Improved loop unwinding in ESBMC 2.1
(Competition Contribution)

Mikhail Ramalho¹, Jeremy Morse², Lucas Cordeiro³, and Denis Nicole¹

¹ Electronics and Computer Science, University of Southampton, UK
² Department of Computer Science, University of Bristol, UK
³ Electronic and Information Research Center, Federal University of Amazonas, Brazil
esbmc@ecs.soton.ac.uk

Abstract. We implement an alternative loop unwinding strategy for ESBMC at the GOTO level. This substantially improves the reliability of unwinding nested loops.

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 was originally based on CBMC (v2.9) [1] and has inherited its object-based memory model. We then implemented an improved memory model for ESBMC, adapting a fully byte-precise memory model as for example used by LLBMC [2].

In this paper we focus on a new approach used by ESBMC to verify programs, the unwinding of loops at goto level. An overview of ESBMC’s architecture and more details are given in our previous work [3–6].

2 Differences to ESBMC 1.24

In the last year we have mostly made changes to improve ESBMC’s stability, precision and performance, as well as adding new options for verification. We adjusted our union implementation as because SMT does not have a good way of representing unions; we now allocate a byte array as storage for unions, and force all union accesses to be performed through pointers. The dereference layer handles the decomposition of these accesses to byte array accesses. This seems to work well; the only limitation is that assignments of type union become assignments of type array, which the dereference layer cannot handle. The ϵ-induction algorithm was also completely rewritten and is again available on ESBMC.

ESBMC now also offers the option to fully inline a program before verification (this feature is not, however, supported when verifying concurrent programs), and the option to unroll loops at the GOTO level, transforming any program into a loop free program. This paper focuses on the latter added feature.
3 Loop Unrolling

We recap notions about the control-flow graph (CFG) and some of its properties, which are standard in the compiler literature [7]. These concepts are needed because the loop unrolling algorithm transforms a reducible CFG into an acyclic CFG.

We follow Alastair et al. [8] and define CFGs as a tuple $\langle V, in, E, code \rangle$ where $V$ is a finite set of nodes, $in \in V$ is an initial node, $E \subseteq V \times V$ and $code : V \rightarrow \text{stmt}$ is a mapping from nodes to statements on the program. The CFG is then a direct translation of a program. We say that $d$ dominates $n$, for $d, n \in V$, if every path from $in$ to $n$ goes through $d$. Under this definition, every node dominates itself. A back edge is an $\text{Edge}(a, b) \in E$ whose head $b$ dominates $a$. A CFG is said reducible if all its edges that induce cycles are back edges and is said to be acyclic if the CFG contains no back edges.

The loop unrolling algorithm implemented in ESBMC is similar to the unrolling algorithm described by Alastair et al. [8] to create loop-free programs for their $k$-algorithm. The algorithm removes all back edges and instead appends copies of the loop, replacing the loop statement containing the condition by an IF statement. Earlier exits (e.g., break statements on the original program) and iterations skips (e.g., continue statements on the original program) are treated accordingly by creating edges to the last node of the last copy and creating edges to the first node of the next copy, respectively.

Our difference from their algorithm is in how nested loops are handled. Our algorithm works from the inner nested loop outwards before creating the copies, while the one implemented by Alastair works from the outermost loop downwards. The problem with the latter is that the creation of copies of the loop body might replicate nested loops that are not unrolled by the algorithm. By working on the opposite direction, we avoid this problem.

4 Competition Approach

In bounded model checking, the choice of a unwinding bound that completely unrolls all the loops makes a huge difference. Differently from previous years, we set a global value of 128 for the number of unwindings and call CPAChecker to validate the witness before we present the result of the verification. The call to ESBMC has a global set of parameters that run for all properties:

```
esbmc --timeout 895s --memlimit 15g -DLLV_ERROR=ERROR
-D_Bool=int --boollector --unroll-loops --unwind 128
--no-unwinding-assertions --no-div-by-zero-check
--error-label ERROR --force-malloc-success
```

Here, --unroll-loops tells ESBMC to unroll the loops at goto level and the option --unwind 128 set the number of unwindings to 128. We also add the option --no-unwinding-assertions to replaces the unwinding assertions with a loop assumption and thus a correctness claim is not a full correctness proof; however, this to leads to correct results on most of the benchmarks on SV-COMP. --boollector
force the use of boolector as the solver. --force-malloc-success tells that all dynamic allocations succeed (a requirement from SV-COMP).

The memory model is the same for all categories. If the property to be checked is memory safety, we append --memory-leak-check to the parameters, otherwise we append --no-pointer-check --no-bounds-check. Finally, in order to check overflow properties, we append the parameter --overflow-check.

5 Results

With the approach described, ESBMC correctly claims 1803 benchmarks correct and correctly find errors in 786 benchmarks, in a total of 2589 correct results. However, it also gives 26 incorrect results (5 false successes and 21 false failures). The 5 false successes are spread between the Concurrency (3) and Loops (2) categories. These arise in situations where we replace the loop unwinding assertion by unwinding assumptions. The false failures are found across almost all the categories, but in particular the Recursive (4) category, presented the higher number of failures. The results are expected since the number of iterations is bounded to 128 and is not deep enough to claim correctness on these programs. The Float category also presented 5 false failures; they are not related to this technique but to our fixed-pointing float representation which does not handle NaNs. ESBMC won two medals on SVCOMP16, a gold medal in Arrays category (190 scores) and a bronze medal in BitVectors category (84 scores). On the overall category, ESBMC achieves 4145 scores, which places it on the fourth position among all competitors.

Availability. The script and self-contained binaries for 32-bit and 64-bit Linux environments are available at www.esbmc.org; the source code can also be found at https://github.com/esbmc/esbmc; versions for other operating systems are available on request. The competition version only uses the boolector solver (V2.0.1).

References