CPA-BAM: Block-Abstraction Memoization with Value Analysis and Predicate Analysis
(Competition Contribution)

Karlheinz Friedberger

University of Passau

Abstract. The software verification framework CPAchecker is built on basic approaches like CPA and CEGAR. The configuration for the SV-COMP'16 uses the concept of block-abstraction memoization and combines it with the parallel execution of value analysis and predicate analysis. The CEGAR loop uses a refinement strategy that prefers to refine the precision of the lightweight value analysis, such that the precision of the predicate analysis remains abstract and concise as long as possible. The usage of mature analyses like value analysis and predicate analysis allows us to bring together the potential of lazy abstraction and interpolation and the benefits of block-abstraction memoization.

1 Software Architecture

CPAchecker is a software verification framework that is build on Configurable Program Analysis (CPA) [1] and allows developers to easily integrate new analyses in a predefined way. CPAs are available for distinct tasks like tracking program locations, call stacks, function pointers, and assignments to variables. Also well-known approaches like value analysis and predicate analysis are integrated in CPAchecker in this manner. CPAs can be combined to form a more complex program analysis. The framework can execute a (configurable) algorithm like the CEGAR algorithm or a sequence of algorithms to verify reachability properties. There are analyses that support checking memory-safety properties and overflow detection, but this contribution does not use them.

CPAchecker is written in Java and uses the C-parser of the Eclipse CDT project (`https://eclipse.org/cdt/`). There are bindings for external libraries that allow to use BDDs, octagons, and SMT formulas. The predicate analysis in our configuration uses the SMT solver MathSAT5 (`http://mathsat.fbk.eu/`), because it supports bit-precise reasoning and interpolation for SMT formulae.

2 Verification Approach

Our configuration uses block-abstraction memoization (BAM) [4] to speedup the analysis. BAM divides the program into blocks and analyzes them separately. We choose functions as block size, such that a function call corresponds with a block entry and a function exit refers to a block exit, respectively. BAM aims for

a modular analysis, i.e. if a block has been already analyzed, the re-analysis of this block uses the stored result from a cache.

In SV-COMP'12, BAM was used with predicate analysis [3], and in SV-COMP'15, value analysis and predicate analysis were combined in a sequential way [2]. With several improvements and extensions done in the last year, we are now able to combine BAM not only with predicate analysis, but also with value analysis, interval analysis, and combinations thereof. We have defined and implemented the operators of BAM for the corresponding domains. For this year's SV-COMP, value analysis and predicate analysis are executed in a parallel manner to leverage the advantages of both approaches within the analysis with BAM.

BAM itself does not track any assignments or predicates over variables, but delegates this task to other more precise analyses. In our submission, the value analysis tracks assignments of variables and the predicate analysis uses predicates to analyze the program. Each of these two analyses is implemented as a CPA and uses a precision that determines which facts (assignments or predicates) are important for reasoning over the program, for example, for the reachability of a property violation. Figure 1 shows the CEGAR loop that updates the precisions during the refinements of the corresponding analysis. In CPAchecker, a reachability analysis uses the configured CPAs to examine the program until either a counterexample is reached or the program is analyzed completely. The second case refers to a program without any property violation. In the first case however, if the reachability analysis finds a counterexample, we check it for feasibility with both analyses in sequence. For a spurious counterexample one of the analyses should find the cause and perform the refinement, i.e. updating the corresponding precision. As the value analysis is more efficient in tracking many assignments, the counterexample is first checked with this analysis. As soon as one of the analyses cannot confirm the counterexample, the precision of this analysis is refined in order to exclude the spurious counterexample in the next iteration of the CEGAR loop. If both analyses confirm the counterexample, we report an error witness.

Recursive tasks are analyzed by an extension of BAM that was already used in SV-COMP'15. However, last year's contribution is improved by using the parallel combination of value analysis and predicate analysis in the way described above. Additionally, if no cached block abstraction can be reused before unrolling the recursive function up to a depth of 30, we abort the analysis of any deeper recursion. This bound is sufficient for the currently available recursive tasks.

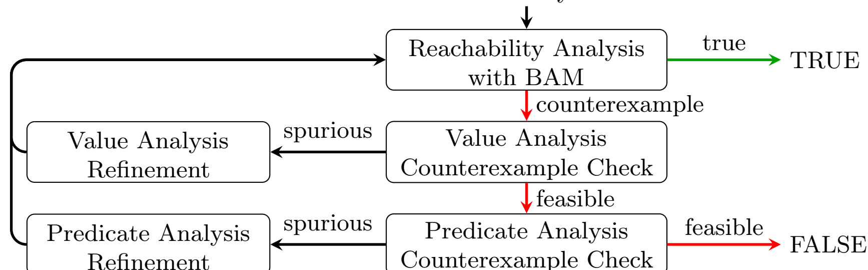

Fig. 1: Refinement for value analysis and predicate analysis in the CEGAR loop

3 Strengths and Weaknesses

The contributed configuration of BAM is most effective for solving large programs consisting of many functions, such that the benefit of using a cache justifies the overhead of BAM itself, i.e. the reuse of block abstractions outperforms the application of special operators in BAM. We report only a few wrong results for all tasks and none of them is a wrong proof. As our approach in CPAchecker uses its available analyses, some weaknesses are inherited. For example, value analysis and predicate analysis do not support large arrays or complex data structures. Our configuration does not check for memory-safety properties, termination or overflows, but simply ignores those cases and reports `UNKNOWN`.

4 Setup and Configuration

The CPAchecker project is available at `http://cpachecker.sosy-lab.org` and needs a Java 7 runtime environment. We submit version `1.4-svcomp16c` for participation in all categories. The tool can be downloaded from
`http://cpachecker.sosy-lab.org/CPAchecker-1.4-svcomp16c-unix.tar.bz2`.
CPAchecker has to be executed with the following command line:

```
scripts/cpa.sh -sv-comp16-bam -disable-java-assertions -heap 10000m -spec prop.prp program.i
```

The parameter `-64` should be added for C programs in categories assuming a 64-bit environment. CPAchecker will report the result of the verification to the console, including the violated property and the name of the output directory. In case of finding a property violation, the witness is written to the file `witness.graphml` within the output directory. CPAchecker can be executed using the tool-info module `cpachecker.py` and the benchmark definition `cpa-bam.xml` available at `http://sv-comp.sosy-lab.org/2016/systems.php`.

5 Project and Contributors

CPAchecker is licensed as an open-source project, headed by Dirk Beyer, and developed by members of the Software Systems Lab at the University of Passau. The framework is utilized and extended by an international group of developers. Our thanks go to all contributors for their work on CPAchecker, especially the members of the Institute for System Programming of the Russian Academy of Sciences for reporting several bugs in our implementation of block-abstraction memoization. More information about CPAchecker is provided at `http://cpachecker.sosy-lab.org`, where also a list of all contributors is available.

References

1. D. Beyer, T. A. Henzinger, and G. Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. In *Proc. CAV*, LNCS 4590, pages 504–518. Springer, 2007.
2. M. Dangl, S. Löwe, and P. Wendler. CPAchecker with support for recursive programs and floating-point arithmetic. In *Proc. TACAS*. Springer, 2015.
3. D. Wonisch. Block abstraction memoization for CPAchecker (competition contribution). In *Proc. TACAS*. Springer, 2012.
4. D. Wonisch and H. Wehrheim. Predicate analysis with block-abstraction memoization. In *Proc. ICFEM*, LNCS 7635, pages 332–347. Springer, 2012.