CBMC Path: A Symbolic Execution Retrofit of the C Bounded Model Checker
(Competition Contribution)

Kareem Khazem[1]* and Michael Tautschnig[2]

[1] University College London
[2] Queen Mary University of London

Abstract We gave CBMC the ability to explore and model check single program paths, as opposed to its default whole-program model-checking behaviour. This means that CBMC, when invoked with the `--paths` flag, symbolically executes one program path at a time—saving unexplored paths for later—and attempts to prove properties for only that path. By doing this repeatedly for each path that CBMC encounters, CBMC can detect property violations in a scalable and incremental way.
Implementing single-path exploration raises the question of which order the paths should be explored in. Our implementation makes it easy for researchers to implement and investigate alternative path exploration strategies. Our competition contribution uses a breadth-first strategy, where diverging paths are each pushed onto a queue at program decision points, and the path to explore next is gotten by dequeueing the oldest path to have been added.

1 Overview

CBMC Path is an extension to the C Bounded Model Checker. The original CBMC tool was first described in [1] and is also competing in this year's SV-COMP (as previously described in [3]). CBMC symbolically executes C programs up to a user-defined loop unrolling bound. It generates a bit-precise encoding of the unrolled program, annotated with assertions that are the negation of properties the user wishes to verify. Such an encoding is satisfiable if the original program violates the properties. Thus, CBMC can be used to demonstrate that bugs occur when the program is run up to the unwinding bound for some input.

The original CBMC tool generates a single formula describing the disjunction of all program paths, dispatching this entire formula off to a SAT solver. The solver thus decides whether a bug exists at any point in the entire program. In contrast, CBMC Path dispatches the formula for a *single program path* to the SAT solver, checking whether that path violates any properties before continuing to execute and dispatch subsequent program paths. As with CBMC, the SAT solver used in CBMC Path for SV-COMP is MiniSat 2.2.0 [2].

*Juror

The intermediate representation that CBMC uses has no control-flow structures. Instead, GOTO statements explicitly define control flow. Furthermore, all functions in the intermediate representation have a single return point, at the end of the function. Thus, program paths that diverge—either due to an explicit conditional in the original code or an unrolled loop—eventually reunite at a later point in code. The treatment of these join points is an important difference between CBMC and CBMC Path.

CBMC Path is merged into the main CBMC codebase; users activate it by passing the `--paths` flag to CBMC in addition to the other options that they wish to use. Since it is a fairly focussed change to the codebase, we describe only the aspects that differ from CBMC's default mode in this report.

2 Architecture

The changes that CBMC Path makes to the original codebase are confined to the symbolic execution phase. There are three main changes, illustrated in fig. 1:

– the symbolic execution (symex) state, including the path taken so far, can now be saved. This means that the symex process as a whole can be paused and subsequently resumed from a saved state.
– The symex code now has the option to avoid merging two divergent program paths at their join point. When combined with the above point, this means that CBMC Path can save both divergent paths at their branch point, execute one of the paths, and then continue executing past the join point without considering the other branch at all.
– The top-level symex code now maintains a worklist of symex states (partially-executed paths), rather than a single state that lives through execution of the entire target program. The top-level symex code includes a loop that repeatedly pops the worklist and executes the popped state until it reaches a branch point (conditional goto), at which point the states corresponding to each branch are pushed onto the worklist. When combined with the above two points, this means that control returns to the top-level each time a pair of divergent paths in the target program are saved onto the worklist. The top-level code then decides which path to continue executing, and pops that path from the worklist.

Program paths are thus pushed to and popped from the worklist until one of the paths has been executed to the end of the program; at this point, the top-level code dispatches its formula to the SAT solver, before continuing to execute the remaining paths.

The decision of which path should be resumed can be expressed as a list-popping strategy. We have so far implemented the last-in, first-out (LIFO) and first-in, first-out (FIFO) disciplines. LIFO explores the program depth-first, completing a single path before starting to explore any others; this strategy uses memory efficiently (since only a single full path is maintained in memory), but takes far longer to cover a variety of program paths. FIFO expands the explored-instruction frontier of all program paths in a round-robin manner; this achieves excellent coverage at the (considerable) expense of keeping all paths in memory until each of them reaches the end of the program. The user can choose the strategy as an argument to the `--paths` option. For SV-COMP, we chose to use FIFO for all benchmarks, since the modest size of the benchmarks means that memory use was rarely a problem. The strategy system is designed to be easily extensible to encourage researchers to experiment with the different trade-offs of more sophisticated path-popping strategies.

3 Motivation, Strengths, and Weaknesses

CBMC Path shares many of the strengths and weaknesses that CBMC has compared to other tools, which are discussed in the CBMC system report. In this section, we focus on a comparison with CBMC.

CBMC Path is aimed at users wishing to discover bugs quickly and efficiently, while being not so suited to proving program correctness. By only model-checking individual program paths, each SAT solver call returns much more quickly, ensuring that bugs along those paths are discovered without having to wait for the rest of the program to be encoded and checked. In addition, CBMC Path can be parsimonious in its memory usage, since the formulas dispatched to the solver are much smaller. These qualities make CBMC Path complementary to whole-program bug-finding or correctness-proving tools. The intended use-case is that CBMC Path is used to quickly find bugs of the low-hanging fruit variety, so that users can avoid model-checking the entire program only to discover an error residing a few lines into the program entry-point.

The overhead of saving and resuming paths, and of multiple calls to the SAT solver, means that CBMC Path will always take significantly longer to check an entire program than CBMC does. Proving absence of property violations always requires checking the entire program, so CBMC is expected to outperform CBMC Path on most SV-COMP benchmarks. On the other hand, CBMC Path usually has a lower maximum memory use than CBMC when using the LIFO strategy, since both the in-memory symbolic state and the formulas dispatched to the solver are smaller at all times. This means that CBMC Path can still

```
1        if(x > y)
2            GOTO 3;
3        GOTO 2;
4   3:   ret = 1;
5        GOTO 1;
6   2:   ret = 0;
7   1:   return;
```

Figure 1. The program on the left (in CBMC's intermediate representation) illustrates the difference between CBMC's and CBMC Path's exploration strategies. CBMC executes all seven lines. At the join point (line 7), CBMC creates a disjunction representing both paths. In contrast, at line 1, CBMC Path pauses symbolic execution and saves two paths onto the top-level worklist: one path whose program counter is line 3, and another whose program counter is line 4. The top-level workloop then chooses a path from the worklist to resume executing.

be useful for proving correctness, specifically when used on large programs that require more-than-available system memory under CBMC. CBMC Path is most useful when used for finding bugs, since it will often find a property violation before CBMC has had a chance to fully explore the program. In addition, the user experience is often more encouraging using CBMC Path, since results are displayed incrementally. This is an important contrast to tools which terminate due to memory exhaustion before displaying any results at all. We view CBMC Path as complimentary to CBMC, and hope that developers use them both to their respective strengths.

This year (CBMC Path's SV-COMP debut), CBMC outperformed CBMC Path on most benchmarks. For small SV-COMP benchmarks, the time that CBMC spends in the solver for the entire program is less than the path pushing and popping overhead. We hope to introduce more sophisticated path strategies that will mitigate this overhead for future competitions.

4 Tool Setup

The competition submission is based on CBMC version 5.10, with additional patches. The archive of the competition binary is available at https://gitlab.com/sosy-lab/sv-comp/archives-2019/raw/svcomp19/2019/cbmc-path.zip.

To process a benchmark `FOO.c` (with properties in `FOO.prp`), the wrapper `cbmc-path.py` should be invoked as follows:

```
cbmc-path.py --graphml-cex witness.cex \
                --propertyfile FOO.prp --32 FOO.c
```

for all categories with a 32-bit memory model; for those with a 64-bit memory model, `--32` should be replaced by `--64`.

Participation. CBMC Path competes in all categories.

Output. The last line of output produced by `cbmc` is one of TRUE, FALSE, FALSE(no-overflow), FALSE(valid-free), FALSE(valid-deref), or FALSE(valid-memtrack). Absence of such a final line is treated as UNKNOWN by the wrapper script.

5 Software Project

CBMC Path is fully merged into the original CBMC codebase. It is maintained by Daniel Kroening with patches supplied by the community. It is made publicly available under a BSD-style license. The source code and binaries for popular platforms are available at http://www.cprover.org/cbmc.

Acknowledgements

We would like to thank the reviewers for their feedback and Mark R. Tuttle for his assistance with this work.

References

1. Clarke, E.M., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS. pp. 168–176. Springer (2004)
2. Eén, N., Sörensson, N.: An extensible SAT-solver. In: SAT. pp. 502–518. Springer (2003)
3. Kroening, D., Tautschnig, M.: CBMC - C bounded model checker - (competition contribution). In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2014). Lecture Notes in Computer Science, vol. 8413, pp. 389–391. Springer (Apr 2014), http://dx.doi.org/10.1007/978-3-642-54862-8_26, *CBMC won the overall Gold medal.*