2LS for Program Analysis
(Competition Contribution)

Peter Schrammel(✉) and Daniel Kroening

University of Oxford, Oxford, UK
peter.schrammel@cs.ox.ac.uk

Abstract. 2LS is a program analysis tool for C programs built upon the CPROVER infrastructure. 2LS is bit-precise and it can verify and refute program assertions. 2LS implements invariant generation techniques, incremental bounded model checking and incremental k-induction. The competition submission uses an algorithm combining all three techniques, called kIkI (k-invariants and k-induction). As a back end, the competition submission of 2LS uses Glucose 4.0.

1 Overview

2LS is a static analysis and verification tool for C programs that can perform interprocedural abstract interpretation, verification and refutation of assertions and termination analysis [3]. The competition version is configured for monolithic verification and refutation of assertions using an algorithm called kIkI (k-invariants and k-induction) [2], which elegantly combines bounded model checking, k-induction and invariant generation. The algorithm discharges these analyses to a sequence of incremental calls to a SAT or an SMT solver.

2 Architecture

2LS performs the following main steps, which are outlined in Fig. 1, and are explained below.

Front end. The *command-line front end* first configures 2LS 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. Being built upon the CPROVER infrastructure [4], 2LS uses *GOTO programs* as an 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. Similar to CBMC, 2LS 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 guarding against invalid pointer operations or memory leaks are inserted.

M. Chechik and J.-F. Raskin (Eds.): TACAS 2016, LNCS 9636, pp. 905–907, 2016.
DOI: 10.1007/978-3-662-49674-9_56

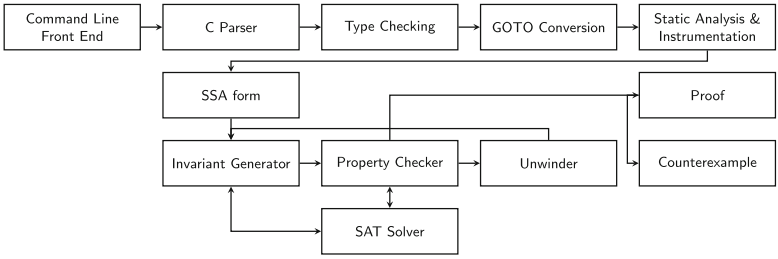

Fig. 1. 2LS architecture (using *k*I*k*I)

Middle end. 2LS performs a static analysis to derive the data flow equations for each function of the GOTO program. The result is a static single assignment (SSA) form in which loops have been cut at the back edges to the loop head. The effect of these cuts is a havocking of the variables modified in the loop at the loop head. This SSA is hence an over-approximation of the GOTO program. Subsequently, 2LS refines this over-approximation by computing invariants. 2LS performs local constant propagation and expression simplification to increase efficiency.

Back end. 2LS requires incremental back end solvers. Since support for incremental solving in SMT solvers is still lagging behind in comparison to SAT solvers, we use Glucose 4.0[1]. Consequently, as in CBMC, the SSA equation is translated into a CNF formula by bit-precise modelling of all expressions plus the Boolean guards. This formula is incrementally extended to perform invariant generation using template-based synthesis (see [2]; the competition version simply uses interval templates over numerical variables), to add further loop unwindings, and to the assertions for property checks. All this happens using a single solver instance so that information learned by the solver is never discarded. If a property check is satisfiable and model computed by the SAT solver does not take a path through an invariant (where over-approximation is used), then it corresponds to a path violating at least one of the assertions in the program under scrutiny. Subsequently, the model is translated back to a sequence of assignments to provide a human-readable counterexample. Conversely, if the property check is unsatisfiable, we have proven the assertions.

3 Strengths and Weaknesses

*k*I*k*I can provide both proofs as well as refutations using bit-precise algorithms. Refutations are essentially obtained via loop unwinding, whereas proofs are achieved by invariant generation as well as *k*-induction. This combination is quite powerful – 2LS won the gold medal in the Floats category, and is ranked

[1] http://www.labri.fr/perso/lsimon/glucose/#glucose-4.0.

2nd for the Loops benchmarks [1]. However, some benchmarks, e.g. those requiring reasoning about arrays contents or linked data structures, demand stronger invariants than we are currently able to infer. The monolithic analysis of the competition version does not support recursion, and there are limitations regarding irreducible control flow. Moreover, we observed issues with the counterexample witness GraphML output.

4 Tool Setup

The competition submission is based on 2LS version 0.3.[2] The full source code of the competition version is available at

http://www.cprover.org/svn/deltacheck/releases/2ls-0.3-sv-comp-2016.

Installation instructions are given in the file `COMPILING`. The executable `2ls` is in the directory `src/summarizer`. The competition version must be given the options `--k-induction` and `--competition-mode`. For all categories with a 32-bit memory model, use `--32`; for those with a 64-bit memory, use `--64`. There is no distinction between simple and precise memory model. In order to write the counterexample to file `CEX.graphml` add the option `--graphml-cex CEX.graphml`.[3]

Participation / Opt Out. 2LS competes in the following categories: Bit Vectors - BitVectorsReach, Floats, Integers and Control Flow, Overall, and Falsification.

5 Software Project

2LS is maintained by Peter Schrammel with patches supplied by the community. It is publicly available under a BSD-style license. The source code is available at http://www.cprover.org/2LS.

References

1. Beyer, D.: Reliable and reproducible competition results with benchexec and witnesses (report on sv-comp 2016). In: Chechik, M., Raskin, J.-F. (eds.) TACAS2016. LNCS, vol. 9636, pp. xx–yy. Springer, Heidelberg (2016)
2. Brain, M., Joshi, S., Kroening, D., Schrammel, P.: Safety verification and refutation by k-invariants and k-induction. In: Blazy, S., Jensen, T. (eds.) SAS 2015. LNCS, vol. 9291, pp. 145–161. Springer, Heidelberg (2015)
3. Chen, H.Y., David, C., Kroening, D., Schrammel, P., Wachter, B.: Synthesising interprocedural bit-precise termination proofs. In: Automated Software Engineering (ASE). ACM (2015)
4. 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)

[2] All relevant information for reproducing the results (including an archive containing the executable) can be found at http://sv-comp.sosy-lab.org/2016/systems.php.

[3] See BenchExec wrapper script `two_ls.py` and the benchmark definition file `2ls.xml`.