Verification Witnesses *

Dirk Beyer [1], Matthias Dangl [1], Daniel Dietsch [2],
Matthias Heizmann [2], and Andreas Stahlbauer [1]

[1] University of Passau, Germany [2] University of Freiburg, Germany

http://www.sosy-lab.org/~dbeyer/verification-witnesses/

Abstract: It is commonly understood that a verification tool should provide a counterex-ample to witness a specification violation. Until recently, software verifiers dumped error witnesses in proprietary formats, which are often neither human- nor machine-readable, and an exchange of witnesses between different verifiers was impossible. We have defined an *exchange format for error witnesses* that is easy to write and read by verification tools (for further processing, e.g., witness validation). To eliminate manual inspection of false alarms, we develop the notion of *stepwise testification*: in a first step, a verifier finds a problematic program path and, in addition to the verification result FALSE, constructs a witness for this path; in the next step, another verifier re-verifies that the witness indeed violates the specification. This process can have more than two steps, each reducing the state space around the error path, making it easier to validate the witness in a later step. An obvious application for testification is the setting where we have two verifiers: one that is efficient but imprecise and another one that is precise but expensive. The technique of error-witness-driven program analysis is implemented in two state-of-the-art verification tools, CPACHECKER and ULTIMATE AUTOMIZER.

Overview

Software verification becomes more and more important in practice; several breakthroughs in verification research were achieved during the last decade, and several successful verification tools were developed. The TACAS International Competition on Software Verification (SV-COMP) [1] [Bey14, Bey15] serves as a showcase of the state-of-the-art. Users can choose from a wide range of verifiers, and the SV-COMP categories give an approximate guidance on which verifier is good for which kind of programs. One important and unsolved problem of applying verification technology in practice is that verification tools sometimes produce false alarms, and it still requires an enormous manual effort to find out if a reported bug indeed represents a genuine specification violation.

Our solution comprises two components: we developed an *exchange format for error witnesses* and evaluated its effectiveness by a thorough experimental evaluation, and we develop the notion of *stepwise testification*, as the technique of witness validation immediately leads to the notion of witness refinement, enabling a chain of verifiers (or testifiers) to continuously refine the erroneous state space until a test vector for the error is found.

---

* This is a summary of a full article on this topic that appeared in Proc. FSE 2015 [BDD+15].

[1] http://sv-comp.sosy-lab.org/

Testification is the process of giving evidence for a claim that a given program satisfies, or violates, its specification. The evidence of the absence, or presence, of a specification violation is given by one or more witnesses. A verification tool is a *testifier* if it provides evidence to support its claim, i.e., if it produces a witness for correctness or for a violation of the specification. *Stepwise testification* is the process of applying testification in several steps, on ever refined witnesses, possibly using different verification tools, combining different strengths. Figure 1 illustrates the process of stepwise testification. Our study explores stepwise testification of specification violations by producing error witnesses (left part), while conditional model checking [BHKW12] focuses on stepwise testification of correctness.

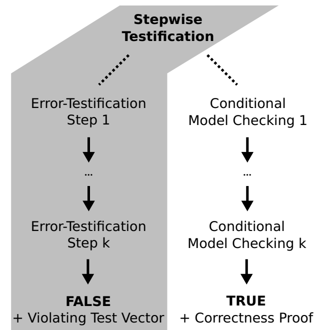

Figure 1: Stepwise testification: conceptual view

We accompany the bug report of verifier V_1 with an error witness, which represents information that can effectively guide another verifier V_2 to efficiently re-explore the state space that verifier V_1 reported to contain a bug. Our experimental study [BDD$^+$15] confirms the following insights: (1) our exchange format makes it possible to communicate error witnesses across verifiers, (2) verifier V_2 needs on average considerably less resources to validate the witness than verifier V_1 needed to find the error, even if V_2 uses a more expensive verification technology (e.g., V_1 using linear and V_2 using bit-precise arithmetic), (3) stepwise testification can be more efficient than verification, i.e., the CPU time for V_1-verification + V_2-witness-validation can be less than the CPU time for V_2-verification alone, (4) the state-space to be analyzed by V_2 is effectively reduced.

On the syntactic level, we use XML, more specifically GraphML, as a language to represent error witnesses. On the semantic level, we use the standard concept of (non-deterministic) finite automata to represent an error witness. A witness automaton observes the paths that the verifier explores and directs the exploration engine along the paths that the witness describes, i.e., towards the violation of the specification. Witnesses can be read by humans or a witness validator.

Our technique was already used in the two most recent editions of the competition on software verification. The SV-COMP community manifested in the competition rules that each answer FALSE must be accompanied by an error witness [Bey15], and requires the organizer to reasonably validate each witness before assigning a success score, in order to get more confidence that the error witness indeed represents a valid bug.

References

[BDD$^+$15] D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, and A. Stahlbauer. Witness Validation and Stepwise Testification across Software Verifiers. In *Proc. ESEC/FSE*, pages 721–733. ACM, 2015.

[Bey14] D. Beyer. Status Report on Software Verification. In *Proc. TACAS*, LNCS 8413, pages 373–388. Springer, 2014.

[Bey15] Dirk Beyer. Software Verification and Verifiable Witnesses (Report on SV-COMP 2015). In *Proc. TACAS*, LNCS 9035, pages 401–416. Springer, 2015.

[BHKW12] D. Beyer, T. A. Henzinger, M. E. Keremoglu, and P. Wendler. Conditional Model Checking: A Technique to Pass Information between Verifiers. In *FSE*. ACM, 2012.