CBMC – C Bounded Model Checker
(Competition Contribution)

Daniel Kroening[1] and Michael Tautschnig[2]

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

Abstract CBMC implements bit-precise bounded model checking for C programs and has been developed and maintained for more than ten years. CBMC verifies the absence of violated assertions under a given loop unwinding bound. Other properties, such as SV-COMP's ERROR labels or memory safety properties are reduced to assertions via automated instrumentation. Only recently support for efficiently checking concurrent programs, including support for weak memory models, has been added. Thus, CBMC is now capable of finding counterexamples in all of SV-COMP's categories. As back end, the competition submission of CBMC uses MiniSat 2.2.0.

1 Overview

The C Bounded Model Checker (CBMC) [2] demonstrates the violation of assertions in C programs, or proves safety of the assertions under a given bound. CBMC implements a bit-precise translation of an input C program, annotated with assertions and with loops unrolled to a given depth, into a formula. If the formula is satisfiable, then an execution leading to a violated assertion exists. For SV-COMP, satisfiability of the formula is decided using MiniSat 2.2.0 [4].

2 Architecture

Bounded model checkers such as CBMC reduce questions about program paths to constraints that can be solved by off-the-shelf SAT or SMT solvers. With the SAT back end, and given a program annotated with assertions, CBMC outputs a CNF formula the solutions of which describe program paths leading to assertion violations. In order to do so, CBMC performs the following main steps, which are outlined in Figure 1, and are explained below.

Front end. The *command-line front end* first configures CBMC according to user-supplied parameters, such as the bit-width. The *C parser* utilises an off-the-shelf C preprocessor (such as `gcc -E`) and builds a parse tree from the preprocessed source. Source file- and line information is maintained in annotations. *Type checking* populates a symbol table with type names and symbol identifiers by traversing the parse tree. Each symbol is assigned bit-level type information. CBMC aborts if any inconsistencies are detected at this stage.

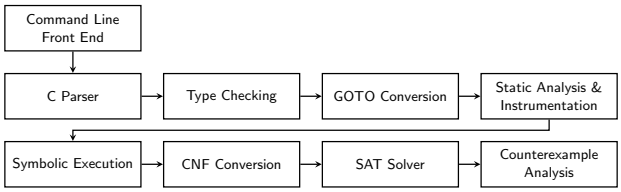

Figure 1. CBMC architecture

Intermediate Representation. CBMC uses *GOTO programs* as intermediate representation. In this language, all non-linear control flow, such as if or switch-statements, loops and jumps, is translated to equivalent *guarded goto* statements. These statements are branch instructions that include (optional) conditions. CBMC generates one GOTO program per C function found in the parse tree. Furthermore, it adds a new main function that first calls an initialisation function for global variables and then calls the original program entry function.

At this stage, CBMC performs a light-weight static analysis to resolve function pointers to a case split over all candidate functions, resulting in a static call graph. Furthermore, assertions to guard against invalid pointer operations or memory leaks are inserted.

Middle end. CBMC performs symbolic execution by eagerly unwinding loops up to a fixed bound, which can be specified by the user on a per-loop basis or globally, for all loops. In the course of this unwinding step, CBMC also translates GOTO statements to static single assignment (SSA) form. Constant propagation and expression simplification are key to efficiency, and prevent exploration of certain infeasible branches. At the end of this process the program is represented as a system of equations over renamed program variables in guarded statements. The guards determine whether an assignment is actually performed in a given concrete program execution. In [1] we presented an extension to perform efficient bounded model checking of concurrent programs, which symbolically encodes partial orders over read and write accesses to shared variables.

Back end. While CBMC also supports SMT solvers as back ends, we use MiniSat 2.2.0 in this competition. Consequently, the resulting equation is translated into a CNF formula by bit-precise modelling of all expressions plus the Boolean guards [3]. A model computed by the SAT solver corresponds to a path violating at least one of the assertions in the program under scrutiny, and the model is translated back to a sequence of assignments to provide a human-readable counterexample. Conversely, if the formula is unsatisfiable, no assertion can be violated *within the given unwinding bounds*.

3 Strengths and Weaknesses

As a bounded model checker, and in absence of additional loop transformations or k-induction, CBMC cannot provide proofs of correctness for programs

with unbounded loops in general. Yet we decided to enforce termination with a TRUE/FALSE answer within the time bounds specified in SV-COMP to provide best-effort answers. Consequently there may be unsound results on certain benchmarks. To reduce the number of such results, the wrapper script (see below) runs CBMC with increasing loop bounds of 2, 6, 12, 17, 21, and 40 until the timeout is reached. These values were obtained as educated guesses informed by the training phase.

Apart from this fundamental limitation, we observed several errors (both false positives and false negatives) caused by current limitations in treatment of pointers. This affects at least one benchmark in the Concurrency category and possibly several in MemorySafety.

The strengths of bounded model checking, on the other hand, are its predictable performance and amenability to the full spectrum of categories.

4 Tool Setup

The competition submission is based on CBMC version 4.5. The full source code of the competing version is available at

http://svn.cprover.org/svn/cbmc/releases/cbmc-4.5-sv-comp-2014/.

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

```
cbmc-wrapper.sh --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`.

5 Software Project

CBMC 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.

References

1. Alglave, J., Kroening, D., Tautschnig, M.: Partial orders for efficient bounded model checking of concurrent software. In: CAV. pp. 141–157 (2013)
2. Clarke, E.M., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS. pp. 168–176. Springer (2004)
3. Clarke, E.M., Kroening, D., Yorav, K.: Behavioral consistency of C and Verilog programs using Bounded Model Checking. In: DAC. pp. 368–371 (2003)
4. Eén, N., Sörensson, N.: An extensible SAT-solver. In: SAT. pp. 502–518. Springer (2003)