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

Rafael Sá Menezes[1,2], Mohannad Aldughaim[1,6], Bruno Farias[1], Xianzhiyu Li[1], Edoardo Manino[1], Fedor Shmarov[1,5], Kunjian Song[1], Franz Brauße[1⋆], Mikhail R. Gadelha[3], Norbert Tihanyi[4], Konstantin Korovin[1], and Lucas C. Cordeiro[1,2]

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

Abstract. ESBMC implements many state-of-the-art techniques that combine abstract interpretation and model checking. Here, we report on new and improved features that allow us to obtain verification results for previously unsupported programs and properties. ESBMC now employs a new static interval analysis of expressions in programs to increase verification performance. This includes interval-based reasoning over booleans and integers, and forward-backward contractors. 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 Software Architecture

ESBMC [4,6] is a mature, permissively licensed open-source 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. In addition, ESBMC implements state-of-the-art incremental BMC and k-induction proof-rule algorithms based on SMT and Constraint Programming (CP) solvers.

⋆ Jury member

2 Verification Approach

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 statement in the program, keeping the box interval (i.e., a *minimum* and *maximum*) for all variables. ESBMC also supports interval arithmetic and widening strategies (through extra- and interpolation). Once computed, the intervals are used for optimizations (i.e., dead code elimination and constant folding) and invariant instrumentation.

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. In addition, the set of variables used for these assumptions has been reduced to those occurring in conditional statements and guards only. Lastly, we expanded the types of instrumented statements: assertions, conditionals, and function calls.

Contractors ESBMC v7.4 employs another method to refine intervals based on contractors. Contractors [10,14] are commonly used in the context of Constraint Satisfaction Problems (CSPs), that is, when variables, their (real-valued) domains, and constraints over those variables are fixed. A contractor is an operation on n-dimensional boxes (product of intervals) respecting the given constraints, i.e., it refines the domains such that no solutions to the CSP are lost. A particularly efficient one for CSPs containing a single constraint is the Forward-backward contractor [7,8,15]. It operates in two stages: forward evaluation and backward propagation [14,1]. In scenarios with multiple constraints, the forward-backward contractor is applied to each constraint independently.

ESBMC utilizes the forward-backward contractor implemented in the Ibex library [2] to refine the results of the interval analysis mentioned above. That is, conditions of statements such as "if" and loops in the program are relaxed to conditions over reals, where possible, and then the contractor is applied to this relaxed condition. The result is a refined set of intervals for the variables involved. These refined intervals are then restricted to the original variable domains, which – in case of, e.g., integers – results in a further reduction of the size of intervals. The intervals contracted in this way generally enhance the results of the interval analysis employed by ESBMC and benefit its k-induction strategy.

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.

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 hold at the end of execution.

C mathematical library ESBMC v7.4 offers extended support for the `math.h` library. Accurate modeling of its semantics 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].

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`, `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', initially true, to each variable A involved in an assignment. Directly after the assignment to A, A' is reset 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 – subject to ensuring accuracy, the atomic block encompasses the assertion on A', original assignment to A, and setting A', in sequence. 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 pointer 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

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).

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.

The new algorithm for the *valid-memtrack* sub-property allowed ESBMC to identify 70/153 violations correctly with no incorrect verdicts (last 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 incorrect-false verdicts, but it has not been observed in test cases, yet.

Without operational models of the `math.h` library, ESBMC would assign nondeterministic 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.

From the competition 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 still missing support for detecting data races during dereferences of pointers to compound types.

We will address the weaknesses identified in this competition in the future.

4 Tool Setup and Configuration

To setup and run ESBMC, follow the instructions in the `README.md` file. ESBMC can also be run via the Python wrapper `esbmc-wrapper.py` for simplified usage in the competition. An example command line is:

```
esbmc-wrapper.py -s kinduction -a 64 -p unreach-call.prp example.c
```

5 Software Project

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). 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 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, D. Kroening, and F. Lerda. 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, B. Fischer, and J. Marques-Silva. 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, F. R. Monteiro, J. Morse, L. C. Cordeiro, B. Fischer, and D. A. Nicole. 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 and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *International symposium on code generation and optimization*, 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. *Interval methods for systems of equations*, volume 37. Cambridge University Press, 1990.