ESBMC v7.7: Automating Branch Coverage Analysis Using CFG-Based Instrumentation and SMT Solving

(Competition Contribution)

Chenfeng Wei[1]*, Tong Wu[1], Rafael Sa Menezes[1,6], Fedor Shmarov[2], Fatimah Aljaafari[3], Sangharatna Godboley[4], Kaled Alshmrany[1,5], Rosiane de Freitas[6], and Lucas C. Cordeiro[1,6]

[1] The University of Manchester, Manchester, UK
[2] Newcastle University, Newcastle upon Tyne, UK
[3] King Faisal University, Hofuf, Saudi Arabia
[4] National Institute of Technology Warangal, Warangal, India
[5] Institute of Public Administration, Jeddah, Saudi Arabia
[6] Federal University of Amazonas, Manaus, Brazil

Abstract. ESBMC, a bounded model checking (BMC) verifier based on SMT solving, has demonstrated its effectiveness in bug detection in recent software verification competitions. We extend its capabilities to enable branch coverage analysis and test suite generation. Our contributions are twofold: (1) we define a branch coverage property and instrument the control flow graph (CFG) to compute branch coverage using SMT solving, and (2) we propose an incremental multi-property reasoning algorithm for efficient and sound test case generation. ESBMC is ranked 7th in the `Cover-Branches` category of Test-Comp 2025.

1 ESBMC's General Workflow

ESBMC [9] (Efficient SMT-Based Bounded Model Checker) verifies C programs for errors such as arithmetic overflow, array out-of-bounds, and user-defined assertions. Using a Clang-based [11] frontend, it converts the input program into a GOTO program [6], a simplified Control Flow Graph (CFG), to analyze the program's execution flow. The CFG is transformed into a state-space representation, specifically static single-assignment form (SSA) [7], for symbolic execution. This execution is performed symbolically [13] within defined bounds (e.g., through loop unrolling) and is eventually encoded as a verification condition (VC). This VC is an SMT formula incorporating constraints (execution conditions) and properties (expected behaviours). Backend solvers like Boolector [12] or Z3 [8] determine satisfiability, reporting failed properties with counterexamples. Additionally, ESBMC supports incremental and k-induction [4] reasoning for unbounded scenarios.

* Jury member

2 ESBMC's Test-Generation Approach

We extended ESBMC to include a *branch coverage* analysis capability, tracking the program execution across all decision branches and generating corresponding test suites. The process involves two main steps: branch coverage property instrumentation in the GOTO program and multi-property verification in the backend.

2.1 Coverage Property Instrumentation

Instrumentation. In symbolic execution of state systems, entering a branch requires satisfying a precondition, i.e., a branch constraint [1]. For instance, to execute a branch guarded by a non-constant condition like `if(cond)`, there must exist an assignment that satisfies `cond`. This is equivalent to checking if a counterexample satisfies `assert(!cond)`. Based on this insight, we introduce a method that models branch entry by transforming branch constraints into properties, represented as assertions, and instrumented into the program for verification. The instrumentation is applied to the GOTO program rather than the source code, as at this stage, control-flow constructs such as `if-else` statements, `while` loops, `for` loops, and `switch-case` statements are normalized into `if-goto` structures. This normalization simplifies subsequent operations and ensures the instrumentation is applied to all decision points, thereby maintaining soundness. Specifically, constraints from `if` statements (`cond` in statement `if(cond)`) are extracted and converted into two assertions: one representing the false condition (`assert(!cond)`) and the other representing its negation (`assert(!(!cond))`). These two assertions reflect the requirement for executing the `if` and `else` branches, respectively. Ultimately, they are inserted in sequential order before that `if` statement. Performing instrumentation before the `if` statement, rather than within its branches, enhances performance. Since the BMC engine unwinds loops, placing coverage properties inside loop bodies would result in duplicate assertion instances during unwinding, distorting coverage statistics, and hindering test generation. Additionally, this "instrument-before" strategy facilitates the elimination of redundant code, i.e., `if-goto` blocks, during the program slicing stage.

Isolation. Potential interferences are excluded to isolate the analysis of instrumented properties from others. First, all original program properties, e.g. user-defined assertions, are converted into tautologies (`assert(True)`), which are removed during the program slicing [5]. Second, internal safety checks within ESBMC are disabled to prevent unnecessary assertions from being introduced during analysis.

2.2 Multi-Property Verification

Multiple branch coverage properties are inserted during the CFG instrumentation. To reason and generate test suites for each property, we propose a method to verify multi-property in incremental, following these steps:

Symbolic execution & Slicing. The GOTO program is first symbolically executed with loops unwound up to a predefined threshold k, generating verification condition VC. Next, program slicing minimizes constraints by removing irrelevant parts based on the property under verification, reducing the state space.

Property Splitting. Given the sliced VC, the global property is decomposed into atomic properties, and the satisfiability of each assertion is verified within the bounded execution. Eq. (1) demonstrates the key process for this checking procedure, where C and P are used to denote the global constraints and properties of the state system (within bound k), respectively. The global property P is decomposed into a set of unit properties P_i. Thus, the verification condition VC is divided into a set of VC_i.

$$VC \Leftrightarrow C \land \neg P \Leftrightarrow \bigvee_{i}^{n}(C \land \neg P_i) \Leftrightarrow \bigvee_{i}^{n} VC_i \xRightarrow[\text{incr}]{\text{kind}} \bigvee_{i}^{m} VC'(m \leq n) \qquad (1)$$

Base Case. The base case evaluates whether each property P_i holds within k steps. For each VC_i, an SMT solver is set up, checking for satisfiability within a fixed bound k in isolation. Outcomes dictate subsequent actions: (1) Violation: A counterexample is identified and reported. Furthermore, the GOTO program is updated by converting the violated property P_i into a tautology (`assert(True)`). (2) Hold: Forward reasoning is performed to verify behaviour beyond k.

Forward Reasoning. This step evaluates program behavior for $k + 1$ steps and beyond, verifying that no violations occur beyond the k-bound (**Forward Condition**) and proving that if the property holds for k steps, it also holds for $k + 1$ steps (**Inductive Step**) [10]. Outcomes dictate subsequent actions: (1) Violation: A counterexample is reported, and the property P_i is converted into a tautology. (2) Hold: The property P_i is proven correct in an unbounded context and converted into a tautology. This indicates that the branch is unreachable. (3) Unknown: This indicates the bound k is insufficient to evaluate the property (e.g., k is too small to explore deep loop iterations), making forward reasoning undecidable. In such cases, the bound k is incremented for further re-verification.

Termination & Re-verification. The whole verification process terminates under the following conditions: (1) all remaining coverage properties are proven during forward reasoning, or (2) all properties are reduced to tautologies and removed through slicing, leaving no properties for further verification. Conversely, if any property remains unknown, a verification re-run is initiated. In this process, the bound k is incremented, and the updated GOTO program undergoes re-verification from symbolic execution until either a maximum k-threshold or a timeout is reached. Eq. 1 shows the reduction in VCs via slicing ($m \leq n$). The repeated program slicing removes the converted tautologies and their property-relevant code, mitigating state space explosion as the bound k increases. Additionally, to further accelerate k-increment verification, we implement a jump strategy starting k from a larger i and incrementing it by j, where $i, j > 1$.

Test Generation. Test generation occurs whenever a property P_i violation is reported during the **Base Case** or **Forward Reasoning** stage. Assignments with nondeterministic initial values are extracted from the counterexample traces and transformed into corresponding test suites.

3 Strengths and Weaknesses

ESBMC v7.7 uses forward reasoning for instrumentation-based branch coverage analysis, automatically exploring deeper execution paths and terminating when verification goals are met, ensuring full branch coverage. Furthermore, the repeated program slicing during incremental verification mitigates state-space explosion caused by the k-increment, limiting `OUT-OF-MEMORY` issues to 17 cases (`esbmc-incr`) and 12 cases (`esbmc-kind`) in `Cover-Branches`. The jump strategy offsets additional steps from forward reasoning, mitigating timeout issues as shown in `Cover-Error`. With a relatively high starting bound (5) and a jumping step (3), `TIMEOUT` cases decreased from 463 in 2024 Test-Comp [2] to 224 in 2025 [3] (a 48.38% reduction). However, disabling internal safety checks exposed incomplete pointer support within ESBMC. Previously, internal checks detected "null pointer" issues in the program and terminated verification; in their absence, ESBMC attempts to proceed with incorrect pointer encoding, leading to segmentation faults and resulting in 3 `UNKNOWN` cases in the `Cover-Branches`.

4 Tool Setup and Configuration

The tool is run via the wrapper: `esbmc-wrapper.py [options] <PROGRAM>`. Options used in the competition are: `-p <PROPERTY>` for the property path, `-a 32` to set the architecture to 32-bit, `-s kinduction` to enable k-induction in `esbmc-kind` (`incr` for incremental BMC in `esbmc-incr`), and `-o branch` for coverage analysis in `Cover-Branches`. Based on these options, the following ESBMC flags are set during execution: `--base-k-step 2` (3 for `esbmc-incr`) to set the initial bound, `--k-step 3` to set the bound increment, `--unlimited-k-steps` to allow an unlimited upper bound, `--generate-testcase` to output test suites, and `--no-standard-checks` (for `Cover-Branches`) to disable internal safety checks. This configuration is applied globally across all benchmark categories.

5 Software Project and Contributors

ESBMC is open-source under the Apache License 2.0, primarily supported by the University of Manchester, with contributions from other universities and institutions. All people involved are listed as authors of this paper.

6 Data-Availability Statement

The version that participated in Test-Comp 2025 is available in binary form on Zenodo: `https://doi.org/10.5281/zenodo.14340851`. To set up and run ESBMC, follow the instructions in the `README.md` file. ESBMC's official website can be found at `https://ssvlab.github.io/esbmc/`. The ESBMC source code is written in `C++` and publicly available for download on GitHub: `https://github.com/esbmc/esbmc`.

7 Funding Statement

The ESBMC development is funded by ARM, EPSRC EP/T026995/1, EPSRC EP/V000497/1, Ethereum Foundation, EU H2020 ELEGANT 957286, UKRI Soteria, Intel, and Motorola Mobility (through Agreement N° 4/2021).

References

1. Angeletti, D., Giunchiglia, E., Narizzano, M., Puddu, A., Sabina, S.: Automatic test generation for coverage analysis using cbmc. In: Computer Aided Systems Theory-EUROCAST 2009: 12th International Conference, Las Palmas de Gran Canaria, Spain, February 15-20, 2009, Revised Selected Papers 12. pp. 287–294. Springer (2009)
2. Beyer, D.: Automatic testing of C programs: Test-Comp 2024. In: TBA. Springer (2024)
3. Beyer, D.: Advances in automatic software testing: Test-Comp 2025. In: Proc. FASE. Springer (2025)
4. Beyer, D., Dangl, M., Wendler, P.: Boosting k-induction with continuously-refined invariants. In: International Conference on Computer Aided Verification. pp. 622–640. Springer (2015)
5. Cho, C.Y., D'Silva, V., Song, D.: Blitz: Compositional bounded model checking for real-world programs. In: 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 136–146 (2013). `https://doi.org/10.1109/ASE.2013.6693074`
6. Cordeiro, L., Fischer, B., Marques-Silva, J.: Smt-based bounded model checking for embedded ansi-c software. IEEE Transactions on Software Engineering **38** (07 2009). `https://doi.org/10.1109/TSE.2011.59`
7. Cordeiro, L., Fischer, B., Marques-Silva, J.: Smt-based bounded model checking for embedded ansi-c software. In: 2009 IEEE/ACM International Conference on Automated Software Engineering. pp. 137–148 (2009). `https://doi.org/10.1109/ASE.2009.63`
8. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer (2008)
9. Gadelha, M.R., Menezes, R.S., Cordeiro, L.C.: Esbmc 6.1: automated test case generation using bounded model checking. International Journal on Software Tools for Technology Transfer **23**, 857–861 (2021)

10. Gadelha, M.R., Monteiro, F.R., Morse, J., Cordeiro, L.C., Fischer, B., Nicole, D.A.: Esbmc 5.0: an industrial-strength c model checker. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. pp. 888–891 (2018)
11. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis and transformation. In: International symposium on code generation and optimization. pp. 75–88. San Jose, CA, USA (Mar 2004)
12. Niemetz, A., Preiner, M., Biere, A.: Boolector 2.0. Journal on Satisfiability, Boolean Modeling and Computation **9**(1), 53–58 (2014)
13. Ramalho Gadelha, M., et al.: Scalable and precise verification based on k-induction, symbolic execution and floating-point theory. Ph.D. thesis, University of Southampton (2019)