2LS: Memory Safety and Non-termination
(Competition Contribution)

Viktor Malík[1,3], Štefan Martiček[1,3], Peter Schrammel[1,2](✉), Mandayam Srivas[4], Tomáš Vojnar[3], and Johanan Wahlang[4]

[1] Diffblue Ltd., Oxford, UK
[2] University of Sussex, Brighton, UK
p.schrammel@sussex.ac.uk
[3] FIT BUT, IT4Innovations Centre of Excellence, Brno, Czech Republic
[4] Chennai Mathematical Institute, Chennai, India

Abstract. 2LS is a C program analyser built upon the CPROVER infrastructure. 2LS is bit-precise and it can verify and refute program assertions and termination. 2LS implements template-based synthesis techniques, e.g. to find invariants and ranking functions, and incremental loop unwinding techniques to find counterexamples and k-induction proofs. New features in this year's version are improved handling of heap-allocated data structures using a template domain for shape analysis and two approaches to prove program non-termination.

1 Overview

2LS is a static analysis and verification tool for sequential C programs that is based on an algorithm called kIkI (k-invariants and k-induction) [1], which combines bounded model checking, k-induction, and abstract interpretation into a single, scalable framework. 2LS relies on incremental SAT solving to employ all these techniques simultaneously in order to find proofs and refutations of assertions, as well as to perform termination analysis [2].

This year's competition version introduces a new *abstract shape domain* allowing 2LS to reason about properties of programs manipulating heap and dynamic data structures, and a *non-termination analysis*, which serves as a counterpart to the existing termination analysis and allows 2LS to prove non-termination of a program.

Architecture. 2LS is built upon the CPROVER infrastructure [3] and thus uses *GOTO programs* as the internal program representation. It first performs various static analyses and transformations of the program, including resolution of function pointers, points-to analysis, and insertion of assertions guarding against

The Czech authors were supported by the Czech Science Foundation project 17-12465S, the IT4IXS: IT4Innovations Excellence in Science project (LQ1602), and the FIT BUT internal project FIT-S-17-4014.
P. Schrammel—Jury member.

invalid pointer and memory operations. The analysed program is then translated into an acyclic, over-approximate single static assignment (SSA) form, in which loops are cut at the edges returning to the loop head. Subsequently, 2LS refines this over-approximation by computing inductive invariants in various abstract domains represented by parametrised logical formulae, so-called templates [1]. The competition version uses the interval domain for numerical variables and the new shape domain for pointer-typed variables described below.

The kIkI algorithm [1] operates on the SSA form, which is translated into a CNF formula over a bitvector representation of program configurations and given to a SAT solver. This formula is incrementally extended and amended to perform loop unwindings and abstract domain operations. The model returned by the solver is then used either to refine the predicates representing abstract values or to find a counterexample refuting the property to be checked. A more detailed description of the 2LS architecture can be found in the tool paper [7].

2 New Features

For SV-COMP'18, apart from various bug fixes and minor improvements, two major improvements of 2LS have been implemented: namely, a support for dealing with inductive list-like data structures and a support for proving program non-termination. Although 2LS supports certain interprocedural analyses, the competition version performs both analyses in a monolithic way, i.e. after inlining function calls. These improvements tackle weaknesses observed in previous years in the heap and memory safety categories, as well as they give a boost to 2LS' capabilities in non-termination analysis.

2.1 Memory Safety and Heap Invariants

To support shape analysis of dynamic data structures, a new abstract domain has been added to 2LS to express invariants describing heap configurations in the context of the bitvector logic used by 2LS [4]. The domain is based on recording (1) information about abstract heap objects pointed to by pointer variables and (2) information about reachability of abstract objects using *pointer access paths* [6]. Here, an abstract heap object represents all objects allocated at a given program location. The access paths then record which target abstract objects can be reached from a given source abstract object while going through some set of intermediary objects. For instance, the list in Fig. 1 would be encoded as `list` $= \&o_1 \wedge path(o_1, \texttt{nxt}, \{o_1, o_2\}, \texttt{NULL})$, meaning that `list` points to an object o_1 and there is a path from o_1 via `nxt` fields of abstract objects o_1 and o_2 to `NULL`. This representation is integrated as a template over pointer-typed variables

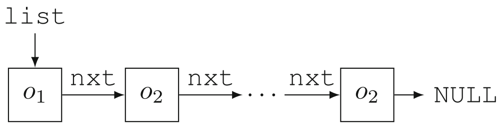

Fig. 1. A singly-linked list with nodes allocated at two different program locations.

and fields of dynamic objects into kIkI. The template is a parametrised logical formula. The parameters encode sets of memory objects that can be pointed by each pointer-typed variable as well as the set of paths that can lead from each dynamic object to other objects. 2LS computes these sets using an incremental SAT solver. This allows 2LS to prove or to refute assertions related to manipulation of dynamically linked data structures. The supported properties include null-pointer dereferencing, double-free, or memory leaks, for instance. Assertions for these properties are automatically instrumented into the code.

2.2 Proving Non-termination

Last year's version of 2LS provided a technique for proving termination based on linear lexicographic ranking functions synthesised using templates over bitvectors [2], but the tool was unable to prove non-termination except for trivial cases. For SV-COMP'18, two techniques for *proving non-termination* have been added [5]. Both of the approaches are relatively simple, yet appear to be reasonably efficient on the SV-COMP benchmarks.

The first approach is based on finding *singleton recurrence sets*. All loops are unfolded k times (with k being incrementally increased), followed by a check whether there is some loop L and a program configuration that can be reached at the head of L after both k' and k unwindings for some $k' < k$. Such a check can be easily formulated in 2LS as a formula over the SSA representation of programs with loops unfolded k times. This technique is able to find lasso-shaped executions in which a loop returns to the same program configuration every $k - k'$ iterations after k' initial iterations.

The second approach tries to reduce the number of unwindings by looking for loops that generate an *arithmetic progression* over every integer variable. More precisely, it looks for loops L for which each integer variable x can be associated with a constant c_x such that every iteration of L changes the value of x to $x + c_x$, keeping non-integer variables unchanged. Two queries are used to detect such loops: the first one asks whether there is a configuration $\overline{x}$ and a constant vector $\overline{c}$ (with the vectors ranging over all integer variables modified in the loop and constants from their associated bitvector domains) such that one iteration of L ends in the configuration $\overline{x} + \overline{c}$, while the second makes sure that there is no configuration $\overline{x}'$ over which one iteration of L would terminate in a configuration other than $\overline{x}' + \overline{c}$. If such a loop L and a constant vector $\overline{c}$ are found, non-termination of L can be proved as follows: First, we gradually exclude each configuration $\overline{x}$ reachable at the head of L for which there is some k such that L cannot be executed from $\overline{x} + k.\overline{c}$ (intuitively meaning that L cannot be executed $k + 1$ times from $\overline{x}$). Second, we check whether there remains some non-excluded configuration reachable at the head of L.

The termination and non-termination analyses are run in parallel, and the first definite answer is used. Among the new non-termination analyses, several rounds of unwinding are first tried with the singleton recurrence set approach. If that is not sufficient, the arithmetic progression approach is tried. If that does not succeed either, further rounds of unwinding with the former approach are run.

3 Strengths and Weaknesses

2LS' core algorithm, *k*I*k*I, is designed to be efficient for simultaneously finding proofs as well as refutations. Our SSA encoding allows us to introduce abstractions only at certain program points where these are necessary to infer the predicates required to construct proofs (e.g. invariants, ranking functions, recurrence sets). The remaining program is represented in a bit-precise large-block encoding.

Compared to the previous editions of the competition, 2LS is now able to reason about dynamic linked data structures. The approach used is currently able to handle various forms of linked lists (singly- or doubly-linked, a subset of nested or circular lists). However, more elaborate template domains will be required to handle other dynamic data structures such as trees and more general graph structures.

2LS' template-based approach to abstract interpretation allows easy combination of domains. We combine the heap domain with intervals over bitvectors, which is sufficient for many benchmarks. However, some benchmarks, e.g. those requiring reasoning about arrays contents, demand stronger invariants than we are currently able to infer.

The termination analysis scales well, but is currently limited to rather simple termination conditions (lexicographic linear). The newly implemented non-termination analyses are surprisingly effective on many SV-COMP termination benchmarks (638 out of 657 non-termination benchmarks proved). However, if a larger number of unwindings is needed the approach becomes quite inefficient. *k*I*k*I does not yet support recursion, which is another limitation, in particular w.r.t. the SV-COMP termination benchmark set, which contains a large number of recursive programs. The output of witnesses in the new categories (memory safety and termination) is still lacking (more than 550 points have been lost there).

4 Tool Setup

The competition submission is based on 2LS version 0.6.[1] Installation instructions are given in the file `COMPILING`. The executable `2ls` is in the directory `src/2ls`. See the `2ls` wrapper script (contained in the tarball) for the relevant command line options given to 2LS. The BenchExec script is called `two_ls.py` and the benchmark definition file `2ls.xml`. As a back end, the competition submission of 2LS uses Glucose 4.0. 2LS competes in all categories except Concurrency.

5 Software Project

2LS is maintained by Peter Schrammel with pull requests contributed by the community. It is publicly available under a BSD-style license. The source code is available at http://www.github.com/diffblue/2ls.

[1] Executable available at https://gitlab.com/sosy-lab/sv-comp/archives/tags/svcomp18.

References

1. 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). https://doi.org/10.1007/978-3-662-48288-9_9
2. Chen, H.Y., David, C., Kroening, D., Schrammel, P., Wachter, B.: Bit-precise procedure-modular termination proofs. TOPLAS, **40** (2017)
3. Clarke, E., Kroening, 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). https://doi.org/10.1007/978-3-540-24730-2_15
4. Malík, V.: Template-based synthesis of heap abstractions. Master's thesis, Brno University of Technology, Brno (2017)
5. Martiček, Š.: Synthesizing non-termination proofs from templates. Master's thesis, Brno University of Technology, Brno (2017)
6. Rinetzky, N., Bauer, J., Reps, T., Sagiv, M., Wilhelm, R.: A semantics for procedure local heaps and its abstractions. In: POPL. pp. 296–309. ACM (2005)
7. Schrammel, P., Kroening, D.: 2LS for program analysis (competition contribution). In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 905–907. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49674-9_56