Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

ESBMC 6.1: Automated Test Case Generation using Bounded Model Checking

Mikhail R. Gadelha[1], **Rafael S. Menezes**[2], **Lucas C. Cordeiro**[3]

[1] Sidia Instituto de Ciência e Tecnologia, Manaus, Brazil
e-mail: `mikhail.gadelha@sidia.com`
[2] Electronic and Information Research Center, Federal University of Amazonas, Brazil
[3] School of Computer Science, University of Manchester, UK

The date of receipt and acceptance will be inserted by the editor

Abstract. ESBMC is an SMT-based bounded model checker that provides a bit-precise verification of both C and C++ programs. Bounded Model Checking (BMC) was developed to provide faster results when finding property violations; BMC achieves this by limiting the number of loop unwindings and recursion depth. The technique, however, is unable to prove correctness unless all loops and recursions are fully unwound, which might not be possible for some programs (e.g., infinite loops). The version of ESBMC described here is designed to avoid the problem of guessing the number of unwindings, which leads to a property violation; it incrementally verifies the program, searching only for property violations. Once ESBMC has found a property violation, it produces a test suite that contains at least one test to expose a bug. ESBMC can correctly produce 312 test cases, which are confirmed by the test validator employed by Test-Comp 2019.

1 Overview

ESBMC is an SMT-based context-bounded model checker that can verify single- and multi-threaded C programs [1]. The current ESBMC submission is configured to run in incremental bounded model checking mode, which will incrementally verify the benchmark, looking only for property violations with the goal of producing a test suite; this represents a novel application of ESBMC since we are now able to automatically generate a test suite for single-threaded C programs, which can expose specific bugs related to reachability, memory safety, and user-specified assertions.

There exist some related studies that describe how to apply bounded model checking (BMC) tools to generate test cases automatically [2,3,4]. Petrov et al. [2] implement a unit test generation tool using a working BMC solution called Borealis. The authors evaluated their prototype over a set of simple test programs. They showed that their solution allows software developers to achieve adequate coverage based on the generated test suite. However, their prototype does not support dynamic memory or intraprocedural effects, which are standard features found in C programs. Anielak et al. [3] describe an approach to incrementally produce test cases using the C Bounded Model Checker (CBMC) [5]. In particular, the authors focus on the task of rating solutions to a programming exercise, namely automatic rating, and present positive experimental results to confirm a substantial increase in rating accuracy compared with rating based on manually designed test cases. Rocha et al. [4] propose the closest approach to the one described here. The authors present a method to automatically generate and check memory management test cases for unit tests based on assertions extracted from safety properties automatically produced by BMC tools.

Our competition entry ESBMC is written in C++ and uses clang [6] as its front-end. ESBMC supports various SMT solvers as back-ends (Boolector [7], Z3 [8], Yices [9], Mathsat [10] and CVC4 [11]) and for this submission ESBMC uses Boolector v3.0.0 to check satisfiability.

2 Test-Generation Approach

Here we discuss (incremental) bounded model checking and our approach to the competition: an incremental BMC model focused only on finding bugs, named *falsification* mode.

The BMC technique was developed to reduce the complexity that a system presents to a model checker; otherwise, it might need an infinite amount of resources (e.g., processor, memory, time) to complete the verification of all possible states [12]. BMC tools drop completeness (i.e., the ability to prove that a program does not contain a bug) in favor of falsification [13]. They are used mainly to find bugs, as they are only able to prove the absence of bugs if the whole state space is explored (e.g., all loops have been fully unwound). BMC tools have already been applied successfully in the verification of real-world software, including software written in languages usually used for low- and medium-level development, such as C [1,5,14] and C++ [15].

When running a BMC tool, one usually has to specify a bound k explicitly; this will be used to limit the visited regions of data structures (e.g., arrays) or the number of recursions and loop iterations. This limits the state space to be explored during verification, leaving enough so that real errors in applications can be found [5,15,16]; BMC tools are still susceptible to exhaustion of time or memory limits for programs with loops whose bounds are too large, but unlike unbounded model checking tools, this can be prevented with a smaller value of *k*.

The loop unwinding process is essential to the BMC technique, as it is responsible for restricting the state space explored by the algorithm. Given a bound k, it removes states from the verification, which are only reachable for unwindings $> k$, while preserving the program behavior for loop unwindings $\leq k$. This is done by using either *unwinding assumptions* that symbolically encode state-space constraints or *unwinding assertions* that can produce counterexamples to indicate that the state space was not fully explored.

An unwinding assertion prevents a BMC tool from presenting a false negative result; it detects if the states beyond the loop bound were not evaluated. An unwinding assumption removes states beyond the loop, and if ill-chosen may remove all states, leading to false safety claims. In summary, an unwinding assertion/assumption is an assertion/assumption where the condition is the loop termination condition.

Formally, given a state transition system of a program P unwound k times, a BMC procedure can be formulated as:

$$\pi = I(s_0) \wedge \bigwedge_{i=0}^{k-1} tr(s_i, s_{i+1}) \wedge \bigvee_{j=0}^{k} \neg\phi(s_j) \tag{1}$$

where a predicate $I(s_0)$ denotes that s_0 is the initial state of the unwound program P, a transition $tr(s_i, s_{i+1})$ is a transition from s_i to s_{i+1}, $\phi(s_j)$ is a safety property. If the formula is satisfiable then there is a sequence of states that triggers a property violation; the sequence of states found by the decision procedure is called counterexample or witness.

Finding a good value of k, i.e., the minimum number of unwinding to find a property violation, is a challenging task. BMC tools can try to statically determine a value of *k* that will fully unwind all the loops; if such a value is found, there is no need for unwinding assertions, however, determining this value is not always possible (e.g., for infinite loop programs or loop conditions that depend on user input).

Incremental BMC was proposed to avoid this pitfall [17]. In an incremental BMC, the program is incrementally unwound until a bug is found or until the completeness threshold [18] is reached, ensuring that smaller problems are solved sequentially instead of guessing an upper bound for the verification. The incremental algorithm, however, has its limitations. In particular, the BMC has to redo all the parsing, generation and solving for each bound *k*, and no record of previous step *1* to *k-1* is used when solving for *k*. Even though incremental solving first appeared in the 1990s [17], the problem of how to efficiently reuse information learned from previous instances remains.

```c
int main() {
  unsigned int M = __VERIFIER_nondet_uint();
  int A[M], B[M], C[M];
  unsigned int i;
  for(i=0;i<M;i++) A[i] = __VERIFIER_nondet_int();
  for(i=0;i<M;i++) B[i] = __VERIFIER_nondet_int();
  for(i=0;i<M;i++) C[i]=A[i]+B[i];
  for(i=0;i<M;i++) __VERIFIER_assert(C[i]==A[i]-B[i]);
}
```

Fig. 1. C program extracted from Test-Comp 2019.

```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>1</input>
  <input>0</input>
  <input>2147483646</input>
</testcase>
```

Fig. 2. Generated test case in XML.

Our new falsification approach uses an incremental BMC approach to find property violations. Intuitively, we aim to find a counterexample π with up to k loop unwindings. This approach replaces all unwinding assertions (e.g., assertions to check if a loop was completely unwound) with unwinding assumptions. Normally, this would lead to unsound behavior but, since the falsification algorithm cannot provide correctness validation, it will not affect the search for bugs. This approach is focused on bug finding and does not care if a loop was not completely unwound; it only cares if the number of unwindings will lead to a property violation.

The falsification approach in ESBMC also offers the option to change the granularity of the increment; the default value is 1, but can be increased to meet any expected behavior. Note that changing the value of the increment can lead to slower verification time and might not present the shortest counterexample possible for a property violation.

When a property violation is found, a witness [19] is produced: it contains one path in the program from the entry point to the property violation, with all non-deterministic assignments in that path replaced by concrete values found by the underlying SMT solver. From the witness, we derive the files required by the competition: a metadata file which specifies information about the tool and the verified program, and one or more test case files, which specify the values to non-deterministic assignments that lead to a property violation. Our test-generation approach is focused on the ability to discover bugs. In particular, we produce a test suite,[1] which contains at least one test that exposes the bug. As an example, Figure 1 shows a program extracted from the *ReachSafety-Loops* subcategory.[2] ESBMC produces a test case illustrated

[1] We produce test suites that adhere to the exchange format described in `https://gitlab.com/sosy-lab/software/test-format/blob/master/doc/Format.md`

[2] `https://raw.githubusercontent.com/sosy-lab/sv-benchmarks/master/c/loops/sum_array-1.c`

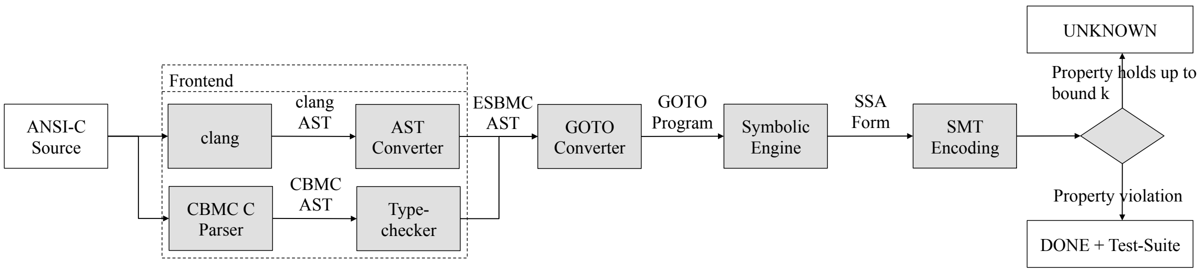

Fig. 3. ESBMC architecture customized for automated test-case generation.

in Fig. 2 where $M = 1$, $A[0] = 0$, $B[0] = 2147483646$, which violates the `__VERIFIER_assert` in the program.

3 Architecture

Fig. 3 shows the current architecture of ESBMC customized for the automated test-case generation of C programs; white rectangles represent input and output while grey rectangles represent the verification steps. The tool is composed of several modules, each with a specific goal.

Front-end. Converts the program into an abstract syntax tree (AST) [20]. Currently, ESBMC has three front-ends as follows: one clang-based [21] C front-end, one CBMC-based C front-end and one CPROVER-based C++ front-end [5]. In Figure 3, the C++ front-end is omitted but it works similarly to the C front-end, with a parser [22] and a type-checker [23].

GOTO converter. Converts the AST generated by the front-end into a state transition system called GOTO program. In this module, the GOTO program can be changed to add property checks and *k*-induction specific instructions.

Symbolic Engine. Converts the GOTO program into a sequence of static single assignments (SSA) [24]. This module unwinds the loops of the GOTO program [25], propagating constants to generate a minimal set of SSAs. This module can also add property checks, most of them related to dynamically allocated memory.

SMT encoding. Converts the set of SSAs into SMT and then checks for satisfiability. If the formula is satisfiable, then the SMT solver is queried for relevant information to build the respective counterexample. ESBMC supports five state-of-the-art SMT solvers: Z3 [8], MathSAT [10], Boolector [7], Yices [9], and CVC4 [11].

ESBMC produces, as a result, a tuple (answer, test-suite). If ESBMC has found a property violation, then it produces “DONE + Test-Suite”, i.e., the returned test suite tries to expose the bug as defined in the test spec.; otherwise, ESBMC returns “UNKNOWN”, i.e., it does not succeed in computing a test suite due to a crash, time-out, or out-of-memory.

4 Strengths and Weaknesses

The falsification mode in ESBMC is a simple but effective approach to find property violations. The program is bit-precise encoded into SMT and all property violations up to a given bound k are checked in conjunction. The big advantage of this approach is that we can jointly check all paths in one call to the solver, instead of exploring and encoding them separately. The falsification mode in ESBMC can find property violations in a large number of programs, thereby strengthening our claim that this is an effective approach to test generation. Despite its effictiveness, however, due to bugs in our test case generation, only a fraction of the results were validated.

One drawback of the falsification mode is that coverage test generation support is non-trivial: some researches proposed solutions for these issues with different degrees of success [2] but ESBMC has yet to implement a solution. Secondly, our approach generates the full set of SSA from scratch for every new unwinding: one would expect that using incremental SMT solving offers better results but empirical experiments show otherwise, as described by Henning et al. [26].

4.1 Results

When running in falsification mode, ESBMC can correctly produce 312 test cases, which are confirmed by the test validator employed by Test-Comp 2019. Our remaining results are divided into 194 timeouts or memory outs, 136 unconfirmed test cases and 4 unknown results. As expected, falsification provides no correctness claim.

In particular, the unconfirmed test cases were mainly due to our approach of converting the ESBMC witnesses into test cases, which does not convert all generated values of the C primitive data-types into the test case format as expected by the test validator employed by Test-Comp 2019; in particular, our test case converter does not handle structs and unions correctly. Out of the 136 unconfirmed, 4 were because ESBMC did not generate a witness file, 17 were due to the witness not containing all variable values and 115 were due to wrong encoding of primitive data-types (e.g., *booleans* and *floats*).

Most failures happened in the *ReachSafety Floats* (24) and *ReachSafety Sequentialized* (88) categories; these categories contain massive use of *float* and *booleans* variables, which led to various failures when validating our test cases. Our best results are concentrated in the *ReachSafety ECA*; we did not produce any invalid result, and thus we were able to validate 251 test cases.

5 Tool Setup and Configuration

We need to set the architecture (32 or 64 bits), the competition strategy, the property file path, and the benchmark path, to our new wrapper script as follows:

```
esbmc-wrapper.py [-h] -a {32,64}
-p PROPERTYFILE -s falsi BENCHMARK
```

`-a` sets the architecture, `-p` sets the property file path, `-s` sets the strategy (in this case, `falsi` for falsification) and `BENCHMARK` is the benchmark path.

When using the falsification mode, the following options are set for every property: `--no-div-by-zero-check`, which disables the division by zero check (required by Test-Comp); `--witness-output`, which sets the witness output path; `--floatbv`, which enables floating-point SMT encoding; `--unlimited-k-steps`, which removes the upper limit of iteration steps in the *k*-induction algorithm; `--no-align-check`, which deactivates pointer alignment checks; `--falsification`, which enables the falsification mode; and `--force-malloc-success`, which sets that all dynamic allocations must succeed (required by Test-Comp).

The Benchexec tool info module is named `esbmc.py` and the benchmark definition file is `esbmc-falsi.xml`. For Test-Comp 2019, ESBMC uses Boolector v3.0.0 [27] and competes in *Cover-Error* and *Overall* categories.

6 Software Project and Contributors

The ESBMC source code can be downloaded at `https://github.com/esbmc/esbmc`, while self-contained binaries for ESBMC v6.1 64-bit can be downloaded at `https://github.com/esbmc/esbmc/releases`. ESBMC is available under the terms of the Apache License 2.0. Instructions to build ESBMC from source are given in the file `BUILDING` (including the description of all dependencies). ESBMC is a joint project with the Federal U. of Amazonas (Brazil), U. of Southampton (UK), U. of Manchester (UK), and U. of Stellenbosch (South Africa).

References

1. Cordeiro, L.C., Fischer, B., Marques-Silva, J.: SMT-based bounded model checking for embedded ANSI-C software. IEEE TSE **38**(4), pp. 957–974, 2012.
2. Petrov, M., Gagarski, K., Belyaev, M.A., Itsykson, V.M.: Using a bounded model checker for test generation: How to kill two birds with one SMT solver. Automatic Control and Computer Sciences, **49**(7), pp. 466–472, 2015.
3. Anielak, G., Jakacki, G., Lasota, S.: Incremental test case generation using bounded model checking: An application to automatic rating. STTT **17**(3), pp. 339–349, 2015.
4. Rocha, H., Barreto, R.S., Cordeiro, L.C.: Memory management test-case generation of C programs using bounded model checking. In: SEFM, LNCS 9276, pp. 251–267, 2015.
5. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS, LNCS 2988, pp. 168–176, 2004.
6. Lopes, B.C., Auler, R.: Getting Started With LLVM Core Libraries. 1st edn. Packt Publishing (2014)
7. Brummayer, R., Biere, A.: Boolector: An efficient SMT solver for bit-vectors and arrays. In: TACAS, LNCS 5505, pp. 174–177, 2009.
8. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS, LNCS 4963, pp. 337–340, 2008.
9. Dutertre, B.: Yices 2.2. In: CAV, LNCS 8559, pp. 737–744, 2014.
10. Cimatti, A., Griggio, A., Schaafsma, B., Sebastiani, R.: The mathSAT5 SMT solver. In: TACAS, LNCS 7795, pp. 93–107, 2013.
11. Barrett, C., Conway, C., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: CAV, LNCS 6806, pp. 171–177, 2011.
12. Biere, A., Heule, M., van Maaren, H., Walsh, T.: Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications. Volume 185. IOS Press (2009)
13. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: TACAS, LNCS 1633, pp. 193–207, 1999.
14. Morse, J., Cordeiro, L.C., Nicole, D., Fischer, B.: Model checking LTL properties over ANSI-C programs with bounded traces. SoSym **14**(1), pp. 65–81, 2015.
15. Merz, F., Falke, S., Sinz, C.: LLBMC: Bounded model checking of C and C++ programs using a compiler IR. In: VSTTE, LNCS 7152, pp. 146–161, 2012.
16. Ivančić, F., Shlyakhter, I., Gupta, A., Ganai, M.K.: Model checking C programs using F-SOFT. Computer Design, pp. 297–308, 2005.
17. Hooker, J.N.: Solving the incremental satisfiability problem. The Journal of Logic Programming **15**(1), pp. 177–186, 1993.
18. Kroening, D., Ouaknine, J., Strichman, O., Wahl, T., Worrell, J.: Linear completeness thresholds for bounded model checking. In: CAV. LNCS 6806, pp. 557–572, 2011.
19. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Stahlbauer, A.: Witness validation and stepwise testification across software verifiers. In: FSE, pp. 721–733, 2015.
20. Neamtiu, I., Foster, J.S., Hicks, M.: Understanding source code evolution using abstract syntax tree matching. In: Mining Software Repositories. (2005) 1–5
21. Xu, Z., Kremenek, T., Zhang, J.: A memory model for static analysis of C programs. In: Symposium On Leveraging Applications Of Formal Methods, Verification And Validation. LNCS, 6415, pp. 535–548, 2010.
22. Levine, J.: Flex & Bison. 1st edn. O'Reilly Media, Inc., 2009.
23. Pierce, B.C.: Types And Programming Languages. 1st edn. The MIT Press, 2002.
24. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: An efficient method of computing static single assignment form. In: POPL. (1989) 25–35
25. Baldoni, R., Coppa, E., D'elia, D.C., Demetrescu, C., Finocchi, I.: A survey of symbolic execution techniques. ACM Computing Survey **51**(3), 50:1–50:39, 2018.
26. Günther, H., Weissenbacher, G.: Incremental bounded software model checking. In: SPIN. pp. 40–47, 2014.
27. Niemetz, A., Preiner, M., Biere, A.: Boolector 2.0 system description. Journal on Satisfiability, Boolean Modeling and Computation **9** pp. 53–58, 2014.