CPAchecker with Adjustable Predicate Analysis (Competition Contribution)

Stefan Löwe and Philipp Wendler

University of Passau, Germany

Abstract. CPAchecker is a freely available software-verification framework, built on the concepts of Configurable Program Analysis (CPA). CPAchecker integrates most of the state-of-the-art technologies for software model checking, such as counterexample-guided abstraction refinement (CEGAR), lazy predicate abstraction, interpolation-based refinement, and large-block encoding. The CPA for predicate analysis with adjustable-block encoding (ABE) is very promising in many categories, and thus, we submit a CPAchecker configuration that uses this analysis approach to the competition.

1 Verification Approach

Predicate analysis is a common approach to software verification, and tools like Blast and SLAM showed that it can be used effectively for software verification of medium sized programs. CPAchecker [2] constructs —like Blast— an abstract reachability graph (ARG) as a central data structure, by continuous successor computations along the edges of the control-flow automaton (CFA) of the program. The nodes of the ARG, representing sets of reachable program states, store relevant information like control-flow location, call stack, and, most importantly, the formulas that represent the abstract data states.

When single-block encoding (as implemented in Blast) is used, abstractions are computed for every single edge in the CFA. The major drawback of this approach is the large number of successor computations, each requiring expensive calls to a theorem prover. Furthermore, boolean abstraction is prohibitive for such a large number of successor computations, and only the more imprecise cartesian abstraction can be used.

Therefore, CPAchecker implements an approach called *adjustable-block encoding* [3], which completely separates the process of computing successors from the process of computing a predicate abstraction for a formula. The post operations in this approach (purely syntactically) assemble formulas for the strongest postcondition. Then, at certain points that can be chosen arbitrarily, the procedure applies an (expensive) computation of the predicate abstraction for a given abstract state. This method reduces the number of theorem-prover calls by effectively combining program blocks of arbitrary size into a single formula before computing an abstraction. Because the model checker now delegates much larger problems to the SMT solver (the formulas will contain a disjunction for

each control-flow join point inside a block), this technique is able to leverage the huge performance increase of SMT solvers being witnessed over the last decade. Experiments have shown that using adjustable blocks (e.g., loop-free blocks spanning across function calls) is orders of magnitudes faster than computing an abstraction for every single abstract state. Furthermore, the reduced number of abstractions (and refinements) makes it feasible to use the more expensive boolean abstraction, which makes the analysis more precise. This predicate analysis is wrapped in an algorithm for counterexample-guided abstraction refinement that uses Craig interpolation and lazy abstraction.

2 Software Architecture

CPAchecker is designed as an extensible framework for software verification and is written in Java. The framework provides the parsing of the input program (by using the C parser from the Eclipse CDT project[1]), interfaces to the SMT solver and interpolation procedures (using the SMT solver MathSAT4[2]), and the central verification algorithms. In CPAchecker, every analysis is implemented as a Configurable Program Analysis (CPA) [1], which makes it easier to implement new concepts (separation of concerns). Different CPAs can be flexibly combined on demand, enabling reuse of verification components. For the software verification competition, we use a configuration consisting of the CPAs for predicate analysis, program-counter tracking, call-stack analysis, and function-pointer analysis.

3 Strengths and Weaknesses

CPAchecker is meant as an infrastructure for implementing and evaluating innovative verification algorithms. Due to that, the framework is not focused on optimizing as much as possible, but instead advocates a strong compliance of the theoretical concepts and its respective implementation, thus easing the integration of new algorithms and concepts. Furthermore, the use of CPAs provides a high degree of re-usability, which makes the tool kit highly interesting for other groups, some of which already use CPAchecker to build their own extensions.

From a conceptional point of view, CPAchecker, and the CPA for predicate analysis in particular, lack support for checking multi-threaded or recursive programs. Further areas of improvement, well documented by the false positives given in the categories `DeviceDrivers` and `HeapManipulation`, include a more complete handling of pointers as well as proper support for more advanced constructs of the C programming language, like structs and unions.

4 Setup and Configuration

The source code for CPAchecker is released under the Apache 2.0 license and is available online at `http://cpachecker.sosy-lab.org`. Because the tool is written

[1] `http://www.eclipse.org/cdt/`
[2] `http://mathsat4.disi.unitn.it/`

in Java, it runs on almost any platform. The predicate analysis currently works only under GNU/Linux because the MathSAT library is available only for this platform. CPAchecker requires a Java 1.6 compatible JDK (e.g., OpenJDK), Ant 1.7, and the GNU Multiprecision library for C++ (required by MathSAT). The build process is performed by calling `ant` from the CPAchecker root directory. For the purpose of the software-verification competition, we use the `trunk` directory in revision 4569 and the configuration `-sv-comp12`. Thus the command line for running CPAchecker is

```
./scripts/cpa.sh -sv-comp12 -heap 12500m path/to/sourcefile.cil.c
```

For C programs that assume a 64-bit environment (i.e., those in the category `DeviceDrivers64`) the below parameter needs to be added:

```
-setprop cpa.predicate.machineModel=64-Linux
```

The programs in the category `DeviceDrivers` need the following additional option, because they make heavy use of pointers:

```
-setprop cpa.predicate.handlePointerAliasing=true
```

For general purpose verification tasks (outside the competition), we recommend the configuration `-predicateAnalysis` instead. Also, the amount of memory given to the Java VM needs to be adjusted on machines with less RAM. 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

The CPAchecker project was founded in 2007 by Dirk Beyer, and is hosted by the Software Systems Lab at the University of Passau. CPAchecker is an international open-source project which is used and contributed to by several research groups, e.g., the Russian Academy of Science, the Technical University of Vienna, and the University of Paderborn.

We thank all contributors for their help and efforts spent on the CPAchecker project. A complete list of contributors is provided on the project homepage at `http://cpachecker.sosy-lab.org/`. In particular, we would like to thank Dirk Beyer as the project leader and main architect, and Peter Häring, Michael Käufl, and Andreas Stahlbauer for their eager implementation work on CPAchecker as student assistants.

References

1. D. Beyer, T. A. Henzinger, and G. Théoduloz. Configurable software verification. In *Proc. CAV*, LNCS 4590, pages 504–518. Springer, 2007.
2. D. Beyer and M. E. Keremoglu. CPAchecker: A tool for configurable software verification. In *Proc. CAV*, LNCS 6806, pages 184–190. Springer, 2011.
3. D. Beyer, M. E. Keremoglu, and P. Wendler. Predicate abstraction with adjustable-block encoding. In *Proc. FMCAD*, pages 189–197. FMCAD, 2010.