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.
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-data races, 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 not only to 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) \cite{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) \cite{7}. For recursive tasks we add block-abstraction memoization (BAM) \cite{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 \cite{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 \cite{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 \cite{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 \cite{21,27} in combination with value analysis (Val) and predicate abstraction (PredAbs). If the program is multi-threaded, a concurrency analysis \cite{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 \cite{3}. The four strategies are all based on sequential combinations \cite{19} of various bit-precise analyses with different time limits. For loop-free programs, we apply bounded model checking (BMC) \cite{16} with a fallback to PredAbs \cite{22}. For programs with a single loop, we apply a sequence of symbolic execution \cite{13}, Val \cite{14}, PredAbs \cite{11}, interval-based data-flow analysis (DF) \cite{2}, and interpolation-based model checking (IMC) \cite{12}. For programs with multiple loops and non-integer data types, we apply Val and k-induction \cite{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 \cite{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


Open Access. This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution, and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made. The images or other third party material in this chapter are included in the chapter’s Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter’s Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.