Proc. CAV 2016, © Springer

Verification-Aided Debugging: An Interactive Web-Service for Exploring Error Witnesses

Dirk Beyer and Matthias Dangl

University of Passau, Germany

Abstract. Traditionally, a verification task is considered solved as soon as a property violation or a correctness proof is found. In practice, this is where the actual work starts: Is it just a false alarm? Is the error reproducible? Can the error report later be re-used for bug fixing or regression testing? The advent of *exchangeable witnesses* is a paradigm shift in verification, from simple answers *true* and *false* towards qualitatively more valuable information about the reason for the property violation. This paper explains a convenient web-based toolchain that can be used to answer the above questions. We consider as example application the verification of C programs. Our first component collects witnesses and stores them for later re-use; for example, if the bug is fixed, the witness can be tried once again and should now be rejected, or, if the bug was not scheduled for fixing, the database can later provide the witnesses in case an engineer wants to start fixing the bug. Our second component is a web service that takes as input a witness for the property violation and (re-)validates it, i.e., it re-plays the witness on the system in order to re-explore the state-space in question. The third component is a web service that continues from the second step by offering an interactive visualization that interconnects the error path, the system's sources, the values on the path (test vectors), and the reachability graph. We evaluated the feasibility of our approach on a large benchmark of verification tasks.

1 Introduction

The answer of a verification tool to a given verification task (consisting of a specification and a system) is either that the system satisfies the specification or that the system violates the specification (or the answer 'unknown' is returned) [9]. If a violation of the specification is detected, an error path through the system is reported that exhibits the problem, such that the user can understand the problem and fix the bug: counterexamples to verification have been described as invaluable to debugging complex systems and have been a common feature of model checkers for several decades [7]. In particular, the successful technique of counterexample-guided abstraction refinement (CEGAR) [8] is based on analyzing error paths through the system.

In the past few years, there was a strong focus in the community on using common exchange formats and reproducing errors described by previously computed counterexamples. ESBMC was extended to reproduce errors via instantiated

code [11], and CPAchecker was used to re-check previously computed error paths by interpreting them as automata that control the state-space search [6]. While these internal approaches to witness validation can reduce the amount of false alarms reported by a tool, they establish no additional trust in a report produced and validated by an untrusted verifier. The advantages of considering error witnesses as a valuable verification artifact were explained and supported by two completely different implementations of witness validators [4], namely CPAchecker and Automizer. Also, competitions in the community required exchangeable witnesses: the competition on termination uses a certification-problem format (CPF) [1] and the competition on software verification uses a machine-readable, exchangeable format for error witnesses [2]. Our toolchain is based on the common exchange format that was used in SV-COMP [2, 4], which allows specifying counterexample traces using control-flow paths and data values. Previous efforts towards helping users understand the counterexamples have lead to interactive trace visualizations [1, 5, 10], but the user was locked-in to a certain toolchain. The introduction of machine-readable error witnesses has opened up new possibilities for collecting, accumulating, and validating counterexample traces from different verifiers [4]. A wide range of software verifiers already supports a common exchange format, as shown by the competition on software verification [3], which has adopted error-witness validation already two years ago.

Error witnesses support traditional debugging very well: the test values that a witness might contain can direct a classic debugger through the system to the problematic part of the implementation or model. But the exchangeable witnesses support even a more abstract form of debugging, based on a graphical visualization of error paths and reachability graphs.

Figure 1 gives an overview over the components involved in our toolchain. There are three subsystems that the user interacts with: (1) We developed a *witness store* for persistently keeping error witnesses that different verification tools have produced. The database enables the user to select and retrieve specific witnesses for a given set of verification tasks. One possible use case is to fetch all witnesses that document a bug in a specific C program, to help the developer better understand the issue. (2) We offer an *online witness validator* with a convenient web-service API that enables validation without the need to install software. A bug report that a verifier returns can potentially be a false alarm, so it is convenient for the user to first automatically cross-examine the report, before manual effort is invested (and perhaps wasted). To validate an error witness, the user can send the validation task, which consists of the source-code file, the property, and a corresponding error witness (potentially obtained from the witness database), to the validation service. The service then validates the error witness. If the witness is rejected, the user is advised to prioritize other tasks,

[1] `http://cl-informatik.uibk.ac.at/software/cpf`

[2] `http://sv-comp.sosy-lab.org/2016/witnesses/`

[3] For example, see the list of systems in SV-COMP 2016: `http://sv-comp.sosy-lab.org/2016/systems.php`.

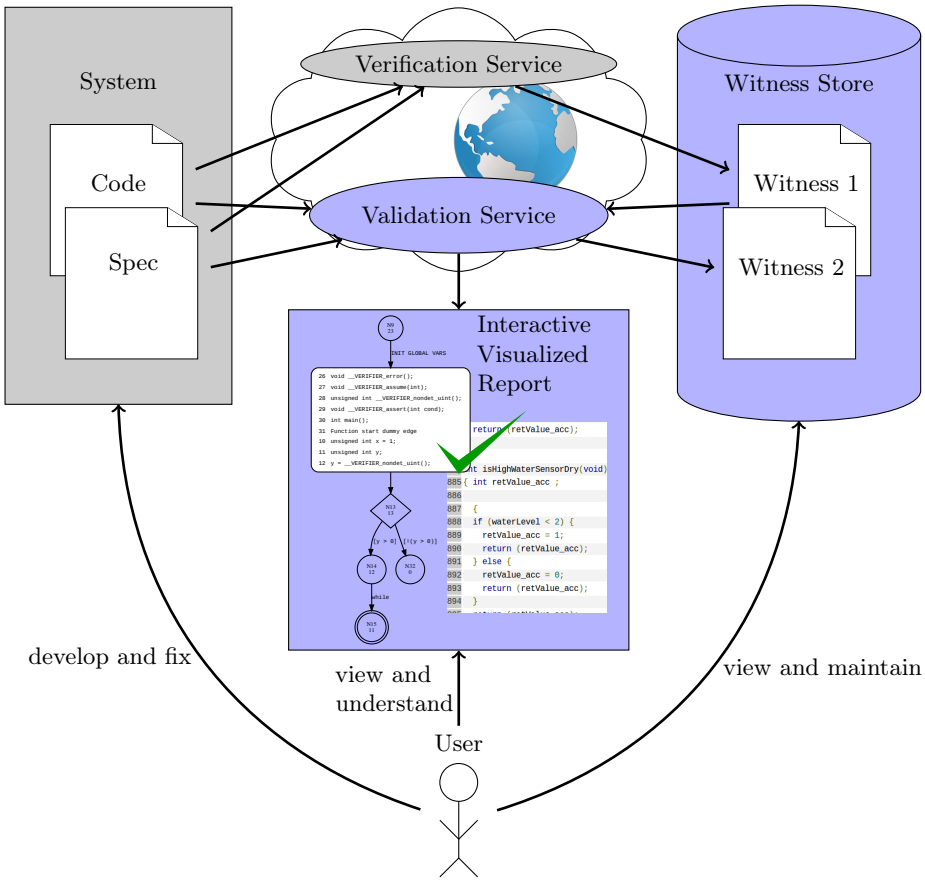

Fig. 1: System overview, blue parts are discussed in this paper

because the specific error path that the witness describes has been declared as infeasible. If, instead, the witness is validated, the validation service feeds all information gained about the bug into the third component, the interactive report. (3) Successful witness validations produce a detailed and interactive web-based bug report. The report contains a debugger-like feature for stepping through the error path, while providing several context-sensitive representations of the buggy program. The report also encompasses all information required to reproduce the validation externally.

Application Example. Our application example is the verification of system programs written in the language C. While the concepts of our toolchain can be applied to other programming languages, we restrict our tools to C. The web service that we describe is available on the internet, and our primary target is to support open-source projects. Organizations that develop proprietary software can still benefit from our system, because it is easily installed on a local web server that is restricted to the organization's intranet.

Data to Experiment. As part of our evaluation, we ran several verification tools that participated in the competition on software verification (because those tools are known to generate useful witnesses) and fed the witnesses into our database. For the reader to assess our toolchain, we have compiled an archive with witnesses, validation results and error-path visualizations for offline use. The archive is available as supplement, and the validation results and visualization results can be reproduced via our live web service or offline using the CPAchecker-based witness validator [4] The archive contains reports for a total of 1 382 witnesses for 26 verification tasks that contain a bug. The average number of witnesses that we collected is 53 witnesses per verification task, the program with the fewest has 4, the program with the most has 114 witnesses in our database.

2 Collection of Error-Paths in a Witness Store

We consider witnesses as a prime-value verification artifact, because they can make it (a) efficient to re-run a partial verification to explore the bug again and (b) easy to use different verification tools for validation.

Permanently storing witnesses opens many new practical applications to let verification technology have a larger impact on system development. Our witness store provides a means to take advantage of the various beneficial properties of machine-readable witnesses in a common exchange format:

- Witness Validation: Imprecise verifiers may sometimes produce false alarms and thus waste valuable developer time. With witness validation, users no longer need to trust the answer False. Instead, they can concentrate on paying attention to witnesses that are confirmed by an automatic witness validator. Each validation run that confirms a witness can increase the user's confidence in the bug report.
- Witness Inspection: Witness validators with complementing strategies can be applied to a witness, each leveraging its strengths to add diagnostic information that the others may be incapable to derive. Therefore, witness validation can be understood as a chain of ever refining details for identifying, understanding, and fixing the bug.
- Bug Reports: In bug reports, attached witnesses can be used to provide a precise description of the erroneous behavior, including test-vector values.
- Re-Verification: Working with error witnesses is cheap in terms of resources, because the verification result can often be re-established with reduced effort. This is not only beneficial for validating a given witness, but also when checking for regressions: If the witness is still valid for a changed version of the system, the bug has been reintroduced or was not yet fixed [6].

[4] The URL to our supplementary web page, which includes the live web service, the archive for offline use, and a virtual machine set up for validating the witnesses and reproducing the results using CPAchecker 1.6, is: `https://www.sosy-lab.org/~dbeyer/witness-based-debugging/`.

3 Convenient Witness Validation

A witness validator is a verifier that analyzes the synchronized product of the system with the witness automaton, where transitions are synchronized using system operations and transition annotations. This means that the witness automaton observes the system paths that the verifier wants to explore: if the operation on the system path does not match the transition of the witness automaton, then the verifier is forbidden to explore that path further; if the operation on the path matches, then the witness automaton and the system proceed to the next state, possibly restricting the system's state such that the assumptions given in the data annotation are satisfied. Implementations of witness validators are available, see for example CPAchecker and Automizer [4]. Our validation service uses the CPAchecker witness validator as back-end. CPAchecker supports and combines many different verification strategies, for example value analysis, predicate abstraction, CEGAR, bounded model checking, k-induction, and concrete memory graphs. The specific configuration that is effectively used to validate the witnesses via our web service is bit-accurate and combines value analysis and predicate abstraction. Our web service does not yet support arrays, concurrency, and termination analysis.

Conceptually, an *error-witness automaton* is a protocol automaton, and an *error-witness analysis* is a protocol analysis for an error-witness automaton [4], which runs as a component of a composite program analysis. Unlike observer automata [3], which can be used to represent the specification the analyzed program is verified with, error-witness automata not only observe the state-space exploration of the program analysis, but also *restrict* it to those successor states that lead the exploration toward a specification violation, whereas an observer automaton follows all abstract successor states. Therefore, the program analysis is *guided* by the error-witness automaton to explore the state space that violates the specification.

The process of determining if it is possible to independently re-establish a verification result, given the program, specification, result, and witness, is called *witness validation*. One way of implementing error-witness validation is by constructing a composite program analysis that has both a witness analysis and a specification analysis as components, which simultaneously restrict and observe the state-space exploration: the specification analysis checks if an analyzed path actually violates the specification, and the search of the composite program analysis is restricted by the witness validation such that only paths that the error-witness automaton can match are explored. For example, the analysis stops exploring a path, if, during the analysis of that path, the witness automaton takes a transition to a sink state. An error witness is confirmed by the witness validator if both, the witness automaton and the specification automaton, take a transition to their respective (accepting) error state [4].

4 Visualizing and Interactively Exploring Error-Paths

Figure 2 shows a screenshot of an interactive counterexample report. The screen is divided into two columns: The left column provides detailed information that is specific to the error path, namely the source code on the path to the property

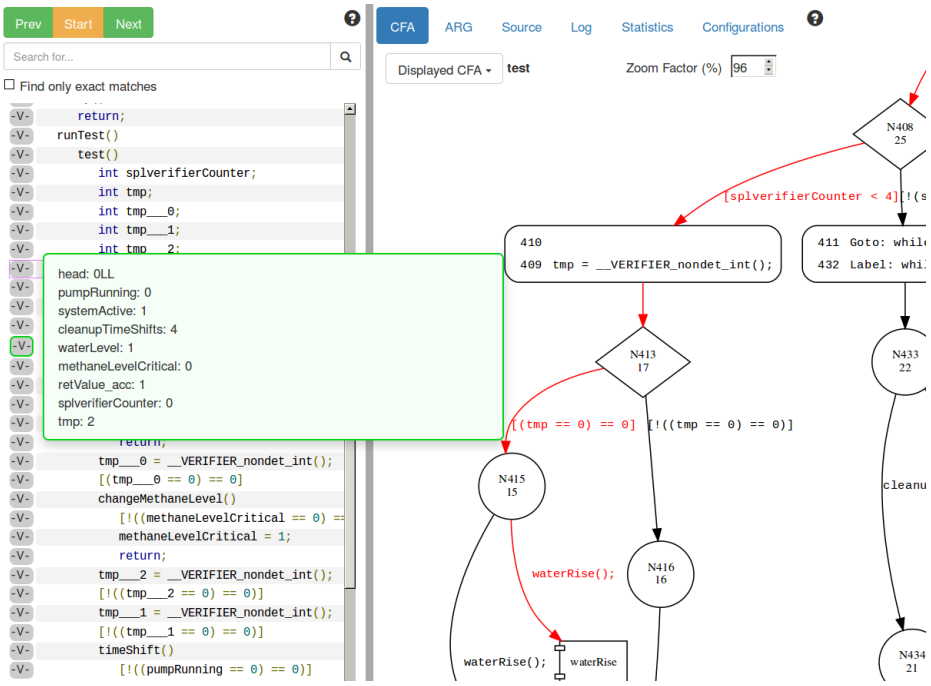

Fig. 2: Typical view of the error-path visualizer: program source code with violating test vector (left, green) and CFA with violating path (right, red); left view top shows the menu for debugger-like step-through, right view top shows the display options: CFA, ARG, Source, Log, Statistics, Configurations

violation, and, like in a debugger, the program locations are decorated with test values that were computed by the witness validator. The right column embeds the specific information from the left column into the general context of the system and the analysis. It contains control-flow automata (CFA) for each of the functions, the abstract reachability graph (ARG) of the verification, full source code of the verification task, the verification log, statistics, and configuration parameters of the validation run. In all CFA and the ARG, the states on the path to the property violation are marked in red. Double clicking on a control-flow state that precedes a function call displays the CFA of the called function. Both columns, however, are not only useful in isolation: clicking on a line of code in the left column while viewing the ARG or CFA will navigate to the state corresponding to the clicked line of source code.

The visualization is built upon the JavaScript framework AngluarJS and the jQuery and Bootstrap web-development libraries. The layout of the graphs is computed using GraphViz and exchanged in SVG format. The complete data for one such error-path visualization takes on average 120 kB of memory.

5 Conclusion

Over the past decades, the algorithmic abilities of verification tools were considerably increased, but in practice, verification technology is still not as popular as testing. Why? Because because it is inconvenient to use. Our work contributes to closing this gap, by considering not only the true/false answers as value, but actively using other results of the verification process, most prominently the error witnesses. We have presented a toolchain that supports engineers in understanding the error reports of verification systems. First, we archive verification witnesses permanently in a database. Second, we provide a convenient web service for witness validation, i.e., a verification task together with a witness can be given as input, and the results are presented via the web API (for manual inspection or automatic retrieval). Third, we explain an error-path visualization that supports an interactive investigation of the source code, the control-flow graph, the reachability graph, and test values. We believe that the proposed method is a step towards a more convenient usage of verification results.

References

1. H. Aljazzar and S. Leue. Debugging of dependability models using interactive visualization of counterexamples. In *Proc. QEST'08*, pages 189–198. IEEE, 2008.
2. D. Beyer. Reliable and reproducible competition results with BenchExec and witnesses. In *Proc. TACAS*, LNCS 9636, pages 887–904. Springer, 2016.
3. D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. The Blast query language for software verification. In *Proc. SAS*, LNCS 3148, pages 2–18. Springer, 2004.
4. D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, and A. Stahlbauer. Witness validation and stepwise testification across software verifiers. In *Proc. FSE*, pages 721–733. ACM, 2015.
5. D. Beyer and M. E. Keremoglu. CPAchecker: A tool for configurable software verification. In *Proc. CAV*, LNCS 6806, pages 184–190. Springer, 2011.
6. D. Beyer and P. Wendler. Reuse of verification results: Conditional model checking, precision reuse, and verification witnesses. In *Proc. SPIN*, LNCS 7976, pages 1–17. Springer, 2013.
7. E. M. Clarke, E. A. Emerson, and J. Sifakis. Model checking: Algorithmic verification and debugging. *Commun. ACM*, 52(11):74–84, 2009.
8. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.
9. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT, 1999.
10. A. Groce, D. Kröning, and F. Lerda. Understanding counterexamples with explain. In *Proc. CAV'04*, pages 453–456. Springer, 2004.
11. H. Rocha, R. S. Barreto, L. Cordeiro, and A. D. Neto. Understanding programming bugs in ANSI-C software using bounded model checking counter-examples. In *Proc. IFM*, LNCS 7321, pages 128–142. Springer, 2012.