Fault Localization on Verification Witnesses (Poster Paper)

Dirk Beyer
LMU Munich
Germany

Matthias Kettl
LMU Munich
Germany

Thomas Lemberger
LMU Munich
Germany

ABSTRACT

Verifiers export violation witnesses, which help independent validators to confirm a reported specification violation. It is assumed that violation witnesses are helpful if they are very precise: ideally, they should describe a single program path for the validator to check. But we claim that this leads verifiers to produce large, detailed witnesses that include a lot of unnecessary information that actually hinders validation. We reduce violation witnesses with automated fault localization to only that information which fault localization suspects as fault. We performed a large experimental evaluation on the witnesses produced in the International Competition on Software Verification (SV-COMP 2023) to explore the effect of our reduction. Our experiments show that the witnesses reduced using our approach shrink considerably and can be confirmed better.

CCS CONCEPTS

• **Software and its engineering** → **Software verification**; *Model checking*; **Formal software verification**; *Software defect analysis.*

KEYWORDS

Software Verification, Program Analysis, Model Checking, Result Validation, Fault Localization, Violation Witnesses, Error Paths

ACM Reference Format:
Dirk Beyer, Matthias Kettl, and Thomas Lemberger. 2024. Fault Localization on Verification Witnesses (Poster Paper). In *Companion Proceedings of the 46th IEEE/ACM International Conference on Software Engineering (ICSE '24 Companion), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3639478.3643099

1 INTRODUCTION

When a formal software verifier reports an alarm, it produces a *violation witness* [6] to increase the confidence in its result. A violation witness describes a subset of program paths of which at least one contains the reported specification violation. This helps to reason about the alarm and allows to validate the alarm. All participants of the International Competition on Software Verification (SV-COMP) [3] produce such violation witnesses.

Contrary to a test case, violation witnesses are hardly readable for humans and can grow to thousands of lines. Additionally, we observe that witnesses contain unnecessary states and assumptions. We solve these problems through fault localization: Given a violation witness, we reconstruct a potential error path, apply fault localization to get suspects for faults along the path, and delete all edges that are deemed irrelevant from the witness. Figure 1 illustrates the workflow of our approach.

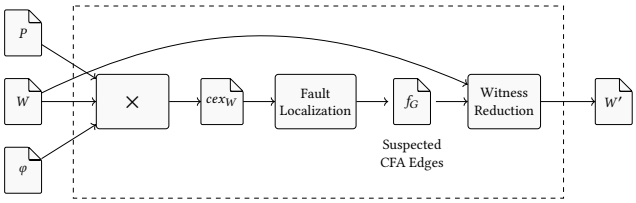

Figure 1: Workflow of fault localization on violation witnesses; after creating the product automaton of specification ϕ, witness W, and program P, we apply fault localization on the obtained error path cex_W to find suspects f_G; these are used to reduce the violation witness W to W'

Related Work. Many fault-localization techniques exist [1, 10, 13, 14, 17]. Other approaches [5, 11, 15, 16] provide step-by-step simulators for error paths. We reduce the size of witnesses to relevant information beyond input values. This works on witnesses that miss input values, and makes simulators focus on relevant information.

2 FAULT LOCALIZATION ON VIOLATION WITNESSES

Program Representation. Our implementation and experiments work on GNU C programs. For presentation, we consider an imperative, sequential programming language with two types of operations: assign operation (`x = x + 1`) and assume operation (`[x <= 0]`). *Ops* is the set of all valid operations. We represent programs as control-flow automata (CFA). A CFA (L, l_0, G) consists of program locations L, initial location $l_0 \in L$ and edges $G = L \times Ops \times L$.

Fault Localization. A *suspect* is a set of program lines that, together, may be responsible for a program error. Given a feasible error path that ends in a program error, fault localization determines a finite set $\mathcal{F} = \{f_0, f_1, \ldots, f_n\}$ of suspects. All three different fault-localization techniques that we consider work on error paths with negated final assumption, yielding an infeasible error path: MaxSat [14] finds the maximum satisfiable subsets of the infeasible error path and returns the complement as suspects. MinUnsat computes minimal unsat cores as suspects. Unsat returns an arbitrary unsat core and serves as baseline in our experiments.

Violation Witness. A violation witness describes program executions of which at least one leads to a specification violation by restricting the set of all possible program executions through source-code guards and state-space guards: Source-code guards restrict the control flow, and state-space guards restrict the potential program states. A witness validator [6] checks whether a violation witness describes any program execution that reaches the claimed specification violation. If it does, the violation witness is *confirmed*. If it does not, the violation witness is *rejected*.

ICSE 2024, April 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0502-1/24/04
https://doi.org/10.1145/3639478.3643099

Table 1: Remaining transitions after reduction with r_{all}

	MaxSat	MinUnsat	Unsat
Mean	45.92 %	45.47 %	42.92 %
Median	66.62 %	66.71 %	40.81 %

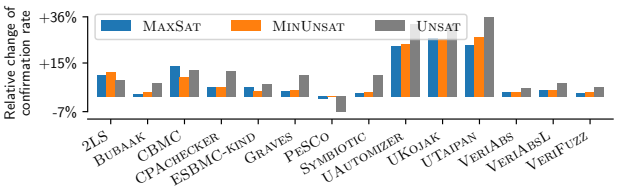

Figure 2: Change of confirmation rate of MetaVal with r_{all}

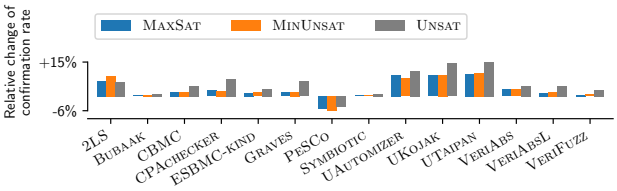

Figure 3: Change of confirm. rate of UAutomizer with r_{state}

Our Witness Reduction. Figure 1 illustrates the workflow of our approach. Given a program P, a violation witness W, and a specification φ, we build the product automaton, extract the error path cex_W, run fault localization on it, and produce a set of suspects. We then take the suspects and the original witness to obtain the reduced witness W'. A transition is irrelevant if it is not part of the suspects. We experiment with three reduction strategies: Strategy r_{all} deletes all irrelevant edges that contain no information about program branches. Strategy r_{state} deletes irrelevant state-space guards and keeps all source-code guards. Strategy r_{match} turns irrelevant state-space guards into trivial *true* state-space guards. The witnesses still steer the validation. We proved that all three strategies are sound.

3 EVALUATION

Experiment Setup. We conduct the experiments on machines with an Intel Xeon E3-1230 v5 @ 3.40 GHz 8-core processor and 33 GB memory, limited to 2 cores and 7 GB of memory. The timeout for validation is set to 90 s. This aligns with the SV-COMP 2023 setup.

We use all violation witnesses [4] that were produced by 14 non-hors-concours verifiers on 3 225 non-recursive unsafe verification tasks with property *unreach-call* of SV-COMP 2023. For witness reduction, we use CPAchecker in revision 44 191 and our tool Flow[1] in revision cc5e4f8d. For witness validation, we use four validators from SV-COMP 2023: CPAchecker [7], UAutomizer [12], MetaVal [9], and Symbiotic-Witch [2]. The presented data exclude witnesses for which the respective fault-localization approach did not work.

Results. With reduction r_{all}, we decrease the number of transitions in witnesses to about 45 % with MaxSat and MinUnsat, and to 43 % with Unsat—on average, across the three fault localization techniques (see Table 1). We exemplarily show data for the validators MetaVal and UAutomizer after applying r_{all} and r_{state}, respectively (Figs. 2 and 3). In both cases, we experience an increase in confirmed witnesses. Both plots show that the reduction increases the number of confirmed witnesses independently of the used fault-localization technique. This results in up to 36 % more confirmations through our reduction approach.

4 CONCLUSION

Applying fault localization to violation witnesses is an effective and sound way to increase the confirmation rate and to lower the number of unnecessary transitions in the witness automaton. Next to boosting the performance of validators, our approach makes witnesses easier to store and comprehend.

Data-Availability Statement. The experiment setup and all experimental data are archived and available at Zenodo [8].

Funding Statement. This project was funded in part by the Deutsche Forschungsgemeinschaft (DFG) – 378803395 (ConVeY), 418257054 (Coop), and 496588242 (IdeFix).

REFERENCES

[1] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. C. van Gemund. 2009. A practical evaluation of spectrum-based fault localization. *J. Syst. Softw.* 82, 11 (2009), 1780–1792. https://doi.org/10.1016/j.jss.2009.06.035

[2] P. Ayaziová and J. Strejček. 2023. Symbiotic-Witch 2: More Efficient Algorithm and Witness Refutation (Competition Contribution). In *Proc. TACAS (2) (LNCS 13994)*. Springer, 523–528. https://doi.org/10.1007/978-3-031-30820-8_30

[3] D. Beyer. 2023. Competition on Software Verification and Witness Validation: SV-COMP 2023. In *Proc. TACAS (2) (LNCS 13994)*. Springer, 495–522. https://doi.org/10.1007/978-3-031-30820-8_29

[4] D. Beyer. 2023. Verification Witnesses from Verification Tools (SV-COMP 2023). Zenodo. https://doi.org/10.5281/zenodo.7627791

[5] D. Beyer and M. Dangl. 2016. Verification-Aided Debugging: An Interactive Web-Service for Exploring Error Witnesses. In *Proc. CAV (2) (LNCS 9780)*. Springer, 502–509. https://doi.org/10.1007/978-3-319-41540-6_28

[6] D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, T. Lemberger, and M. Tautschnig. 2022. Verification Witnesses. *ACM Trans. Softw. Eng. Methodol.* 31, 4 (2022), 57:1–57:69. https://doi.org/10.1145/3477579

[7] D. Beyer and M. E. Keremoglu. 2011. CPAchecker: A Tool for Configurable Software Verification. In *Proc. CAV (LNCS 6806)*. Springer, 184–190. https://doi.org/10.1007/978-3-642-22110-1_16

[8] D. Beyer, M. Kettl, and T. Lemberger. 2023. Reproduction Package for Article 'Fault Localization on Witnesses'. https://doi.org/10.5281/zenodo.10554862

[9] D. Beyer and M. Spiessl. 2020. MetaVal: Witness Validation via Verification. In *Proc. CAV (LNCS 12225)*. Springer, 165–177. https://doi.org/10.1007/978-3-030-53291-8_10

[10] E. Ermis, M. Schäf, and T. Wies. 2012. Error Invariants. In *Proc. FM (LNCS 7436)*. Springer, 187–201. https://doi.org/10.1007/978-3-642-32759-9_17

[11] G. Ernst, J. Blau, and T. Murray. 2021. Deductive Verification via the Debug Adapter Protocol. In *Proc. F-IDE@NFM (EPTCS, Vol. 338)*. arXiv, 89–96. https://doi.org/10.4204/EPTCS.338.11

[12] M. Heizmann, J. Hoenicke, and A. Podelski. 2013. Software Model Checking for People Who Love Automata. In *Proc. CAV (LNCS 8044)*. Springer, 36–52. https://doi.org/10.1007/978-3-642-39799-8_2

[13] J. A. Jones and M. J. Harrold. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proc. ASE*. ACM, 273–282. https://doi.org/10.1145/1101908.1101949

[14] M. Jose and R. Majumdar. 2011. Cause Clue Clauses: Error Localization using Maximum Satisfiability. In *Proc. PLDI*. ACM, 437–446. https://doi.org/10.1145/1993498.1993550

[15] P. Müller and J. N. Ruskiewicz. 2011. Using Debuggers to Understand Failed Verification Attempts. In *Proc. FM (LNCS 6664)*. Springer, 73–87. https://doi.org/10.1007/978-3-642-21437-0_8

[16] P. Rockai and J. Barnat. 2022. DivSIM, an interactive simulator for LLVM bitcode. *STTT* 24, 3 (2022), 493–510. https://doi.org/10.1007/s10009-022-00659-x

[17] W. E. Wong, V. Debroy, R. Gao, and Y. Li. 2014. The DStar Method for Effective Software Fault Localization. *IEEE Trans. Reliab.* 63, 1 (2014), 290–308. https://doi.org/10.1109/TR.2013.2285319

[1] https://gitlab.com/sosy-lab/software/fault-localization-on-witnesses