CPAchecker 4.0 as Witness Validator
(Competition Contribution)

Dirk Beyer and Marian Lingsch-Rosenfeld[⋆]
http://cpachecker.sosy-lab.org
LMU Munich, Munich, Germany

Abstract. CPAchecker is a tool for software verification, witness validation, and test-case generation, based on the concept of *configurable program analysis*. One of its main applications is to validate correctness and violation witnesses in versions 1.0 and 2.0. The witness validation is achieved by strengthening a selection of verification algorithms using the information from the witness. Due to the modular approach of CPAchecker, extending its verification analyses for witness validation can be easily done. Similar to CPAchecker's verification approach, witness validation uses a selection of analyses dependent on the witness type, the specification, and program features. To validate correctness witnesses, CPAchecker uses k-induction and predicate abstraction to verify that the invariants from the witness hold and the correctness of the program can be proven. To validate violation witnesses, CPAchecker uses predicate abstraction, value analysis, SMGs, and BDDs. CPAchecker's many verification algorithms make it a versatile and successful tool for witness validation.

1 Software Architecture

CPAchecker [13] is a tool for automatic software verification, *witness validation*, and test-case generation. It is possible for it to cover these different use-cases due to its modular architecture, based on the concept of *Configurable Program Analysis* (CPA) [12]. Each CPA represents information relevant to an analysis, for example an abstract domain, control-flow information, or an automaton, and can be combined with other CPAs in a modular manner. Exchanging information with other CPAs is done through a well-defined interface called the *strengthening* operator, which is used to incorporate information from one abstract state into the another. This modular approach allows any analysis, of which many exist [2, 4, 5, 7, 8, 15, 16], to be used for witness validation. The information from the witness can be encoded as a CPA, simplifying the adaptation of existing analyses for witness validation. One major challenge for witness validation is relating the input program to its internal representation as a control-flow automaton (CFA). In particular, this challenge occurs when validating witnesses in version 2.0, which are defined purely on the input program [1]. CPAchecker analyses such witnesses by transparently keeping track of the abstract-syntax-tree elements generated during parsing and their correspondence to the CFA nodes and edges.

[⋆] Jury member

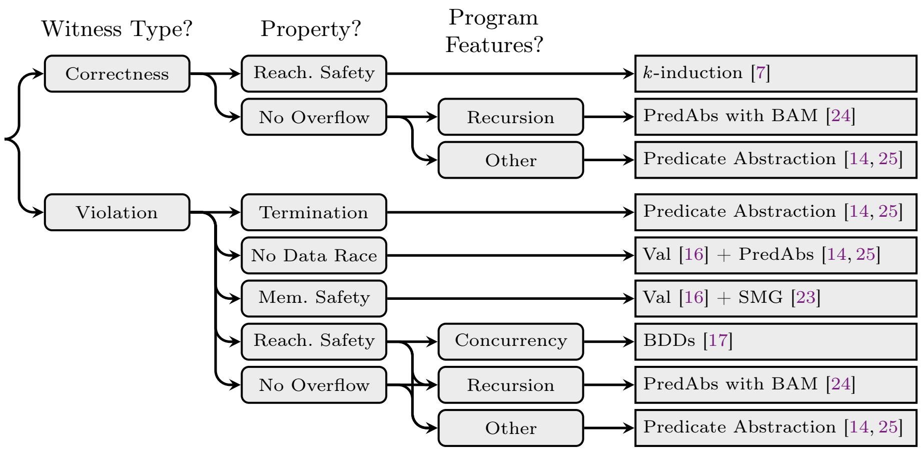

Fig. 1: Algorithm selection based on the witness type, property, and program

2 Validation Approach

When participating as a verifier, CPAchecker uses different verification algorithms [3, 6], since each has specific strengths and supports different properties. To validate witnesses, we choose only the most mature analyses and adapt the selection process to consider the witness type (violation or correctness), specification, and program features for the selected analyses. Figure 1 shows the selection process. The chosen verification algorithm is strengthened with the information from the witness. The analyses and how they interact with the witness are described in the following sections for all validation categories.

2.1 Correctness Witnesses

Validation of correctness witnesses is based on strengthening the chosen analysis using the invariants provided by the witness. Additionally, the invariants are added as proof-goals in order to validate their correctness.

Reachability Safety. CPAchecker uses one of its best-performing and mature analyses for reachability safety, k-Induction [7] augmented with invariants from the witness. First the invariants are matched to their corresponding nodes of the control-flow automaton (CFA). For witnesses 1.0 we do this by a reachability analysis only dependent on the control-flow. For witnesses 2.0 we directly match the location of invariants to their corresponding CFA nodes. Afterwards, k-Induction first verifies that the invariants are correct and then uses them to prove the safety property, while increasing k as necessary. This means that all invariants in the witness need to be k-inductive for some k in order to be validated.

No Overflow. To validate witnesses for proofs showing the absence of overflows, CPAchecker uses *predicate abstraction* [14]. The information in the witness is encoded using an invariant-injection automaton, which is traversed simultaneously to the state-space exploration and strengthens the analysis. Recursive programs are handled by block-abstraction memoization (BAM) [9, 10, 24, 26], which summarizes the input-output behavior of recursive functions.

2.2 Violation Witnesses

Due to the modular nature of CPAchecker, any analysis, except for the validation of non-termination witnesses (see below), can easily be combined with an automaton, represented as a CPA, that *restricts* the state space and *strengthens* the analysis with assumptions. The automaton is based on the witness and guides the analysis towards the violation by exploring it simultaneously to the state-space. We *restrict* the state space by avoiding any transition that does not match the witness. For example, if the `then` branch is desired by the witness, then the analysis stops the exploration of the `else` branch. We *strengthen* the analysis by adding the assumptions to the abstract state. The assumptions are added through a common interface, the strengthening operator [8], which each analysis needs to implement to successfully make use of the information.

Termination. To validate a non-termination witness, we need to show that (1) the stem of the witness can reach the recurrence condition and that (2) whenever a state fulfills the recurrence condition, it returns to a state at the loop-head fulfilling the recurrence condition. Both of these checks are expressed as reachability analyses performed by predicate abstraction [14].

No Data Race. To validate a witness showing that there is a data race, we use an analysis based on value analysis [16] and predicate abstraction [14]. The value analysis uses the assumptions in the witness to determine additional concrete values to be tracked. The predicate abstraction adds the assumptions to the path formula to compute what predicates are valid in the successor state. The same strengthening is done whenever information from the witness needs to be added to either the value analysis or predicate abstraction for any other property.

Memory Safety. For memory safety, a combination of value analysis and symbolic memory graphs (SMGs) is used to explore the state-space while keeping track of the memory structure. In this case, the assumptions in the witness are used to strengthen the abstract state of the value analysis.

Reachability Safety. Predicate abstraction [14] is used to validate violation witnesses for reachability safety. It is aided by BAM [10, 24, 26] for tasks with recursive functions. The assumptions inside the witness are used to strengthen the predicate abstract state. If the task contains concurrency, an analysis based on BDDs [11, 18] is used, which currently ignores the assumptions in the witness.

No Overflow. To validate no-overflow violation witnesses, predicate abstraction [14] is used. The assumptions are used to strengthen the predicate abstract state. To handle tasks with recursion we add BAM [10, 24, 26].

3 Strengths and Weaknesses

CPAchecker validator performed well in SV-COMP 2025 [19], consistently ranking in the top three in almost all categories with a well-defined witness format, the only exceptions being validation of handcrafted violation witnesses in version 2.0, validation of software-systems correctness witnesses in version 1.0, and validation of reach-safety and no-overflow violation witnesses in version 1.0. In particular, it is one of the only three validators participating in all categories for which a witness format exists, together with UAutomizer and MetaVal. In general, CPAchecker performed similarly when validating witnesses in version 1.0 and in version 2.0, since both are encoded in the same manner as a CPA to combine them with the verification analyses.

Notably CPAchecker performed best in the validation of correctness witnesses, but lagged behind other tools when validating violation witnesses. This is due to CPAchecker using primarily the predicate analysis for validation, which is better in finding proofs than bugs. In general, since the analyses used for validation are the same as for verification, only strengthened with the witnesses, they have the same strengths and weaknesses.

We do not use the parallel portfolio of different analyses that boosts CPAchecker 4.0 as a verifier, we rather select one mature analyses, in order to increase the confidence in the validation result. In the future, a portfolio of other mature analyses could be used, improving the performance without compromising the confidence in the results.

4 Setup and Configuration

CPAchecker validator in version 4.0 [20] was used in SV-COMP 2025. It runs on any system with a Java 17 compatible runtime environment, though its default SMT solver MathSAT5 [22] is bundled only for Linux. For platforms other than GNU/Linux, we recommend using the provided container image [21]. To validate a witness using CPAchecker, execute the following command:

```
bin/cpachecker --witnessValidation --benchmark --heap 10000M
    --timelimit limit --32 --witness witness.{yml,graphml}
    --spec property.prp program.i
```

SV-COMP uses a timelimit of 900 s for correctness witnesses and 90 s for violation witnesses, though it will work with other/no time limits. Replace `--32` by `--64` for programs that assume a 64-bit memory model. More information on how to run CPAchecker is available on its website `https://cpachecker.sosy-lab.org` and tutorial paper [2].

5 Project and Contributors

CPAchecker validator builds upon existing verification algorithms with additional support for handling witnesses. The success of CPAchecker is the result of contributions from over 100 developers, primarily from institutions such as LMU Munich, TU Darmstadt, University of Paderborn, University of Passau, TU Prague, University of Oldenburg, TU Vienna, ISP RAS, and numerous other universities and research institutes. We extend our utmost gratitude to all contributors to CPAchecker which made it possible to have such a successful validator. A complete list of contributors and further details about the project can be found at `https://cpachecker.sosy-lab.org`.

Data-Availability Statement. The tool is available at `https://cpachecker.sosy-lab.org` and the version used in SV-COMP 2025 is archived at Zenodo [20].

Funding Statement. This project was funded in part by the Deutsche Forschungsgemeinschaft (DFG) – 378803395 (ConVeY) and 496588242 (IdeFix).

References

1. Ayaziová, P., Beyer, D., Lingsch-Rosenfeld, M., Spiessl, M., Strejček, J.: Software verification witnesses 2.0. In: Proc. SPIN. pp. 184–203. LNCS 14624, Springer (2024). `https://doi.org/10.1007/978-3-031-66149-5_11`
2. Baier, D., Beyer, D., Chien, P.C., Jakobs, M.C., Jankola, M., Kettl, M., Lee, N.Z., Lemberger, T., Lingsch-Rosenfeld, M., Wachowitz, H., Wendler, P.: Software verification with CPAchecker 3.0: Tutorial and user guide. In: Proc. FM. pp. 543–570. LNCS 14934, Springer (2024). `https://doi.org/10.1007/978-3-031-71177-0_30`
3. Baier, D., Beyer, D., Chien, P.C., Jankola, M., Kettl, M., Lee, N.Z., Lemberger, T., Lingsch-Rosenfeld, M., Spiessl, M., Wachowitz, H., Wendler, P.: CPAchecker 2.3 with strategy selection (competition contribution). In: Proc. TACAS (3). pp. 359–364. LNCS 14572, Springer (2024). `https://doi.org/10.1007/978-3-031-57256-2_21`
4. Beyer, D., Chien, P.C., Jankola, M., Lee, N.Z.: A transferability study of interpolation-based hardware model checking for software verification. Proc. ACM Softw. Eng. **1**(FSE) (2024). `https://doi.org/10.1145/3660797`
5. 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`
6. 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`. `https://www.sosy-lab.org/research/pub/2018-ISoLA.Strategy_Selection_for_Software_Verification_Based_on_Boolean_Features.pdf`
7. 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`
8. 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`
9. Beyer, D., Friedberger, K.: Domain-independent multi-threaded software model checking. In: Proc. ASE. pp. 634–644. ACM (2018). `https://doi.org/10.1145/3238147.3238195`
10. Beyer, D., Friedberger, K.: Domain-independent interprocedural program analysis using block-abstraction memoization. In: Proc. ESEC/FSE. pp. 50–62. ACM (2020). `https://doi.org/10.1145/3368089.3409718`
11. Beyer, D., Friedberger, K.: Violation witnesses and result validation for multithreaded programs. In: Proc. ISoLA (1). pp. 449–470. LNCS 12476, Springer (2020). `https://doi.org/10.1007/978-3-030-61362-4_26`
12. 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`

13. 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
14. 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
15. Beyer, D., Lee, N.Z., Wendler, P.: Interpolation and SAT-based model checking revisited: Adoption to software verification. J. Autom. Reasoning **69** (2025). https://doi.org/10.1007/s10817-024-09702-9, preprint: https://doi.org/10.48550/arXiv.2208.05046
16. 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. https://www.sosy-lab.org/research/pub/2013-FASE.Explicit-State_Software_Model_Checking_Based_on_CEGAR_and_Interpolation.pdf
17. Beyer, D., Stahlbauer, A.: BDD-based software model checking with CPAchecker. In: Proc. MEMICS. pp. 1–11. LNCS 7721, Springer (2013). https://doi.org/10.1007/978-3-642-36046-6_1. https://www.sosy-lab.org/research/pub/2013-MEMICS.BDD-Based_Software_Model_Checking_with_CPAchecker.pdf
18. Beyer, D., Stahlbauer, A.: BDD-based software verification: Applications to event-condition-action systems. Int. J. Softw. Tools Technol. Transfer **16**(5), 507–518 (2014). https://doi.org/10.1007/s10009-014-0334-1
19. Beyer, D., Strejček, J.: Report on SV-COMP 2025. In: Proc. TACAS. LNCS , Springer (2025)
20. Beyer, D., Wendler, P.: CPAchecker release 4.0. Zenodo (2024). https://doi.org/10.5281/zenodo.14203369
21. Beyer, D., Wendler, P.: CPAchecker release 4.0 (image). Zenodo (2024). https://doi.org/10.5281/zenodo.14209310
22. 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
23. 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
24. 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
25. 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
26. 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