944

VII. Conclusion and Future Work

We have presented ESBMC, the first open-source SMT-based context-bounded model checker to support full C programs [4], [15]. ESBMC is a mature tool; here, we focussed on three three novel features of the latest version 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 currently extending the *k*-induction proof rule to use information from the inductive step, to make bug finding more efficiently [16].

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] A. Gurfinkel, T. Kahsai, and J. A. Navas, "Seahorn: A framework for verifying C programs (competition contribution)," *TACAS*, LNCS 9035, 2015, pp. 447–450.

[9] 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.

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

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

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

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

[14] H. F. Albuquerque, R. F. Araujo, I. V. de Bessa, L. C. Cordeiro, and E. B. de Lima Filho, "Optce: A counterexample-guided inductive optimization solver," *SBMF*, LNCS 10623, 2017, pp. 125–141.

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

[16] 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.