ESBMC v7.4: Harnessing the Power of Intervals
(Competition Contribution)

Rafael Menezes[1], Mohannad Aldughaim[15], Bruno Farias[1], Xianzhiyu Li[1], Edoardo Manino[1], Fedor Shmarov[14], Kunjian Song[1], Franz Brauße[1], Mikhail R. Gadelha[2], Norbert Tihanyi[3], Konstantin Korovin[1], and Lucas C. Cordeiro[1]

[1] University of Manchester, UK
[2] Igalia, A Coruña, Spain
[3] Eötvös Loránd University, Hungary
[4] Newcastle University, UK
[5] King Saud University, Saudi Arabia

Abstract. ESBMC implements many state-of-the-art techniques for model checking. We report on new and improved features that allow us to obtain verification results for previously unsupported programs and properties. ESBMC employs a new static interval analysis of expressions in programs to increase verification performance. This includes interval-based reasoning over booleans and integers, forward and backward contractors, and particular optimizations related to singleton intervals because of their ubiquity. Other relevant improvements concern the verification of concurrent programs, as well as several operational models, internal ones, and also those of libraries such as pthread and the C mathematics library. An extended memory safety analysis now allows tracking of memory leaks that are considered still reachable.

1 Verification Approach

ESBMC [4,6] is a context-bounded model checker for the verification of single- and multi-threaded C programs for various code safety violations (e.g., buffer overflows, dangling pointers, arithmetic overflows) and user-defined assertions. It has been successfully participating in the SV-COMP competitions for many years due to our continuous work towards improving its performance. ESBMC transforms a given C program using a Clang-based [11] front-end into an intermediate representation in the GOTO language [3], which is symbolically executed to produce verification formulae passed to one or more SMT solvers.

2 Software Architecture

Interval analysis In this year, ESBMC interval analysis was improved using Abstract Interpretation techniques [5]. We used the integer domain (with infinities) as the abstract domain for SV-COMP. The domain consists of, for each variable in the program, keeping the box interval (i.e., a *minimum* and *maximum*) for all statements. ESBMC also supports interval arithmetic and widening strategies (through extrapolation and interpolation). Once computed, the intervals are used for optimizations and code instrumentation.

Two optimizations are used: *singleton propagation* and *guard elimination*[6]. The singleton optimization applies when a variable used in an expression is known to have only one possible value; therefore we replace the variable in the expression with the interval value. For guard elimination, we consider a cast of the interval into a three-valued boolean domain (i.e., *true*, *false*, *maybe*). Using this abstract domain, we can optimize always true or false conditions. This is applied recursively in the expression (i.e., try the full expression and then on its operands), being able to reason on both operands and full expression. After the optimization, ESBMC runs a constant propagation pass over the new expression.

Regarding the new code instrumentation, the main use of intervals is to generate invariants which the k-induction strategy benefits from most. This is done by adding assumptions restricting the value of variables, e.g., $x > 3 \wedge x < 10$. In previous editions, ESBMC would instrument an assumption at the start of a loop (before and inside) with all function variables (even if they do not affect the loop variables). For this year, we changed the approach to instrument the assumption only for variables that are part of the statement, e.g., for an *"if"* statement, we only add an assumption with the variables in the condition and the path condition. Lastly, we expanded the types of instrumented statements: assertions, conditionals, and function calls.

Contractors A contractor $\mathcal{C} : \mathbb{R}^n \rightarrow \mathbb{R}^n$ is an interval method that approximates the solutions of a Constraint Satisfaction Problem (CSP) [14] over a set of variables x and variable domains $[\mathrm{x}] \in \mathbb{IR}^n$ with the following properties. For solution set $\mathbb{S}_\mathrm{x}$ and a contractor $\mathcal{C}$, when applied to a box $[\mathrm{x}]$, it ensures that $\mathcal{C}([\mathrm{x}]) \subseteq [\mathrm{x}]$ and $\mathcal{C}([\mathrm{x}]) \cap \mathbb{S}_\mathrm{x} = [\mathrm{x}] \cap \mathbb{S}_\mathrm{x}$, meeting both contraction and correctness conditions respectively [10].

In ESBMC, we apply contractors on conditional statements (*"if"* and loops) by using their conditions to prune the search space for the variables domains. This is done by partitioning the domain $[\mathrm{x}]$ into the three sets $\mathbb{S}_{in}$, $\mathbb{S}_{boundary}$ and $\mathbb{S}_{out}$ such that $\mathbb{S}_{in} \subseteq \mathbb{S}_\mathrm{x}$ is an under-approximation of the solution set and $\mathbb{S}_{out} \subseteq [\mathrm{x}] \backslash \mathbb{S}_\mathrm{x}$ is an under-approximation of the non-solutions. These sets are then used to refine the ranges of each variable occurring in the conditional statement.

The Forward-backward Contractor is utilized by ESBMC for its simplicity and effectiveness. This type of contractor is specifically designed for CSPs with a single constraint [7,8,15]. It operates in two stages: forward evaluation and backward propagation [14,1]. In scenarios with multiple constraints in a CSP, the forward-backward contractor is applied to each constraint independently. ESBMC uses the library Ibex [2] to implement contractor-based reasoning. Ibex is implemented in C++ and designed for accurate interval arithmetic and constraint processing.

Memory leaks This year, ESBMC employs a refined check for the *valid-memtrack* property. This property is loosely described as only allowing those dynamically allocated objects to survive that are still reachable at the end of the program's execution by following a path of pointers stored in objects eventually referenced by global variables. A property violation witness has to contain proof of *unreachability* of a dynamic allocation starting from any global variable.

[6] ESBMC refers to path conditions as guards

The new algorithm leverages the existing one tracking the lifetime of allocations for the *valid-memcleanup* property, but it specifically excludes still-reachable objects from the check. This condition is encoded into an SMT formula using the paths deterministically described by expressions of type struct, union, pointer, or array with constant size. Each possible successor along the path is obtained through the value-set, and the validity is encoded through guards which have to be held at the end of execution.

C mathematical library ESBMC offers extended support for the `math.h` library. Accurate modeling of the semantics of this library is crucial for reasoning on the behavior of complex floating-point software. For example, most neural network code relies on 32-bit floats and may invoke the `math.h` library to compute the result of activation functions, positional encodings, and vector normalisations [12].

In this respect, the IEEE 754 standard [9] mandates bit-precise semantics for a small subset of the `math.h` library only. This subset includes addition, multiplication, division, `sqrt`, `fma`, and other support functions such as `remquo`. In contrast, the behavior of most transcendental functions (e.g., `sin`, `cos`, `exp`, `log`) is platform-specific. Still, the standard recommends implementing the correct rounding whenever possible.

As a tradeoff between precision and verification speed, ESBMC now features a two-pronged design. For the most commonly-used `float` functions, we borrow the MUSL plain-C implementation of numerical algorithms [13]. For the corresponding `double` functions, we employ less complex algorithms with approximate behavior.

Data races Data races occur when multiple threads concurrently access the same memory location, and at least one of these accesses involves a write operation. ESBMC's algorithm for checking data races extends the static code instrumentation CBMC [3] uses. The idea is to add a flag A' to each variable A involved in an assignment. For instance, when A is assigned a value, we create a new variable A' and set it to true. Directly after the assignment to A, we reset A' to false. To identify races, we assert that the value of A' is false when A is accessed. Subsequently, we outline the challenges encountered by ESBMC and the improvements we have implemented.

As this method introduces additional instructions into the program, the potentially larger number of thread interleavings is counteracted by inserting atomic blocks appropriately. Data races are now also checked on access of arrays with non-constant indices. The most challenging aspect of data race detection is the dereference of pointers, as the pointee would have to be instrumented but is not statically known through the value-set analysis. Thus, the new implementation is hybrid, addressing cases unsuitable for static analysis during symbolic execution, thereby enabling ESBMC to detect more types of data races.

3 Strengths and Weaknesses

Interval analysis The interval analysis improved and provided better invariants for ESBMC. The new optimizations help ESBMC to solve new benchmarks in categories with multiple path conditions (i.e., ECA). The main weakness of the method is that our Abstract Interpreter only has partial support for widening, and it is not context-aware

(i.e., function parameters and global variables cannot be tracked globally). This results in a slowdown for categories with loops with thousands of statements (e.g., Hardware). For this reason, we had to go for more imprecise intervals by disabling interval arithmetic (arithmetic operations are extrapolated to infinity).

Contractors While contractors are highly regarded for their ability to provide assured limits on solutions, their cautious approach may lead to overly broad results and less precise conclusions. Therefore, a more rigorous evaluation of contractors is essential to assess their advantages and limitations effectively.

Data races From the results, the data race detection of ESBMC v7.4 is promising. Compared to the previous version, the new algorithm supports more types of expressions and reduces the verification time. The relatively high number of 2.2% incorrect-true verdicts is mostly due to missing support for detecting data races during dereferences of pointers to compound types. This issue will addressed in the future work.

C mathematical library Without operational models of the `math.h` library, ESBMC would assign non-deterministic results, which may cause incorrect counterexamples to be returned. This behavior is especially evident for older versions of ESBMC on neural network code [12], as it usually contains many mathematical operations. ESBMC v7.4 fixes this semantic issue by providing explicit operational models for many common functions in `math.h`, thus yielding no incorrect results on the benchmarks in [12], and achieving second place in the ReachSafety-Floats sub-category.

Memory leaks The new algorithm for the *valid-memtrack* sub-property allowed ESBMC to identify 70 / 153 violations correctly with no incorrect verdicts (previous year: 0 / 134). There is a theoretical weakness in the current implementation concerning dynamic allocations only reachable through pointers stored in arrays of statically unknown size. It could result in spurious incorrect-false verdicts but has not been observed in test cases, yet. We will address this weakness and submit suitable tasks for this property to SV-COMP in the future.

4 Tool Setup and Configuration

ESBMC can be used via the python wrapper `esbmc-wrapper.py` to simplify its usage for the competition. Please refer to its help message (`-h`) for usage instructions. This wrapper runs the ESBMC executable with command line options specific to each supported property.

5 Software Project

ESBMC is a joint project with the Federal University of Amazonas (Brazil), University of Southampton (UK), University of Manchester (UK), and University of Stellenbosch (South Africa). It is publicly available at `http://esbmc.org` under the terms of the Apache License 2.0 and static release builds of ESBMC are provided at `https://github.com/esbmc/esbmc`. The exact version that participated in SV-COMP 2024 is available at `https://doi.org/10.5281/zenodo.10198805`.

References

1. M. Aldughaim, K. M. Alshmrany, M. R. Gadelha, R. de Freitas, and L. C. Cordeiro. Fusebmc_ia: Interval analysis and methods for test case generation. In L. Lambers and S. Uchitel, editors, *Fundamental Approaches to Software Engineering*, pages 324–329, Cham, 2023. Springer Nature Switzerland.
2. G. Chabert and ibex team. ibex-lib, 2023. `https://github.com/ibex-team/ibex-lib` [Accessed: 19 December 2023].
3. E. Clarke et al. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.
4. L. C. Cordeiro et al. SMT-based bounded model checking for embedded ANSI-C software. *IEEE Transactions on Software Engineering*, 38(4):957–974, 2012.
5. P. Cousot. *Principles of Abstract Interpretation*. MIT Press, 2021.
6. M. Y. R. Gadelha et al. ESBMC 5.0: an industrial-strength C model checker. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering ASE*, pages 888–891. ACM, 2018.
7. L. Granvilliers. Revising hull and box consistency. *Logic Programming*, pages 230–244, 1999.
8. E. Hansen and G. W. Walster. *Global optimization using interval analysis: revised and expanded*, volume 264. CRC Press, 2003.
9. IEEE. IEEE standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019.
10. L. Jaulin, M. Kieffer, O. Didrit, and E. Walter. Applied Interval Analysis. In *Springer London*, 2001.
11. C. Lattner et al. LLVM: A compilation framework for lifelong program analysis and transformation. pages 75–88, San Jose, CA, USA, Mar 2004.
12. E. Manino, R. S. Menezes, F. Shmarov, and L. C. Cordeiro. NeuroCodeBench: a plain C neural network benchmark for software verification, 2023.
13. musl community. musl libc, 2023. `https://musl.libc.org/` [Accessed: 15 December 2023].
14. M. Mustafa, A. Stancu, N. Delanoue, and E. Codres. Guaranteed SLAM—An interval approach. *Robotics and Autonomous Systems*, 100:160–170, 2018.
15. A. Neumaier and A. Neumaier. *Interval methods for systems of equations*, volume 37. Cambridge university press, 1990.