Context-Bounded Model Checking with ESBMC 1.17
(Competition Contribution)

Lucas Cordeiro[1], Jeremy Morse[2], Denis Nicole[2], and Bernd Fischer[2]

[1] Electronic and Information Research Center, Federal University of Amazonas, Brazil
[2] Electronics and Computer Science, University of Southampton, UK
esbmc@ecs.soton.ac.uk

Abstract. ESBMC is a context-bounded symbolic model checker for single- and multi-threaded ANSI-C code. It converts the verification conditions using different background theories and passes them directly to an SMT solver.

1 Overview

ESBMC is a context-bounded symbolic model checker that allows the verification of single- and multi-threaded C code with shared variables and locks. ESBMC supports full ANSI-C (as defined in ISO/IEC 9899:1990), and can verify programs that make use of bit-level operations, arrays, pointers, structs, unions, memory allocation and floating-point arithmetic. It can reason about arithmetic under- and overflows, pointer safety, memory leaks, array bounds violations, atomicity and order violations, local and global deadlocks, data races, and user-specified assertions. However, as with other bounded model checkers, ESBMC is in general incomplete.

ESBMC uses the CBMC [2] frontend to generate the verification conditions (VCs) for a given program, but converts the VCs using different background theories and passes them to a Satisfiability Modulo Theories (SMT) solver. ESBMC natively supports Z3 and Boolector but can also output the VCs using the SMTLib format.

ESBMC supports the analysis of multi-threaded ANSI-C code that uses the synchronization primitives of the POSIX Pthread library. It traverses a reachability tree (RT) derived from the system in depth-first fashion, and calls the SMT solver whenever it reaches a leaf node. It stops when it finds a bug, or has explored all possible interleavings (i.e., the full RT). This combination of symbolic model checking with explicit state space exploration is similar to the ESST approach [1] for SystemC.

2 Verification Approach

We model a program as a transition system $M = (S, R, S_0)$, where S is the set of states, $S_0 \subseteq S$ the initial states, and $R \subseteq S \times S$ the transition relation. A state $s \in S$ consists of the values of all program variables, including the program counter pc. We use $I(s_0)$ to denote that $s_0 \in S_0$ and $\gamma(s_i, s_{i+1})$ to denote the constraints that correspond to a transition between two states s_i and s_{i+1}.

Given a transition system M, a safety property ϕ, a context switch bound C, and a bound k, ESBMC builds an RT that represents the program unfolding for C, k, and ϕ. For each interleaving π that passes through the RT nodes ν_1 to ν_k, it derives a formula $\psi_k^\pi = I(s_0) \wedge \bigvee_{i=0}^{k} \bigwedge_{j=0}^{i-1} \gamma(s_j, s_{j+1}) \wedge \neg\phi(s_i)$ which is satisfiable iff ϕ has a counterexample of depth k or less that is exhibited by π. Since $I(s_0) \wedge \bigwedge_{j=0}^{i-1} \gamma(s_j, s_{j+1})$ represents an execution of M of length i, ψ_k^π is satisfied iff for some time step $i \leq k$ there exists a reachable state along π at which ϕ is violated. The SMT solver then provides a satisfying assignment, from which we can extract a counterexample trace to the property violation. If ψ_k^π is unsatisfiable, we can conclude that no error state is reachable in k steps or less along π. Finally, we use $\psi_k = \bigvee_\pi \psi_k^\pi$ to check all interleavings.

ESBMC uses a quantifier-free fragment of a logic of linear integer arithmetics with arrays and uninterpreted functions to represent the VCs derived for a given ANSI-C program. Scalar datatypes can be represented either as bitvectors (i.e., using the SMTLib logic QF_AUFBV), or as abstract integers (i.e., QF_AUFLIA). Floating point arithmetic is approximated either by abstract reals (i.e., using the SMTLib logic AUFLIRA), by fixed-point arithmetic using bit vectors, or by rational arithmetic over abstract integers. Structures, unions, and pointer types are encoded using tuples [3,4]. ESBMC uses a simple but effective heuristic to select the best data representation and SMT solver.

ESBMC uses an instrumented model of the Pthread synchronization primitives to handle multi-threaded code [3]. The model allows an unbounded set of threads T, but ESBMC will only explore a finite number of context switches. It assumes that an enabled thread $t_j \in T$ can transition between statements to any enabled thread, and computes all states for which a transition exists to (implicitly) build the RT. ESBMC assumes sequential consistency, and by default assumes that variable accesses in individual statements are atomic.

3 Architecture, Implementation, and Availability

ESBMC is implemented in C++. It has been branched off CBMC v2.9, and still uses its parser, goto conversion, and core of the symbolic execution. The goto conversion replaces all control structures by (conditional) jumps, which simplifies the program representation. The symbolic execution of this goto representation converts the program into SSA form and unrolls loops and recursive functions on-the-fly, generating unwinding assertions that fail if the given bound is too small. It also generates the VCs for the safety properties and user-specified assertions.

Changes to the CBMC components include the addition of the new safety properties and more optimizations (e.g., better constant propagation), and the integration of native SMT backends. In order to support the analysis of multi-threaded code ESBMC implements a partial-order reduction (POR) [3,5] scheme to reduce the number of states that have to be explored. It first applies the visible instruction analysis POR, which removes the interleavings of instructions that do not affect the global variables, followed by the read-write analysis POR in which two (or more) independent interleavings can be safely merged into one. Additionally, it implements a two-level symbolic state hashing scheme [6] that represents a particular RT node and all constraints affecting a particular assignment to a variable separately. Since each new RT node can only change the (symbolic) value of at most one variable, this scheme reduces the computational effort, as it allows us to retain the hash values of the unchanged variables.

User Interface. ESBMC can be invoked through a standard command-line interface or configured through an Eclipse plug-in. When a property violation is detected, it produces a counterexample trace in the CBMC format. The plugin visualizes such traces and provides direct access to the corresponding code.

Availability and Installation. Self-contained binaries for 32-bit and 64-bit Linux environments are available at `www.esbmc.org`; versions for other operating systems are available on request. The competition version only uses the Z3 solver (V3.1). It assumes a 64-bit architecture and uses experimentally determined unwinding bounds; setting explicit context switch bounds is not required for the given concurrency benchmarks. It only checks for the reachability of the error label and ignores all other properties, including unwinding assertions. It is called as follows:

```
esbmc --timeout 15m --memlimit 15g --64 --unwind <n>
 --no-unwinding-assertions --no-assertions --error-label ERROR
 --no-bounds-check --no-div-by-zero-check --no-pointer-check <f>
```

4 Results

With unwinding assertions enabled, ESBMC proves 30 programs correct and finds errors in 27. However, it also claims errors in two correct programs and fails to find existing errors in another nine. ESBMC's performance is largely similar across all categories, although unwinding assertions and timeouts are, as expected, concentrated on the larger benchmarks.

With unwinding assertions *disabled*, a correctness *claim* is not a full correctness *proof*, because errors could occur for larger unwinding bounds. In fact, the number of false negatives/positives increases to ten and nine, resp., but their rate remains roughly the same, so that ESBMC's overall performance improves markedly: with 121 programs rightly claimed correct and 71 errors identified, it achieves a total score of 249. Four programs do not conform to the supported ANSI-C standard and cause parsing errors. The remaining programs time out during the symbolic execution (21) or fail with an internal error (41). These errors results are mostly caused by problems in ESBMC's pointer handling that are exposed by the excessive typecasts in the CIL-converted code.

Acknowledgments. The development of ESBMC is funded by EPSRC, EC FP7, the Royal Society, and INdT. Q. Li implemented the Eclipse plugin.

References

1. Cimatti, A., Micheli, A., Narasamdya, I., Roveri, M.: Verifying SystemC: a software model checking approach. In: FMCAD, pp. 121–128 (2010)
2. Clarke, E., Kroning, D., Lerda, F.: A Tool for Checking ANSI-C Programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)

3. Cordeiro, L.: SMT-Based Bounded Model Checking of Multi-Threaded Software in Embedded Systems. PhD Thesis, U Southampton (2011)
4. Cordeiro, L., Fischer, B., Marques-Silva, J.: SMT-based bounded model checking for embedded ANSI-C software. In: ASE, pp. 137–148 (2009)
5. Cordeiro, L., Fischer, B.: Verifying Multi-Threaded Software using SMT-based Context-Bounded Model Checking. In: ICSE, pp. 331–340 (2011)
6. Morse, J., Cordeiro, L., Nicole, D., Fischer, B.: Context-Bounded Model Checking of LTL Properties for ANSI-C Software. In: Barthe, G., Pardo, A., Schneider, G. (eds.) SEFM 2011. LNCS, vol. 7041, pp. 302–317. Springer, Heidelberg (2011)