CPAchecker with Explicit-Value Analysis Based on CEGAR and Interpolation (Competition Contribution)

Stefan Löwe

University of Passau, Germany

Abstract. CPAchecker is a freely available software-verification framework, built on the concepts of Configurable Program Analysis (CPA). Within CPAchecker, several such CPAs are available, e.g., a Predicate-CPA, building on the predicate domain, as well as an Explicit-CPA, in which an abstract state is represented as an *explicit* variable assignment. In the CPAchecker configuration we are submitting, the highly efficient Explicit-CPA, backed by interpolation-based counterexample-guided abstraction refinement, joins forces with an auxiliary Predicate-CPA in a setup utilizing dynamic precision adjustment. This combination constitutes a highly promising verification tool, and thus, we submit a configuration making use of this analysis approach.

1 Software Architecture

CPAchecker is designed as an extensible framework for software verification, which is written in Java. The framework allows for parsing the input program into its internal data structures and provides interfaces to SMT solvers and interpolation procedures (e.g., MathSAT[1]). The paramount design decision of CPAchecker is separation of concerns, thus, each of the ready-made verification algorithms available within CPAchecker is implemented as a single CPA [1]. As these CPAs may be flexibly recombined on a per-demand basis, developing novel verifiers or reusing existing components for other domains is greatly facilitated.

2 Verification Approach

CPAchecker [2] represents the set of reachable states as an abstract reachability graph (ARG), which is built by successor computations along the edges of the program's control-flow automaton. The nodes of the ARG, representing sets of reachable program states, track all the relevant information, such as the program counter, the call stack, and the abstract data states of the main CPAs.

In contrast to our contribution from last year, which was doing software model checking via predicate abstraction, the main CPA in our configuration for this year, namely the Explicit-CPA, performs explicit-state software model checking

[1] `http://mathsat4.disi.unitn.it/`

in order to verify properties of a program. This approach has the advantage over more sophisticated techniques, like, e.g., approaches based on predicate abstraction, that the state representation is simpler and successor computation is more efficient. However, once applied to real-world code, representing each and every state of a program explicitly is bound to fall prey to the problem of state-space explosion, unless a proper abstraction technique is put in place. To this end, we extend the Explicit-CPA by abstraction and interpolation-based counterexample-guided abstraction refinement (CEGAR) [3].

There, the analysis starts with an initially empty precision, and, eventually, a counterexample will be found. The (in)feasibility of the (spurious) counterexample will be determined by a full-precision check, performed by our Explicit-CPA. If infeasible, the interpolation procedure will extract a refined, parsimonious precision to be used in the next iteration of the CEGAR loop. This abstract–refine approach circumvents the problem of state-space explosion in many cases, leading to improved run times and more solved instances than the naive approach.

However, due to the less expressive state representation of the explicit domain, it can occur that during explicit refinement, a spurious counterexample cannot be excluded by means of the explicit domain. To further improve the precision of our analysis, we add a Predicate-CPA in a dynamic precision adjustment approach [3]. There, the precision of the Predicate-CPA gets only refined in just those corner cases where the Explicit-CPA lacks expressiveness, and accordingly, the Predicate-CPA plays only an auxiliary, supporting role in the whole analysis.

Additionally, to limit the number of false positives reported by our verifier, once the analysis finds what it believes to be a real counterexample, the respective error path is given to CBMC.[2] Only if CBMC agrees with our result, the bug will be reported, otherwise, the analysis continues hunting for another bug.

3 Strengths and Weaknesses

The CPAchecker framework is striving for maximal reuse of existing components like the parser front-end, interfaces to the theorem provers, and, as described above, already existing CPAs. Hence, we rather adhere to software-engineering best practices instead of optimizing algorithms into a highly-tuned, but then also monolithic piece of software that becomes ever harder to maintain.

We expect our verifier to perform well where the property to be proven is strongly connected to the control flow. This is confirmed by the impressive results we obtained in the categories “ControlFlowInteger”, “SystemC” and, most notably, in the category “DeviceDrivers64”, where compared to the naive approach, our novel abstract–refine concept also has the most noticeable effect [3].

However, CPAchecker, and in particular the CPAs used in our configuration, do not provide support for the verification of properties in multi-threaded or recursive programs, while also lacking support for properties regarding memory safety. We wish to add more thorough handling of structures, unions, pointers, pointer aliasing and heap data structures into the analysis in the near future.

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

4 Setup and Configuration

CPAchecker is available under the Apache 2.0 license and both source code and binary releases are available for download at `http://cpachecker.sosy-lab.org`. Due to the fact that CPAchecker is written in Java, it is deployable on almost any platform. However, configurations depending on the predicate analysis currently work only under GNU/Linux, because the MathSAT library is available only for this platform. For the purpose of the software-verification competition, we submit version `1.1.10-svcomp13` of CPAchecker, with the configuration `sv-comp13--explitp-pred`. The command line for running this configuration is

```
./scripts/cpa.sh -sv-comp13--explitp-pred -heap 12000m
                 -disable-java-assertions path/to/sourcefile.cil.c
```

For C programs that assume a 64-bit environment (i.e., those in the category "Linux Device Drivers 64-bit") the parameter stated below needs to be added:

```
-setprop cpa.predicate.machineModel=LINUX64
```

For the category "Memory Safety", the property to verify is given by `-spec p` with `p` in {`valid-free`, `valid-deref`, `valid-memtrack`}. On machines with less than 16 GB RAM, we recommend to decrease the amount of memory given to the Java VM accordingly. CPAchecker will print the verification result and the name of the output directory to the console. Additional information, e.g., the error path, will be written to the respective files in this output directory.

5 Project and Contributors

The CPAchecker project is as an international open-source project, maintained at the University of Passau by the Software Systems Lab. It is used and extended by members of the Russian Academy of Science, the Technical University of Vienna, and the University of Paderborn. We would like to thank all contributors for their help and efforts spent on the CPAchecker project, and in particular, we would like to thank Dirk Beyer for maintaining the CPAchecker project.

References

1. Beyer, D., Henzinger, T.A., Théoduloz, G.: Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 504–518. Springer, Heidelberg (2007)
2. Beyer, D., Keremoglu, M.E.: CPAchecker: A Tool for Configurable Software Verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg (2011)
3. Beyer, D., Löwe, S.: Explicit-Value Analysis Based on CEGAR and Interpolation. Technical Report MIP-1205, University of Passau/ArXiv 1212.6542 (2012)