Concurrency Correctness Witnesses with Ghosts

Manuel Bentele[1,2] Dominik Klumpp[1] Frank Schüssele[1]

[1] University of Freiburg, Freiburg im Breisgau, Germany
[2] Hahn-Schickard-Gesellschaft, Villingen-Schwenningen, Germany
{bentele,klumpp,schuessf}@informatik.uni-freiburg.de

June 16, 2023

In the International Competition on Software Verification (SV-COMP) [2], it has been established practice for several years [1] that verifiers accompany a `true` verdict for a *sequential* program with a correctness witness [3]. However, correctness witnesses for *concurrent* programs are conceptually more complex, and have been an open problem. Only recently, a format for concurrency correctness witnesses was proposed [10]. The proposed format is based on the notion of *thread-modular* location invariants. In order to overcome limited expressivity of thread-modular reasoning, we propose an extension of this format. We will present our ongoing work on validating and generating correctness witnesses for concurrent programs in Ultimate [4], and discuss the challenges involved.

Generally, thread-modular annotations are well-suited as a basis for concurrency correctness witnesses. Thread-modular annotations scale across any (even unbounded) number of threads: Individual program locations are annotated with invariants, which must hold whenever some thread is currently at the annotated location. Actions of other threads must not interfere, in the sense that they must not cause a violation of the invariant. In the same spirit as recent developments for sequential concurrency witnesses, witnesses based on thread-modular annotations avoid an explicit description of interleavings, thereby decoupling the witness, as far as possible, from the verifier's internal understanding of control flow and, especially relevant in the concurrency context, atomicity of actions.

However, witnesses based purely on location invariants suffer from the expressivity restrictions and incompleteness of thread-modular reasoning. A correctness witness format should not impose such restrictions, as this would risk invalidating efforts in the development of verification algorithms capable of verifying a larger class of programs. Many programs require more intricate proofs that are, to some degree, aware of the different interleavings of threads. We propose an extension of the suggested format to overcome the expressivity restrictions.

Specifically, we propose to extend the suggested format [10] using the well-known and established concept of *ghost state*. Witnesses may declare (and ini-

tialize) additional variables, called *ghost variables* or *auxiliary variables*. Ghost variables are updated as the program executes (as specified by the witness), but do not influence the program execution. Location invariants can then refer to these ghost variables, and relate ghost variables and program variables. For instance, a ghost variable may record some information about the executed interleaving. An invariant can then perform a case distinction over the ghost variable and allow different values for program variables, depending on the interleaving. The combination of thread-modular reasoning with ghost state is known as the proof rule of Owicki and Gries [8, 7]. This proof rule is known to be relatively complete [9, 5], even for programs with an unbounded number of threads [6].

Our goal is to develop a universal witness format that can be supported by different verifiers and validators. The format should not be tied to the internal notions specific to a particular verification algorithm. We believe that ghost variables can meet this goal. For instance, we expect validation of witnesses to be not much more complex than for the recent proposal [10]: The instrumentation of the program with ghost variables can be performed as a (admittedly non-trivial) syntactic transformation at the beginning of the validation process.

A concrete definition of a practically usable witness format involving ghost variables encounters several challenges. We must define the YAML-based syntax and the semantics of such witnesses. This involves various questions about the scope of ghost variables, evaluation of expressions involving both program variables and ghost variables, supported operations on ghost variables, atomicity of ghost variable updates, etc. We will discuss these challenges, as well as advantages and limitations of our proposal.

References

[1] Dirk Beyer. Software verification with validation of results - (report on SV-COMP 2017). In *TACAS (2)*, volume 10206 of *Lecture Notes in Computer Science*, pages 331–349, 2017.

[2] Dirk Beyer. Competition on software verification and witness validation: SV-COMP 2023. In *TACAS (2)*, volume 13994 of *Lecture Notes in Computer Science*, pages 495–522. Springer, 2023.

[3] Dirk Beyer, Matthias Dangl, Daniel Dietsch, and Matthias Heizmann. Correctness witnesses: exchanging verification results between verifiers. In *SIGSOFT FSE*, pages 326–337. ACM, 2016.

[4] Matthias Heizmann, Max Barth, Daniel Dietsch, Leonard Fichtner, Jochen Hoenicke, Dominik Klumpp, Mehdi Naouar, Tanja Schindler, Frank Schüssele, and Andreas Podelski. Ultimate Automizer and the CommuHash normal form - (competition contribution). In *TACAS (2)*, volume 13994 of *Lecture Notes in Computer Science*, pages 577–581. Springer, 2023.

[5] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3(2):125–143, 1977.

[6] Leonor Prensa Nieto. Completeness of the Owicki-Gries system for parameterized parallel programs. In *IPDPS*, page 150. IEEE Computer Society, 2001.

[7] Susan Owicki. *Axiomatic Proof Techniques for Parallel Programs.* PhD thesis, Cornell University, USA, 1975.

[8] Susan Owicki and David Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6:319–340, 1976.

[9] Susan S. Owicki. A consistent and complete deductive system for the verification of parallel programs. In *STOC*, pages 73–86. ACM, 1976.

[10] Simmo Saan and Julian Erhard. Beyond automaton-based witnesses and location invariants. 4th Workshop on Cooperative Software Verification (COOP 2023), April 2023.