CPV: A Circuit-Based Program Verifier (Competition Contribution)

Po-Chun Chien(✉)[⋆] and Nian-Ze Lee(✉)

LMU Munich, Munich, Germany
{po-chun.chien,nian-ze.lee}@sosy.ifi.lmu.de

Abstract. We submit to SV-COMP 2024 CPV, a circuit-based software verifier for C programs. CPV utilizes sequential circuits as its intermediate representation and invokes hardware model checkers to analyze the reachability safety of C programs. As the frontend, it uses Kratos2, a recently proposed verification tool, to translate a C program to a sequential circuit. As the backend, state-of-the-art hardware model checkers ABC and AVR are employed to verify the translated circuits. We configure the hardware model checkers to run various analyses, including IC3/PDR, interpolation-based model checking, and k-induction. Information discovered by hardware model checkers is represented as verification witnesses. In the competition, CPV achieved comparable performance against participants whose intermediate representations are based on control-flow graphs. In the category *ReachSafety*, it outperformed several mature software verifiers as a first-year participant. CPV manifests the feasibility of sequential circuits as an alternative intermediate representation for program analysis and enables head-to-head algorithmic comparison between hardware and software verification.

Keywords: Software verification · Hardware verification · C programs · Sequential circuits · Btor2 · Aiger · Tool combination · Portfolio

1 Introduction

Software verification is challenging. Numerous intermediate representations have been proposed to capture diverse software features and facilitate the development of program verifiers. Among various encodings of a state-transition system, *sequential circuits*, consisting of memory elements to represent states and combinational logic to capture state transitions, are commonly used in the hardware-verification domain, and abundant techniques have been invented for *hardware model checking*. Using sequential circuits as its intermediate representation, our tool CPV aims to answer the following question: *Are sequential circuits feasible as an alternative foundation to build software verifiers?* While previous studies on translating and cross-applying verification techniques for hardware and software exist [1, 2, 3, 4], to our knowledge, no participants in SV-COMP had used sequential circuits as their intermediate representations. This competition report outlines the software

⋆ Jury member

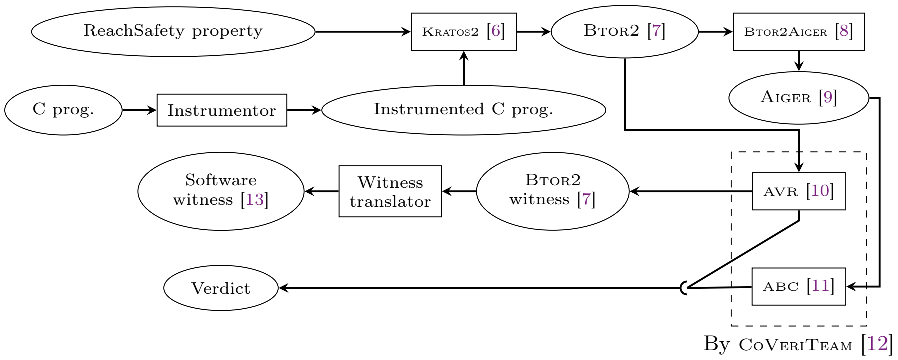

Fig. 1: Software architecture of CPV

architecture and verification approach of CPV and discusses its results against other mature program analyzers in SV-COMP 2024 [5].

2 Software Architecture

The software architecture of CPV is depicted in Fig. 1. Its verification workflow is divided into two stages: (1) In the frontend (the upper half of Fig. 1), an input C program with a reachability-safety property is first instrumented to allow for witness translation (details in Sect. 3) and then translated into a word-level Btor2 [7] circuit by Kratos2 [6]. The Btor2 language [7] is used in the Hardware Model Checking Competitions [14, 15], and many powerful hardware model checkers support this format. A bit-level Aiger [9] circuit is also generated by the tool Btor2Aiger [8]; (2) In the backend (the lower half of Fig. 1), CPV invokes hardware model checkers AVR [10] and ABC [11] to verify the translated circuits. Btor2 verification witnesses produced for the circuits are translated to software witnesses in the GraphML format [13] for the original program. CPV configures and executes the backend model checkers (either solely or as portfolios) via CoVeriTeam [12], a library for cooperative verification [16]. Thanks to the versatility of CoVeriTeam, it is convenient to choose the verification algorithms used by AVR and ABC, and the pool of the backend verifiers in CPV can be expanded with little effort.

3 Verification Approach

The approach of CPV is to translate a program into a circuit and applies hardware model checking to the translated verification task. To generate software-verification witnesses, CPV instruments an input program before translating it to a circuit, such that the information contained in a witness for the translated circuit can be mapped back to the original program.

Program-to-Circuit Translation. CPV utilizes Kratos2 [6] as its frontend to translate a verification task of a C program into a word-level sequential circuit in the Btor2 format [7]. Kratos2 applies *large-block encoding* [17] and introduces a symbolic program counter to fold the summarized program into a state-transition system. Executing a maximal loop-free block of the program is a one-step transition in the system. A call to an external function that models nondeterministic input values to the program, e.g., the functions `__VERIFIER_nondet_X()` in SV-COMP, is represented as an external input to the state-transition system. We configure Kratos2 to export the system as a sequential circuit in the Btor2 format because Btor2 is the prevailing format for hardware model checking. In order to leverage bit-level hardware model checkers, CPV additionally invokes Btor2Aiger [8] to translate the word-level Btor2 circuit into the Aiger format [9]. Currently, CPV supports the property of reachability safety. Violation to the reachability-safety property of the input program is captured by a circuit output asserting the equivalence between the symbolic program counter and the error location.

Hardware Model Checking. CPV employs AVR [10] and ABC [11], two state-of-the-art hardware model checkers for word-level Btor2 and bit-level Aiger circuits, respectively, to analyze the translated circuits. A hardware model checker decides whether the translated circuit has a computation trace to assert its circuit output, which indicates the error location in the original program is reachable. In this case, the verification verdict is `false`, and the original program is unsafe. If there is no trace to assert the circuit output, the verdict is `true`, and the original program is safe.

To achieve synergy, we combine the strengths of various hardware-verification algorithms, including property-directed reachability (PDR) [18, 19], interpolation-based model checking (IMC) [20], k-induction (KI) [21], and bounded model checking (BMC) [22]. For the tasks that can be translated into Aiger circuits,[1] a sequential portfolio of AVR-KI, AVR-PDR, ABC-IMC, ABC-PDR, and AVR-BMC is applied. A pre-determined time limit is imposed on each component in the portfolio by CoVeriTeam. AVR is executed first in the portfolio because it can produce a Btor2 witness [7] for the translated circuit if a property violation was found, whereas ABC does not export witnesses in a standardized format. CPV can then translate a Btor2 witness back to a software violation witness. Currently, CPV outputs a dummy violation witness if a bug is reported by ABC. Since both the Btor2 and Aiger languages do not define a format for correctness witnesses, CPV also outputs a dummy correctness witness in this case. For the remaining tasks that cannot be translated into Aiger circuits, CPV uses a sequential portfolio of AVR's KI, PDR, and BMC.

Program Instrumentation for Witness Translation. To map the information in a Btor2 witness back to the original program, CPV instruments the input program prior to the program-to-circuit translation. A Btor2 violation witness encodes a computation trace that asserts the output of the translated circuit. The trace consists of a sequence of values given to the circuit's external inputs, each corresponding to a call to a function `__VERIFIER_nondet_X()` in the program.

[1] The Btor2-to-Aiger translation may fail if a Btor2 circuit uses data sorts or operations unsupported by Aiger, such as arrays or non-constant register initialization.

Table 1: Summary of CPV's correct results in SV-COMP 2024

ReachSafety verdict	#tasks	#solved	#tasks solved by respective approach AVR-KI	AVR-PDR	AVR-BMC	ABC-IMC
`true`	8 323	3 860	3 405	323	0	132
`false`	2 899	1 092	867	172	2	51

To assume these values at the control-flow locations where they are relevant for triggering the property violation, CPV's instrumentor assigns a fresh counter to each of these calls. A counter is incremented after each call, so its value can be inferred from the BTOR2 witness. An input value is relevant if accompanied by a change in its counter. The witness translator of CPV traverses the BTOR2 witness, extracts the relevant input values by tracking the changes in the counters, and exports the software violation witness in the GraphML format [13].

4 Results in SV-COMP 2024

CPV participated in the category *ReachSafety* of SV-COMP 2024 [5]. As a first-year participant, it surprisingly outperformed several mature software verifiers in terms of the number of correctly solved tasks. CPV is especially effective in the subcategory *ReachSafety-Hardware* and *ReachSafety-ECA*, solving the second and third most tasks among all participants, respectively. Its impressive results manifest the feasibility of using sequential circuits as an alternative intermediate representation to construct program verifiers.

The overall results of CPV is summarized in Table 1. Among the 11 222 verification tasks in the category *ReachSafety*, 8 439 were successfully translated to BTOR2 circuits by KRATOS2, and 7 773 could be further translated to AIGER circuits by BTOR2AIGER. In total, CPV produced 4 952 correct and confirmed results. The k-induction implementation in AVR contributed the most correctly solved and confirmed tasks, followed by PDR of AVR and IMC of ABC.[2]

We will improve CPV in the following directions: First, we will generate non-trivial software correctness witnesses through extracting and translating the fixed points computed by hardware model checkers. We aim to enhance the witness-confirmation rate of CPV, currently about 90 %, to the level of other mature participants (more than 95 %). Second, we will investigate the 27 false alarms in the subcategory *ReachSafety-Hardness.*

5 Setup and Configuration

We submitted CPV at version 0.4 [23] to SV-COMP 2024 [5]. A Linux-based operating system is required to execute the tool, as the used library COVERITEAM [12] relies on Linux-specific features, such as control groups, name spaces, and overlay file systems. Additional Python package requirement and the instructions to set up the execution environment can be found in the `README` file of the submitted tool archive.

[2] The observations are specific to the order of algorithms in CPV's sequential portfolios.

Data-Availability Statement. CPV is an open-source project, developed and maintained by the Software and Computational Systems Lab at LMU Munich. Its source code and executables are archived on Zenodo [23], and the project is maintained on GitLab at https://gitlab.com/sosy-lab/software/cpv.

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

References

1. Mukherjee, R., Tautschnig, M., Kroening, D.: v2c: A Verilog to C translator. In: Proc. TACAS. pp. 580–586. LNCS 9636, Springer (2016). https://doi.org/10.1007/978-3-662-49674-9_38
2. Beyer, D., Chien, P.C., Lee, N.Z.: Bridging hardware and software analysis with Btor2C: A word-level-circuit-to-C translator. In: Proc. TACAS. pp. 1–21. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_12
3. Noureddine, M.A., Zaraket, F.A.: Model checking software with first order logic specifications using AIG solvers. IEEE Trans. Softw. Eng. **42**(8), 741–763 (2016). https://doi.org/10.1109/TSE.2016.2520468
4. Long, J.: Reasoning about High-Level Constructs in Hardware/Software Formal Verification. Ph.D. thesis, University of California, Berkeley (2017). http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-150.html
5. Beyer, D.: State of the art in software verification and witness validation: SV-COMP 2024. In: Proc. TACAS. LNCS , Springer (2024)
6. Griggio, A., Jonáš, M.: Kratos2: An SMT-based model checker for imperative programs. In: Proc. CAV. pp. 423–436. Springer (2023). https://doi.org/10.1007/978-3-031-37709-9_20
7. Niemetz, A., Preiner, M., Wolf, C., Biere, A.: Btor2, BtorMC, and Boolector 3.0. In: Proc. CAV. pp. 587–595. LNCS 10981, Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_32
8. Niemetz, A., Preiner, M., Wolf, C., Biere, A.: Source-code repository of Btor2, BtorMC, and Boolector 3.0. https://github.com/Boolector/btor2tools, accessed: 2023-01-29
9. Biere, A.: The AIGER And-Inverter Graph (AIG) format version 20071012. Tech. Rep. 07/1, Institute for Formal Models and Verification, Johannes Kepler University (2007). https://doi.org/10.35011/fmvtr.2007-1
10. Goel, A., Sakallah, K.: AVR: Abstractly verifying reachability. In: Proc. TACAS. pp. 413–422. LNCS 12078, Springer (2020). https://doi.org/10.1007/978-3-030-45190-5_23
11. Brayton, R., Mishchenko, A.: ABC: An academic industrial-strength verification tool. In: Proc. CAV. pp. 24–40. LNCS 6174, Springer (2010). https://doi.org/10.1007/978-3-642-14295-6_5
12. Beyer, D., Kanav, S.: CoVeriTeam: On-demand composition of cooperative verification systems. In: Proc. TACAS. pp. 561–579. LNCS 13243, Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_31
13. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Lemberger, T., Tautschnig, M.: Verification witnesses. ACM Trans. Softw. Eng. Methodol. **31**(4), 57:1–57:69 (2022). https://doi.org/10.1145/3477579
14. Biere, A., van Dijk, T., Heljanko, K.: Hardware model checking competition 2017. In: Proc. FMCAD. p. 9. IEEE (2017). https://doi.org/10.23919/FMCAD.2017.8102233

15. Biere, A., Froleyks, N., Preiner, M.: 11th Hardware Model Checking Competition (HWMCC 2020). http://fmv.jku.at/hwmcc20/, accessed: 2023-01-29
16. Beyer, D., Wehrheim, H.: Verification artifacts in cooperative verification: Survey and unifying component framework. In: Proc. ISoLA (1). pp. 143–167. LNCS 12476, Springer (2020). https://doi.org/10.1007/978-3-030-61362-4_8
17. Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M.E., Sebastiani, R.: Software model checking via large-block encoding. In: Proc. FMCAD. pp. 25–32. IEEE (2009). https://doi.org/10.1109/FMCAD.2009.5351147
18. Bradley, A.R.: SAT-based model checking without unrolling. In: Proc. VMCAI. pp. 70–87. LNCS 6538, Springer (2011). https://doi.org/10.1007/978-3-642-18275-4_7
19. Eén, N., Mishchenko, A., Brayton, R.K.: Efficient implementation of property directed reachability. In: Proc. FMCAD. pp. 125–134. FMCAD Inc. (2011). https://dl.acm.org/doi/10.5555/2157654.2157675
20. McMillan, K.L.: Interpolation and SAT-based model checking. In: Proc. CAV. pp. 1–13. LNCS 2725, Springer (2003). https://doi.org/10.1007/978-3-540-45069-6_1
21. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: Proc. FMCAD, pp. 127–144. LNCS 1954, Springer (2000). https://doi.org/10.1007/3-540-40922-X_8
22. 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
23. Chien, P.C., Lee, N.Z.: CPV: A circuit-based program verifier. Zenodo (2023). https://doi.org/10.5281/zenodo.10203472, version 0.4