CPAchecker 2.3 with Strategy Selection
(Competition Contribution)

Daniel Baier[⋆], Dirk Beyer, Po-Chun Chien, Marek Jankola, Matthias Kettl, Nian-Ze Lee, Thomas Lemberger, Marian Lingsch-Rosenfeld, Martin Spiessl, Henrik Wachowitz, and Philipp Wendler

http://cpachecker.sosy-lab.org

LMU Munich, Munich, Germany

Abstract. CPAchecker is a versatile framework for software verification, rooted in the established concept of *configurable program analysis*. Compared to the last published system description at SV-COMP 2015, the CPAchecker submission to SV-COMP 2024 incorporates new analyses for reachability safety, memory safety, termination, overflows, and data races. To combine forces of the available analyses in CPAchecker and cover the full spectrum of the diverse program characteristics and specifications in the competition, we use *strategy selection* to predict a sequential portfolio of analyses that is suitable for a given verification task. The prediction is guided by a set of carefully picked program features. The sequential portfolios are composed based on expert knowledge and consist of bit-precise analyses using k-induction, data-flow analysis, SMT solving, Craig interpolation, lazy abstraction, and block-abstraction memoization. The synergy of various algorithms in CPAchecker enables support for all properties and categories of C programs in SV-COMP 2024 and contributes to its success in many categories. CPAchecker also generates verification witnesses in the new YAML format.

1 Software Architecture

CPAchecker [10] is a flexible framework for automatic software verification based on the concept of *Configurable Program Analysis* (CPA) [9]. Abstract domains needed by a verification approach are represented as CPAs, and multiple CPAs can be combined in a modular fashion to achieve synergy. CPAchecker provides basic functionalities for program analysis, such as tracking the control flow or callstack, as standalone CPAs, which facilitate the implementation of new analyses. Through its modular architecture, a rich collection of verification algorithms [7, 12, 14, 24] has been implemented in CPAchecker, and its flexibility and extensibility have been evidenced by many research projects.

Operating Platform. CPAchecker is platform-independent as it is written in Java. However, its default SMT solver MathSAT5 [17] is bundled only for Linux. Thanks to the versatility of the used library JavaSMT [23], a different SMT solver can be chosen on other platforms.

⋆ Jury member

B. Finkbeiner and L. Kovács (Eds.): TACAS 2024, LNCS 14572, pp. 359–364, 2024.
https://doi.org/10.1007/978-3-031-57256-2_21

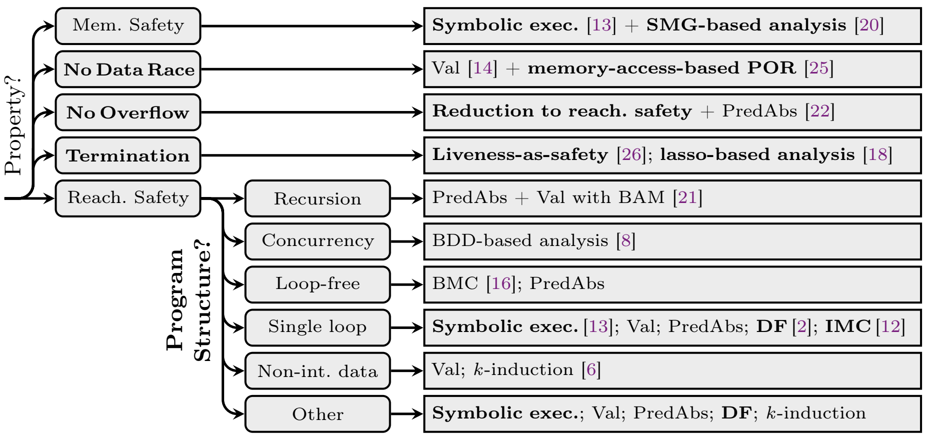
Property?
Mem. Safety
No Data Race
No Overflow
Termination
Reach. Safety
Program Structure?
Recursion
Concurrency
Loop-free
Single loop
Non-int. data
Other
Symbolic exec. [13] + SMG-based analysis [20]
Val [14] + memory-access-based POR [25]
Reduction to reach. safety + PredAbs [22]
Liveness-as-safety [26]; lasso-based analysis [18]
PredAbs + Val with BAM [21]
BDD-based analysis [8]
BMC [16]; PredAbs
Symbolic exec. [13]; Val; PredAbs; DF [2]; IMC [12]
Val; k-induction [6]
Symbolic exec.; Val; PredAbs; DF; k-induction

Fig. 1: Strategy selection based on the property to verify and program structure (New components since the last published system description [19] are marked in boldface. '+' and ';' denote component composition and sequential execution, respectively.)

Witnesses. CPAchecker produces correctness and violation witnesses for all properties where the corresponding witness type is already defined by the community. These are exported in the established GraphML format [4, 5] as well as in the new YAML format that is introduced with SV-COMP 2024.

2 Verification Approaches

To effectively solve the verification tasks from the heterogeneous benchmark set used in the competition, we need different verification strategies. Given a verification task, we select a suitable strategy with a two-level approach according to the property of the task and the structure of the program. A strategy could be a sequential portfolio of different verification techniques, each of which is assigned a time limit that is determined with expert knowledge. Figure 1 shows the selection procedure. The first-level selection is based on the property of the verification task. If the property is among *memory safety*, *no-dataraces*, *no-overflows*, or *termination*, a dedicated strategy is immediately assigned to solve the task. If the property is *reachability safety*, we further distinguish the program structure of a task into six classes by a set of carefully picked features, and a tailored strategy is invoked for each class. The details for each property and program structure are given below.

Memory Safety. Memory safety is checked by an unbounded analysis based on symbolic memory graphs (SMGs) [20]. It utilizes symbolic execution [13] to reason over non-concrete values, enabling us to verify the safety of low-level memory operations. The graph-based approach allows us to not only represent heap memory efficiently, but also to abstract linked memory structures (e.g., linked lists) that are created with low-level memory operations.

No Data Race. Data races are checked with a combination of value analysis (Val) [14], the thread handling from our concurrency analysis [8], and a CPA that

tracks read and write accesses to memory locations. We perform partial order reduction (POR) [25] over thread-local memory accesses to improve performance.
No Overflow. Overflows are checked with a CPA that adds additional constraints for overflow detection and a bit-accurate predicate abstraction (PredAbs) [7]. For recursive tasks we add block-abstraction memoization (BAM) [21, 27], which summarizes the input-output behavior of recursive functions.
Termination. Our termination strategy consists of two techniques. The first technique transforms liveness to a safety property [26]. With a combination of predicate and value analyses we check whether there exists some program state at a loop head that can be visited twice. If the program is recursive or the analysis reaches a time limit of 300 s, we switch to the second techniques, which uses ideas first implemented in Terminator [18]: We apply CPAchecker's predicate-based reachability analysis to detect potentially non-terminating program executions, called candidate *lassos*. A lasso consists of a *stem* (a finite program path) that is followed by a *loop* (a finite program path that describes a syntactic cycle in the program). Found candidate lassos are analyzed with the library LassoRanker [24] to synthesize termination and non-termination arguments. If a non-termination argument is found for at least one candidate lasso, violation of the termination property is reported. Otherwise, the analysis claims the program as terminating.
Reachability Safety. For the reachability of an error location, we tailor our verification strategy based on the structure of the program. If the program contains a *recursive* function, we apply block-abstraction memoization [21, 27] in combination with value analysis (Val) and predicate abstraction (PredAbs). If the program is *multi-threaded*, a concurrency analysis [8] that relies on binary decision diagrams (BDD) is applied. We set an upper limit of five threads for the analysis, and if this threshold is surpassed, the analysis is aborted. For non-recursive and single-threaded programs, we assign one of the four verification strategies in Fig. 1 according to the following structural features: the number of loops and whether the program contains non-integer data types, such as floating-point variables, arrays, or composite data structures [3]. The four strategies are all based on sequential combinations [19] of various bit-precise analyses with different time limits. For *loop-free* programs, we apply bounded model checking (BMC) [16] with a fallback to PredAbs [22]. For programs with a *single loop*, we apply a sequence of symbolic execution [13], Val [14], PredAbs [11], interval-based data-flow analysis (DF) [2], and interpolation-based model checking (IMC) [12]. For programs with multiple loops and *non-integer data types*, we apply Val and k-induction [6]. For all *other* programs, i.e., those with multiple loops but without non-integer data types, we apply a sequential portfolio of symbolic execution, Val, PredAbs, DF, and k-induction.

3 Strengths and Weaknesses

CPAchecker with strategy selection performed well in SV-COMP 2024 [1], winning the second place in category *Overall* and the first place in category *FalsificationOverall*. Notably, it produced 17 968 correct and confirmed results, more than any other participant, and outperformed the winner in category *Overall* by 32 %. CPAchecker is also robust: More than 96 % of its correct results were confirmed by witness validators, and it produced only 17 wrong results (0.06 % of all tasks).

CPAchecker won the third place in category *ReachSafety* by using various analyses orchestrated by strategy selection. For programs with non-integer data types, k-induction was the most effective analysis. In programs with loops, most alarms were found by symbolic execution, and most proofs were delivered by value analysis and predicate abstraction.[1]

The only categories without a medal for CPAchecker were *Termination*, *ConcurrencySafety*, and *MemSafety*. In particular, all wrong results in the category *MemSafety* are due to imprecise abstractions of nested lists. To alleviate them, we intend to improve the precision of our list abstraction and incorporate SMT-based array abstraction, which would make CPAchecker more effective in this category. To improve the termination analysis, we plan to make the analyses more cooperative and carry over partial proofs in the sequential combination. Additionally, CPAchecker needs improvements for finding invariants with quantifiers, which mainly affects verification tasks with large arrays.

4 Setup and Configuration

SV-COMP 2024 ran CPAchecker version 2.3 [15] on all categories with C programs. It runs on a standard GNU/Linux system with a Java 17 compatible runtime environment. To start CPAchecker, execute the following command:

```
scripts/cpa.sh -svcomp24 -benchmark -heap 10000M -timelimit 900s
    -spec property.prp program.i
```

For programs assuming a 64-bit memory model, append the argument `-64` to the command line. At the end of the execution, the verification result is printed to the console output and the witnesses are written to the files `witness.graphml` and `witness.yml` in the directory `output/`.

Note that the configuration `-svcomp24` is optimized specifically for the resource limits used in SV-COMP (15 GB of RAM and 15 min CPU time per task). For other use cases (e.g., with less RAM or a different time limit), please apply a different configuration (e.g., `-default`) and adjust the memory consumption with the command-line option `-heap` as described in the documentation.

5 Project and Contributors

More than 100 developers have contributed to CPAchecker, mainly from LMU Munich, TU Darmstadt, U Paderborn, U Passau, TU Prague, U Oldenburg, TU Vienna, ISP RAS, and several other universities and institutes. We would like to thank all contributors for their investment in CPAchecker. A complete list and more information about the project is available at `https://cpachecker.sosy-lab.org`. A list of bugs that CPAchecker found in the Linux kernel is also available.

[1] Note that the observations are specific to our sequential portfolios and influenced by the orders of analyses in the combination.

Data-Availability Statement. The tool is available at https://cpachecker.sosy-lab.org and the version used in SV-COMP 2024 is archived at Zenodo [15].

Funding Statement. This project was funded in part by the Deutsche Forschungsgemeinschaft (DFG) – 378803395 (ConVeY) and 496588242 (IDEFIX), and the LMU Postdoc Support Fund.

References

1. Beyer, D.: State of the art in software verification and witness validation: SV-COMP 2024. In: Proc. TACAS. LNCS , Springer (2024)
2. Beyer, D., Chien, P.C., Lee, N.Z.: CPA-DF: A tool for configurable interval analysis to boost program verification. In: Proc. ASE. pp. 2050–2053. IEEE (2023). https://doi.org/10.1109/ASE56229.2023.00213
3. Beyer, D., Dangl, M.: Strategy selection for software verification based on boolean features: A simple but effective approach. In: Proc. ISoLA. pp. 144–159. LNCS 11245, Springer (2018). https://doi.org/10.1007/978-3-030-03421-4_11
4. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M.: Correctness witnesses: Exchanging verification results between verifiers. In: Proc. FSE. pp. 326–337. ACM (2016). https://doi.org/10.1145/2950290.2950351
5. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Stahlbauer, A.: Witness validation and stepwise testification across software verifiers. In: Proc. FSE. pp. 721–733. ACM (2015). https://doi.org/10.1145/2786805.2786867
6. Beyer, D., Dangl, M., Wendler, P.: Boosting k-induction with continuously-refined invariants. In: Proc. CAV. pp. 622–640. LNCS 9206, Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_42
7. Beyer, D., Dangl, M., Wendler, P.: A unifying view on SMT-based software verification. J. Autom. Reasoning **60**(3), 299–335 (2018). https://doi.org/10.1007/s10817-017-9432-6
8. Beyer, D., Friedberger, K.: A light-weight approach for verifying multi-threaded programs with CPAchecker. In: Proc. MEMICS. vol. 233, pp. 61–71. EPTCS (2016). https://doi.org/10.4204/EPTCS.233.6
9. Beyer, D., Henzinger, T.A., Théoduloz, G.: Configurable software verification: Concretizing the convergence of model checking and program analysis. In: Proc. CAV. pp. 504–518. LNCS 4590, Springer (2007). https://doi.org/10.1007/978-3-540-73368-3_51
10. Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: Proc. CAV. pp. 184–190. LNCS 6806, Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_16
11. Beyer, D., Keremoglu, M.E., Wendler, P.: Predicate abstraction with adjustable-block encoding. In: Proc. FMCAD. pp. 189–197. FMCAD (2010). https://dl.acm.org/doi/10.5555/1998496.1998532
12. Beyer, D., Lee, N.Z., Wendler, P.: Interpolation and SAT-based model checking revisited: Adoption to software verification. arXiv/CoRR **2208**(05046) (July 2022). https://doi.org/10.48550/arXiv.2208.05046
13. Beyer, D., Lemberger, T.: CPA-SymExec: Efficient symbolic execution in CPAchecker. In: Proc. ASE. pp. 900–903. ACM (2018). https://doi.org/10.1145/3238147.3240478
14. Beyer, D., Löwe, S.: Explicit-state software model checking based on CEGAR and interpolation. In: Proc. FASE. pp. 146–162. LNCS 7793, Springer (2013). https://doi.org/10.1007/978-3-642-37057-1_11

15. Beyer, D., Wendler, P.: CPAchecker release 2.3 (unix) (2023). https://doi.org/10.5281/zenodo.10203297
16. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Proc. TACAS. pp. 193–207. LNCS 1579, Springer (1999). https://doi.org/10.1007/3-540-49059-0_14
17. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT solver. In: Proc. TACAS. pp. 93–107. LNCS 7795, Springer (2013). https://doi.org/10.1007/978-3-642-36742-7_7
18. Cook, B., Podelski, A., Rybalchenko, A.: Terminator: Beyond safety. In: Proc. CAV. pp. 415–418. LNCS 4144, Springer (2006). https://doi.org/10.1007/11817963_37
19. Dangl, M., Löwe, S., Wendler, P.: CPAchecker with support for recursive programs and floating-point arithmetic (competition contribution). In: Proc. TACAS. pp. 423–425. LNCS 9035, Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_34
20. Dudka, K., Peringer, P., Vojnar, T.: Byte-precise verification of low-level list manipulation. In: Proc. SAS. pp. 215–237. LNCS 7935, Springer (2013). https://doi.org/10.1007/978-3-642-38856-9_13
21. Friedberger, K.: CPA-BAM: Block-abstraction memoization with value analysis and predicate analysis (competition contribution). In: Proc. TACAS. pp. 912–915. LNCS 9636, Springer (2016). https://doi.org/10.1007/978-3-662-49674-9_58
22. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: Proc. POPL. pp. 232–244. ACM (2004). https://doi.org/10.1145/964001.964021
23. Karpenkov, E.G., Friedberger, K., Beyer, D.: JavaSMT: A unified interface for SMT solvers in Java. In: Proc. VSTTE. pp. 139–148. LNCS 9971, Springer (2016). https://doi.org/10.1007/978-3-319-48869-1_11
24. Leike, J., Heizmann, M.: Ranking templates for linear loops. Logical Methods in Computer Science **11**(1) (2015). https://doi.org/10.2168/LMCS-11(1:16)2015
25. Peled, D.: Ten years of partial order reduction. In: Proc. CAV. pp. 17–28. Springer (1998). https://doi.org/10.1007/BFb0028727
26. Schuppan, V., Biere, A.: Liveness checking as safety checking for infinite state spaces. Electr. Notes Theor. Comput. Sci. **149**(1), 79–96 (2006). https://doi.org/10.1016/j.entcs.2005.11.018
27. Wonisch, D., Wehrheim, H.: Predicate analysis with block-abstraction memoization. In: Proc. ICFEM. pp. 332–347. LNCS 7635, Springer (2012). https://doi.org/10.1007/978-3-642-34281-3_24