










PIChecker: A POR and Interpolation based Verifier for Concurrent Programs (Competition Contribution)*

Jie Su , Zuchao Yang , Hengrui Xing , Jiyu Yang ,
Cong Tian  **, and Zhenhua Duan 

ICTT and ISN Lab, Xidian University, Xi'an 710071, China
{jsu_3,mujueke,morui,jiyuy2024}@stu.xidian.edu.cn,
{ctian,zhhduan}@mail.xidian.edu.cn

Abstract. PIChecker is a tool for verifying reachability properties of concurrent C programs. It moderates the trace-space explosion problem, aggravated by thread alternation, through utilizing the PC-DPOR and C-Intp techniques. The PC-DPOR technique constructs a constrained dependency graph to refine dependencies between transitions. With this basis, the inherent imprecision of the dependence over-approximation can be overcome. Thereby, many redundant equivalent traces are prevented from being explored. On the other hand, the C-Intp technique performs conditional interpolation to confine the reachable regions of states, so that infeasible conditional branches which occur more frequently in concurrent verification tasks could be pruned automatically. We have implemented the above techniques on top of the open-source program analysis framework CPAchecker.

Keywords: Partial-Order Reduction · Interpolation · Concurrent Program · Model Checking

1 Verification Approach

Program synthesis[11] and verification[5] are two ways to improve the quality of software. In this paper, we propose a tool, namely PIChecker, that utilizes the PC-DPOR [9] and C-Intp [8] techniques to verify the reachability properties of concurrent programs. These techniques work in two different ways, equivalent trace class partitioning and infeasible conditional branch pruning, to reduce the search space in model checking.

The PC-DPOR technique addresses the problem that the coarse dependency approximation of transitions used in many POR [6] approaches significantly increases the number of equivalent trace classes to be explored. In order to reduce

* This research is supported by the National Natural Science Foundation of China (No. 62192734, No. 61732013 and No. 62172322).

** The corresponding author.

unnecessary exploration, the PC-DPOR technique constructs a *constrained dependency graph* (CDG) to refine the dependencies between transitions, where the edges in a CDG represent the dependency constraints that transitions from different threads depend on each other. The first configuration in Fig. 1 combines this technique with BDD-based reachability analysis to explore the reachable state-space of a concurrent program. At each state s , if there are *isolated transitions* which have no connection with the nodes of other threads in the CDG, then only one reachable successor state s' corresponding to an isolated transition will be explored (i.e., the enabled transitions of other threads will be pruned). We have proved that the prioritized exploration strategy for isolated transition still provides full coverage of all program behaviors[9]. This prioritized exploration continues until a *checking state* without any successor of isolated transition is reached. Thereafter, the dependency between any two different transitions t and t' at a checking state can be dynamically determined by checking whether their dependency constraint holds at the checking state. If the constraint does not hold (i.e., t is independent of t' at the current checking state), then only one of the execution orders $t \cdot t'$ and $t' \cdot t$ will be explored. With the basis of CDG, the inherent imprecision of traditional dependence over-approximation is overcome and many redundant equivalent traces can be saved from being explored.

On the other hand, the C-Intp technique focuses on pruning the infeasible conditional branches that may be explored in traditional abstraction-refinement iterations [7] when predicates are insufficient. At each state s , besides the reachability check of error locations, the C-Intp technique also inspects whether there exists any path that contains infeasible conditional branches. If so, the C-Intp technique will treat such a path as another form of spurious path, and additional constraints, namely *conditional interpolants*, will be generated by performing conditional interpolation on these additional spurious paths. Thereafter, infeasible conditional branches can be pruned by introducing these constraints into the reachable regions of states. In order to improve the efficiency of satisfiability checking and Craig interpolation [4] steps performed by C-Intp, the generated conditional interpolants are utilized to shorten the interpolation paths. To do so, *the shortest C-Intp formula chains* which contain only the formulas that affect decision-making are constructed at each choice point to perform the interpolations. With the conditional interpolants and shorter interpolation paths, a sufficient amount of predicates can be generated efficiently, and more attention can be paid to the analysis of feasible paths.

2 Software Architecture

PIChecker is developed on top of CPAchecker with the PC-DPOR and C-Intp extensions. By taking the strength of the CPA concept, PIChecker uses different configurations as shown in Fig. 1 to cover as many concurrent programs as possible. Within the verification time-bound, the verification for a given program starts by executing the first configuration that combines the PC-DPOR technique and BDD-based reachability analysis. If a counterexample is reported,

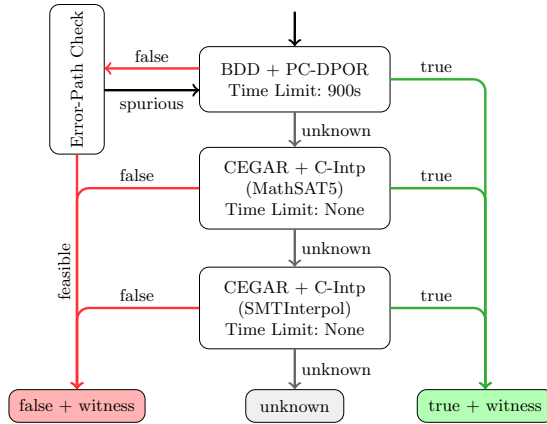


Fig. 1. The verification flow that combines the PC-DPOR and C-Intp strategies.

the feasibility of this error path will be checked since the BDD-based reachability analysis in `CPAchecker` currently only supports the representation of integer variable values and other states in waitlist will continue to be explored if the counterexample is spurious. If the execution of the first configuration terminates unexpectedly within 900s, the verification will continue by using the other two CEGAR + C-Intp based configurations with different back-end solvers. In that case, the second configuration with the `MathSAT5` will be chosen firstly. If its execution also aborts abnormally because the `MathSAT5` solver fails to perform interpolation on the shortest C-Intp formula chains generated by the C-Intp approach, the last configuration with the `SMTInterpol` solver will finally be utilized if the time cost is still within the bound.

3 Strengths and Weakness

Compared to `CPAchecker` which conservatively approximates the independence of transitions by checking whether a transition only accesses local variables [2], the use of CDG in `PIChecker` can improve the precision of estimating the dependencies of enabled transitions at reachable states. Therefore, the exploration of more traces in the same equivalent class can be avoided by utilizing `PIChecker`. In addition, different from most of the abstraction-refinement approaches that generate only a few number of predicates at the end of each iteration, the two CEGAR + C-Intp based configurations can effectively generate a sufficient amount of conditional interpolants within a single round of iteration by performing the conditional interpolation technique at conditional branches. Thus, the exploration of many infeasible conditional branches can be avoided. For the sake of clarifying the improvement from `PIChecker` more clearly, a comparison between `PIChecker` and `CPAchecker`, on checking the `unreach-call` property under the category `ConcurrencySafety` in `SV-COMP 2023`, is made. The results indicate

that `PIChecker` succeeds to verify 394 out of 665 verification tasks, which is more than 375 of `CPAchecker`. Further, for the 372 tasks that can be verified by the both tools, the average time and memory costs of `PIChecker` (37.49s, 672.15MB) only account for 56.58% and 61.71% of the corresponding overheads consumed by `CPAchecker` (66.27s, 1089.19MB), respectively.

In order to guarantee the correctness of verification results, some conservative strategies are adopted by the three configurations. For example, when the program statement corresponding to a transition contains non-deterministic function calls (e.g., `'x = __VERIFIER_nondet_int();'`), the PC-DPOR technique conservatively considers it to be dependent on other transitions if they access the same shared variables. These strategies may significantly reduce the verification efficiency.

4 Tool Setup and Configuration

`PIChecker` is built on the `CPAchecker` codebase and is publicly available¹. It contains all the dependent libraries and requires a `Java 11 Runtime Environment`. In `SV-COMP 2023`, `PIChecker` only participates in the `ConcurrencySafety` category and checks the `unreach-call` property². Before verifying a program, all files from the submitted archive must be extracted into the same folder. Executing `PIChecker` on a task can be done in the same way as executing any other `CPAchecker` configuration by running: `scripts/cpa.sh -svcomp23-pichecker -timelimit <TIME_LIMIT> [-spec <SPEC_FILE>] <SOURCE_FILE>`. The experimental statistics and verification results are written in `output/Statistics.txt`. Moreover, human readable counterexamples `output/Counterexample.%d.txt` will be generated if the reachability property does not hold. For more instructions, please refer to `README.md` and `INSTALL.md`.

5 Software Project and Contributors

Based on the open-source tool `CPAchecker` [3], `PIChecker` has been developed by Jie Su, Zuchao Yang, Hengrui Xing, Jiyu Yang from the ICTT Lab in Xidian University under the supervision of Cong Tian and Zhenhua Duan. We thank Dirk Beyer and his team for their original contributions to `CPAchecker`. `PIChecker` is licensed under the Apache 2.0, and it also contains the copyright of `CPAchecker`.

Data Availability Statement. All data of `SV-COMP 2023` are archived as described in the competition report[1] and available on the [competition web site](#). This includes the verification tasks, results, witnesses, scripts, and instructions for reproduction. The version of `PIChecker` used in the competition is archived on Zenodo [10] and also in its own artifact at [GitLab](#).

¹ `PIChecker` repository: <https://gitlab.com/Lapulatos/pichecker.git>

² The benchmark definition of `PIChecker`: <https://gitlab.com/sosy-lab/sv-comp/bench-defs/-/blob/main/benchmark-defs/pichecker.xml>

References

1. Beyer, D.: Competition on software verification and witness validation: SV-COMP 2023. In: Proc. TACAS (2). LNCS , Springer (2023)
2. Beyer, D., Friedberger, K.: A Light-Weight Approach for Verifying Multi-Threaded Programs with CPAChecker. arXiv preprint arXiv:1612.04983 (2016). <https://doi.org/10.4204/EPTCS.233.6>
3. Beyer, D., Keremoglu, M.E.: CPACHECKER: A Tool for Configurable Software Verification. In: Proceedings of the 23rd International Conference on Computer Aided Verification. pp. 184–190. CAV’11, Springer Berlin Heidelberg, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_16
4. Craig, W.: Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *The Journal of Symbolic Logic* **22**(3), 269–285 (1957)
5. Fetzer, J.H.: Program verification: The very idea. *Communications of the ACM* **31**(9), 1048–1063 (1988)
6. Godefroid, P.: Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem. Springer (1996)
7. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 58–70. POPL’02, Association for Computing Machinery, New York, NY, USA (2002). <https://doi.org/10.1145/503272.503279>
8. Jie, S., Cong, T., Zhenhua, D.: Conditional Interpolation: Making Concurrent Program Verification More Effective. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 144–154. ESEC/FSE’21, Association for Computing Machinery, New York, NY, USA (2021). <https://doi.org/10.1145/3468264.3468602>
9. Jie, S., Cong, T., Zuchao, Y., Jiyu, Y., Bin, Y., Zhenhua, D.: Prioritized Constraint-Aided Dynamic Partial-Order Reduction. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering. ASE’22, Association for Computing Machinery, New York, NY, USA (2022). <https://doi.org/10.1145/3551349.3561159>
10. Jie, S., Zuchao, Y., Hengrui, X., Jiyu, Y., Cong, T., Zhenhua, D.: PIChecker for SV-COMP 2023 (Dec 2022). <https://doi.org/10.5281/zenodo.7471378>
11. Mengfei, Y., Bin, G., Zhenhua, D., Zhi, J., Naijun, Z., Yunwei, D.: Intelligent program synthesis framework and key scientific problems for embedded software. *Chinese Space Science and Technology* **42**(4), 1 (2022)

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.

