CPAchecker with Sequential Combination of Explicit-Value Analyses and Predicate Analyses

(Competition Contribution)

Stefan Löwe[1], Mikhail Mandrykin[2], and Philipp Wendler[1]

[1] University of Passau, Germany
[2] Institute for System Programming of Russian Academy of Science, Russia

Abstract. CPAchecker is a framework for software verification, built on the foundations of Configurable Program Analysis (CPA). For the SV-COMP'14, we file a CPAchecker configuration that runs up to five analyses in sequence. The first two analyses of our approach utilize the explicit-value domain for modeling the state space, while the remaining analyses are based on predicate abstraction. In addition to that, a bit-precise counterexample checker comes into action whenever an analysis finds a counterexample. The combination of conceptually different analyses is key to the success of our verification approach, as the diversity of verification tasks is taken into account.

1 Software Architecture

CPAchecker, which is built on the foundations of Configurable Program Analysis (CPA), strives for high extensibility and reuse. As such, auxiliary analyses, such as tracking the program counter, modeling the call stack, and keeping track of function pointers, all of which is required for virtually any verification tool, are implemented as independent CPAs. The same is true for the main analyses, such as, e.g., the explicit-value analysis and the analysis based on predicate abstraction, which are also available as decoupled CPAs within CPAchecker.

All these CPAs can be enabled and flexibly recombined on a per-demand basis without the need of changing adjacent CPAs. Other algorithms, like CEGAR, counterexample checks, parallel or sequential combinations of analyses, as the one being filed to this year's SV-COMP'14, can be plugged together by simply passing the according configuration options to the CPAchecker framework.

CPAchecker, which is written in Java, uses the C parser of the Eclipse CDT project[3], and MathSAT5[4] for solving SMT formulae and interpolation queries.

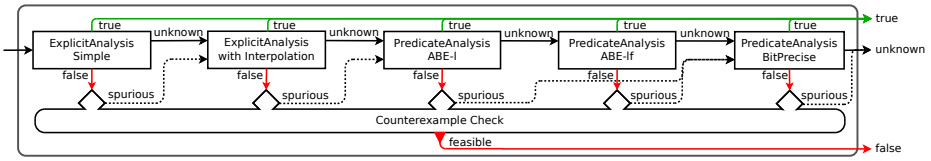

Fig. 1. Overview of the sequential combination used for reachability problems

2 Verification Approach

CPAchecker gets as input a specification and the source of a C program, which is then transformed into a control flow automaton (CFA) of the input program. During the analysis, this CFA is traversed, gradually building the abstract reachability graph (ARG). The nodes of the ARG represent the reachable states of the program, containing all relevant information, such as the program counter, the call stack and the information collected by the main CPAs, like explicit variable assignments or boolean combinations of predicates about program variables.

For reachability problems, we use a sequential combination [1] of up to five analyses using explicit-value analysis and predicate abstraction. The general approach of our sequential combination is as follows. Once any analysis in the sequence reports the verdict *true*, this result is returned. In case a counterexample is found and validated by a subsequent counterexample check, the verdict *false* is returned. If the counterexample is found to be spurious, or when the current analysis reaches a predefined time limit, the next analysis takes over.

The sequence starts with an explicit-value analysis without abstraction or refinement for 20 seconds. The motivation here is, that many control-flow intense programs can be solved with this approach in very little time. However, this simple analysis easily falls prey to state-space explosion. This is why a more sophisticated analysis of the same domain, including an abstract-refine loop [3], is started in case the first one does not come up with a result. Next in line are three analyses using predicate abstraction with adjustable block encoding [2]. The reason for switching to analyses that are conceptually different is motivated by the fact, that different programs have different characteristics. The third and fourth analyses model program variables as real variables and use only linear arithmetic. The first of these two configurations computes predicate abstractions only at loop heads (ABE-l) and runs for at most ten minutes. The second one additionally abstracts at function call and return sites (ABE-lf), and shows different performance characteristics. The final analysis, a bit-precise predicate analysis, is used if all previous analyses failed to provide a result (reasoning about bit vectors is too expensive to use it on all programs). In addition, an analysis similar to the last one, but lacking the abstract-refine loop, checks finite counterexamples found by any of the previously mentioned analyses. The

[3] `http://www.eclipse.org/cdt/`
[4] `http://mathsat.fbk.eu/`

bounded model checker CBMC[5] is used to check counterexamples of the last analysis for an even higher confidence in the result. For checking memory safety properties, we use a bounded analysis consisting of concrete memory graphs in combination with an instance of the explicit-value analysis mentioned above.

3 Strengths and Weaknesses

Similarly to our last years submissions, though far more sophisticated, the key idea of the submitted configuration is the combination of conceptually different analyses. In addition, the predicate analysis now has support for bit vectors, and also allows for more precise and efficient support for pointer aliasing by encoding possibly aliased memory locations with uninterpreted functions. However, CPAchecker lacks support for multi-threaded or recursive programs. Efficient tracking of heap memory remains an issue, yet solvable, e.g., by summarization.

4 Setup and Configuration

CPAchecker is available at `http://cpachecker.sosy-lab.org`. The submitted version is `1.2.11-svcomp14b`. The command line for running CPAchecker is

```
scripts/cpa.sh -sv-comp14 -disable-java-assertions -heap 10000m -spec property.prp program.i
```

Please add the parameter `-64` for C programs assuming a 64-bit environment. For machines with less RAM, the amount of memory given to the Java VM needs to be adjusted with the parameter `-heap`. CPAchecker will print the verification result and the name of the output directory to the console. Additional information (such as the error path) will be written to files in this directory.

5 Project and Contributors

CPAchecker is an open-source project led by Dirk Beyer from the Software Systems Lab at the University of Passau. Several other research groups use and contribute to CPAchecker, such as the Institute for System Programming of the Russian Academy of Sciences, the University of Paderborn and the University of Technology in Brno. 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, T. A. Henzinger, M. E. Keremoglu, and P. Wendler. Conditional model checking: A technique to pass information between verifiers. In *Proc. FSE*. ACM, 2012.
2. D. Beyer, M. E. Keremoglu, and P. Wendler. Predicate abstraction with adjustable-block encoding. In *Proc. FMCAD*, pages 189–197. FMCAD, 2010.
3. D. Beyer and S. Löwe. Explicit-state software model checking based on CEGAR and interpolation. In *Proc. FASE*, LNCS 7793, pages 146–162. Springer, 2013.

[5] `http://www.cprover.org/cbmc`