CPAchecker with Support for Recursive Programs and Floating-Point Arithmetic
(Competition Contribution)

Matthias Dangl, Stefan Löwe, and Philipp Wendler

University of Passau

Abstract. We submit to SV-COMP'15 the software-verification framework CPAchecker. The submitted configuration is a combination of seven different analyses, based on explicit-value analysis, k-induction, predicate analysis, and concrete memory graphs. These analyses use concepts such as CEGAR, lazy abstraction, interpolation, adjustable-block encoding, bounded model checking, invariant generation, and block-abstraction memoization. Found counterexamples are cross-checked by a bit-precise analysis. The combination of several different analyses copes well with the diversity of the verification tasks in SV-COMP.

1 Software Architecture

CPAchecker is a software verification framework built on the concept of Configurable Program Analysis (CPA). One of the main design goals of the framework is to ease the development of new analyses and verification approaches. The CPAs available in the framework can be recombined on a per-demand basis by only passing the according configuration parameters to CPAchecker, without the need of changes in the implementation. Commonly needed tasks, like tracking of program counter, call stack, and function-pointer values, are also implemented as separate CPAs, and may assist the main CPAs, such as the predicate analysis. The framework provides a front-end based on the C-parser of the Eclipse CDT project (`http://www.eclipse.org/cdt/`), and an interface to SMT solvers (MathSAT5 (`http://mathsat.fbk.eu/`) in our submission) for solving and interpolation.

2 Verification Approach

The configuration used by CPAchecker in this year's SV-COMP is conceptually similar to last year [4]: a sequential combination of five analyses [2], as shown in Fig. 1, with the addition of an analysis based on k-induction using continuously-refined auxiliary invariants [1] and the limitation of the predicate analysis to a single ABE-l configuration. Each analysis runs for a predefined time, and if it does not return a result within the time bounds, the next analysis is started. Whenever one of the analyses finds a counterexample, it is cross-checked and if deemed infeasible, the analysis that is currently running gets terminated and the next one takes over.

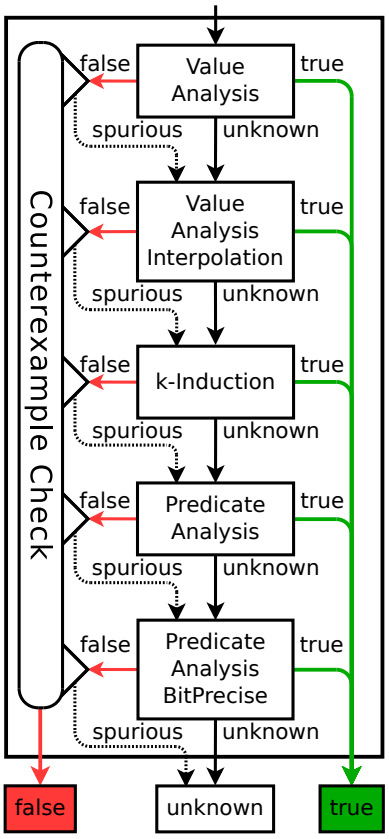

Fig. 1. Sequential combination to verify reachability properties

The time limit for each of the first three analyses is 60 s and the predicate analysis has no time limit. Similarly to last year, the counterexample checks are done by a bit-precise predicate analysis for the first four analyses, and the bounded model checker CBMC (`http://www.cprover.org/cbmc`) for the last analysis. In order to support the category "Floats", we have added support for precisely modeling the floating-point arithmetic to the predicate analysis (the value analysis cannot model non-deterministic values precisely enough to solve the programs in this category). This was made possible because the SMT solver MathSAT5 now supports floating-point arithmetic as an SMT theory, and CPAchecker leverages this by appropriately encoding most of the floating-point semantics of C in SMT formulae. Interpolation for floats is not yet supported, but not necessary for most programs in this category.

In two cases we deviate from the described configuration and use specialized approaches. As last year, we use a bounded analysis based on concrete memory graphs for verifying memory safety properties. For recursive programs, we use the predicate analysis with an extension of block-abstraction memoization [5], which uses two operators reduce and expand to remove information from the abstract state when entering a block (typically a function or loop body) if this information is not necessary inside the block, and restoring this information when leaving the block again. This allows a more efficient analysis and caching of the results for analyzed blocks. We extended this approach to support recursion (a recursive function call creates a new block). Together with an implementation of nested interpolation this allows the predicate analysis to analyze recursive programs with unbounded depth.

3 Strengths and Weaknesses

The sequential combination of several analyses covering different abstract domains allows CPAchecker to be competitive on a wide range of benchmarks. The bit-precise analyses help to minimize the number of wrong answers to only 0.6 % of all programs. Improvements over last year's version include handling of floating-point arithmetic using MathSAT, the addition of an analysis based on k-induction with

continuously-refined invariant generation [1], a novel interpolation routine for the value domain [3], and the use of an extension of block-abstraction memoization [5] as an analysis-independent framework for supporting recursive programs. Weaknesses of CPAchecker are the missing support for concurrent programs and for checking termination. An abstraction technique for memory graphs would allow a more efficient analysis in the categories "Arrays" and "MemorySafety".

4 Setup and Configuration

CPAchecker is available at `http://cpachecker.sosy-lab.org` and needs a Java 7 runtime environment. We submit version `1.3.10-svcomp15` for all categories. The command line for running CPAchecker is

```
scripts/cpa.sh -sv-comp15 -disable-java-assertions -heap 10000m -spec property.prp program.i
```

Please add the parameter `-64` for C programs assuming a 64-bit environment, and `-setprop cpa.predicate.handlePointerAliasing=false` for the simple memory model. For machines with less RAM, the amount of memory given to the Java VM needs to be set accordingly by the parameter `-heap`. CPAchecker will print the verification result and the name of the output directory to the console. In case CPAchecker finds a property violation the witness is written to the file named `witness.graphml` within this directory.

5 Project and Contributors

CPAchecker is an open-source project being developed by the members of the Software Systems Lab, led by Dirk Beyer, at the University of Passau. CPAchecker is used and extended by the members of the Institute for System Programming of the Russian Academy of Sciences, the Universities of Paderborn, Darmstadt and Vienna, as well as at Verimag, Grenoble. We would like to thank all contributors for their work on CPAchecker. The full list can be found at `http://cpachecker.sosy-lab.org`.

References

1. D. Beyer, M. Dangl, and P. Wendler. Combining k-induction with continuously-refined invariants. Technical Report MIP-1503, University of Passau, January 2015. arXiv:1502.00096.
2. D. Beyer, T. A. Henzinger, M. E. Keremoglu, and P. Wendler. Conditional model checking: A technique to pass information between verifiers. In *Proc. FSE*. ACM, 2012.
3. D. Beyer, S. Löwe, and P. Wendler. Domain-type-guided refinement selection based on sliced path prefixes. Technical Report MIP-1501, University of Passau, January 2015. arXiv:1502.00045.
4. S. Löwe, M. Mandrykin, and P. Wendler. CPAchecker with sequential combination of explicit-value analyses and predicate analyses (competition contribution). In *Proc. TACAS*, LNCS 8413, pages 392–394. Springer, 2014.
5. D. Wonisch and H. Wehrheim. Predicate analysis with block-abstraction memoization. In *Proc. ICFEM 2012*, LNCS 7635, pages 332–347. Springer, 2012.