

Correctness Witnesses for Concurrent Software Verification

Csanád Telbisz , Dániel Szekeres 

Department of Artificial Intelligence and Systems Engineering, Budapest University of Technology and Economics

Abstract—Providing proofs of formal verification results and the validation of these proofs have been intensely studied in the last decade. However, the practical applicability of these concepts concerning correctness proofs of concurrent software is still very limited. We propose a novel way to provide and validate correctness witnesses for concurrent program verification based on the idea of event ordering consistency-based decision procedures where properties characterizing invalid scheduling of threads are collected. We show how these properties can be used as witnesses for program correctness and how they can be validated.

I. INTRODUCTION

Software verification algorithms and verifier implementations tend to be very complex and thus bug-prone, reducing the reliability of their proofs and limiting the spread of formal verification in the industry. An emerging trend is to develop validation algorithms and tools to check the verdict of a verifier [1], [2]. Typically, the verifier gets a program and a safety property and produces a verdict on whether the program satisfies the property and a *witness* useful for reestablishing the proof. The validator then gets the same program and property as inputs, along with the produced witness. It has to confirm or refute the verdict of the verifier. In the extreme case, the witness is empty: the validator, in this case, is another verifier with independent algorithms and implementations, making it a form of redundancy. Of course, the verification-validation workflow aims to produce meaningful witnesses. Witnesses for a property violation are typically easier to produce, e.g. by generating an error trace leading to unsafe behavior. Correctness witnesses describe proofs found by the verifier, e.g. a loop invariant [3]. SV-COMP, a competition on software verification, has recently started hosting a validation track where verdicts and witnesses produced by the verifiers in the competition have to be validated [4]. Furthermore, a correct verification verdict only increases the score of a verifier if it is validated by a validator.

In a multi-threaded program, the number of possible executions (thread interleavings) can be exponential in the number of operations and threads, inducing even more complicated algorithms [5], [6]. This highlights the need for validation even more, but there are only a few algorithms and tools for validating concurrent software – even those facing serious limitations. This is showcased by the fact that there is practically no proper validator for safe results in the concurrency category of SV-COMP [4]. This work aims to provide methods for validating safe concurrent programs with certain properties.

Recently, powerful Satisfiability Modulo Theory-based (SMT) algorithms have been developed for verifying concur-

rent programs, symbolically encoding the program into an SMT formula [7], [8], [9]. The initial formula may allow invalid executions of the program, and it is iteratively strengthened by excluding certain invalid executions containing a scheduling inconsistency, a loop in the ordering precedence of instructions. We show how these identified inconsistencies can be used as *correctness witnesses* for a program and describe a validation method for these witnesses.

Related Work: Proofs and witnesses have been used for a long time in verification workflows. Some verifiers provide instrumented code [10], tests reproducing the error [11], [12], or correctness certificates [13], [14]. Correctness proof approaches mainly try to identify invariants, but many suffer from impractically large proofs [15]. A standard format for violation witnesses was first introduced in 2015 [1], later extended to correctness witnesses [16]. This way, verifiers and validators supporting this format can exchange information about the verified program. Though many concepts can be generalized to the concurrent programs [2], there are no specific concurrency-related witness concepts and widely applicable algorithms and validators for safe concurrent programs [4].

II. VERIFICATION WITH WITNESSES AND VALIDATION

The verification algorithm assumes loop-free (or unrolled) concurrent programs communicating with shared variables. We use a partial order-based representation of concurrent programs [7]. Each variable access is represented by an event. An event refers to a memory address, has a guard condition, and has a type *read* or *write*. An *execution* is a sequence of events. A *happens-before* relation $\prec$ between events guarantees the validity of executions: an execution is valid if there is no self-loop in the transitive closure of the happens-before relation (i.e., there is no event e such that $e \prec^* e$). The happens-before relation is the union of several relations. First, the *program order* (po) relation relates events based on the source code: under sequential consistency, an event happens before another if their corresponding instructions occur before one another. Second, the *read-from* (rf) relation relates a write event w and a read event r if r reads its value from w . The rf-related pairs may differ in each execution (for ease of reference, an rf-relation of a write and read event can be enabled or disabled), and it has consequences that must be satisfied (e.g., the value written by w and read by r must be the same). These relations and constraints can be encoded as an SMT formula [9].

The error property can also be encoded as an SMT formula. A satisfiable assignment to the conjunction of the program

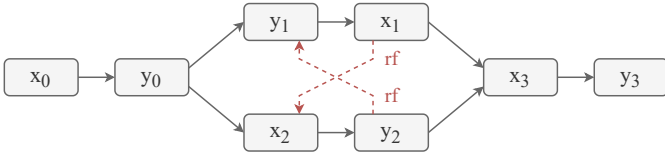


Fig. 1: Events, po-relations and some enabled rf-relations

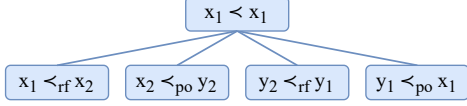


Fig. 2: Derivation tree of the conflict

and happens-before relation constraints and the error property gives a valid execution leading to an error. Unsatisfiability means the program is safe regarding the error property. A standard SMT-solving procedure can be used to determine the satisfiability of the formula. Whenever multiple rf-related pairs can be added, we guess one of them (c.f., decision points in SMT-solving). Then, based on the constraints and an axiomatic system, we derive further happens-before-related events [8]. We must check whether there is a self-loop (or cycle) in the happens-before relation. If we find one, the current execution is invalid as the events cannot be scheduled (an event cannot happen before itself). In this case, we strengthen the formula with a conflict clause excluding this scheduling inconsistency. Finally, we either get a satisfying assignment without inconsistency, or the SMT solver yields an unsatisfiable result. Several solutions implement a similar decision procedure [7], [9].

Based on this algorithm, we propose the following way to produce correctness witnesses. A conflict found during verification can be used as an element of the witness. The full witness is the set of discovered conflicts. A conflict characterizes invalid executions, thus the negation of the conflict must be satisfied by all valid executions. A conflict is a self-loop in the happens-before relation: the conflict clause is the reason of the self-loop (how it was derived). The derivation of a conflict can be represented as a tree that we refer to as a derivation tree. A node of the tree is an element of the happens-before relation. The root is the self-loop formed. The children of a node n are other relations that together imply the relation of n . Leaves are program order or currently enabled rf-relations.

As an example, consider a program whose events and po-relations are represented in Figure 1 (the program code is omitted due to lack of space, but the emphasis is on the conflict and the derivation tree, anyway). Let us assume that the rf-relations marked in the graph are currently enabled. This implies a simple conflict due to the cycle in the happens-before relation (e.g., $x_1 \prec^* x_1$). The reason for the conflict, that is, the derivation tree of the conflict is depicted in Figure 2. This can be exported by the verifier as an element of the witness. Derivation trees can be much more complex in reality when axioms are also used to derive relations as in [9].

In the validation phase, the validator gets the witness,

the verified program, and the error property as input. All information coming from the verifier must be validated. First, the validator checks each element of the witness, if they indeed represent a scheduling inconsistency. This can be done by checking each node of the derivation tree to see if it can be derived from its children; also, a self-loop must be formed in the root. After validating the conflict clauses, the validator checks if the set of conflicts indeed implies the correctness of the program. For this, the program is encoded with the same constraints based on the program structure, but the (validated) conflict clauses in the witness are also added to the formula in the beginning. If the program is indeed safe, and it is indeed proven by the provided witness, the formula should be unsatisfiable, and this result could be obtained without any further derivation that is used by the verifier.

III. EXPERIMENTAL EVALUATION

We aimed to answer the following research questions to evaluate the proposed witness and validation technique:

RQ1 Can the validator validate the witness and the correctness of the program?

RQ2 What is the performance gain achieved by the validator using the witnesses compared to the verifier?

RQ3 What is the computational burden of witness validation?

We carried out the evaluation on a subset of the SV-COMP concurrency category safe benchmarks [4] supported by our proof-of-concept implementation in the open-source verification tool THETA [17]: we had 97 loop-free or safely unrollable tasks. First, we ran the verifier to verify the programs (87 verified before timeout) and produce witnesses, then we used the validator to validate the results and proofs of the verifier.

RQ1 All generated conflicts were validated by the validator, so derivation trees indeed provide enough information to reestablish the scheduling inconsistencies. The safety of the program could be reestablished in all 87 cases.

RQ2 The validator took significantly less time, especially considering the SMT solver time (66 seconds aggregated for all tasks compared to 320 seconds in case of the verifier). The difference is somewhat reduced in the overall CPU time (408 seconds, and 689 seconds, respectively) due to common overheads such as parsing.

RQ3 Conflict validation being 0.7% of all CPU time of validation on average is not a significant part of validation.

The typical size of a single conflict is not huge, even human-readable in most cases. However, the number of conflicts (210 on average) is often above the limits of human processing.

IV. CONCLUSION

We proposed an approach to provide and validate correctness witnesses for concurrent programs. Experimental evaluation has shown that validators can efficiently confirm the safety of the program using the witnesses. The witnesses may also provide meaningful insight to human readers on the correctness of programs. Continuing our research, we will strive to reduce limitations of the approach (e.g., combining with loop invariant techniques to handle programs with complex loops).

REFERENCES

- [1] D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, and A. Stahlbauer, “Witness validation and stepwise testification across software verifiers,” E. D. Nitto, M. Harman, and P. Heymans, Eds. ACM, 2015, pp. 721–733.
- [2] D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, T. Lemberger, and M. Tautschnig, “Verification witnesses,” *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 4, pp. 57:1–57:69, 2022.
- [3] D. Beyer, M. Dangl, and P. Wendler, “Boosting k-induction with continuously-refined invariants,” ser. Lecture Notes in Computer Science, D. Kroening and C. S. Pasareanu, Eds., vol. 9206. Springer, 2015, pp. 622–640.
- [4] D. Beyer, “State of the art in software verification and witness validation: SV-COMP 2024,” ser. Lecture Notes in Computer Science, B. Finkbeiner and L. Kovács, Eds., vol. 14572. Springer, 2024, pp. 299–329.
- [5] P. A. Abdulla, S. Aronis, B. Jonsson, and K. Sagonas, “Optimal dynamic partial order reduction,” S. Jagannathan and P. Sewell, Eds. ACM, 2014, pp. 373–384.
- [6] J. Huang, “Stateless model checking concurrent programs with maximal causality reduction,” D. Grove and S. M. Blackburn, Eds. ACM, 2015, pp. 165–174.
- [7] J. Alglave, D. Kroening, and M. Tautschnig, “Partial orders for efficient bounded model checking of concurrent software,” ser. Lecture Notes in Computer Science, N. Sharygina and H. Veith, Eds., vol. 8044. Springer, 2013, pp. 141–157.
- [8] F. He, Z. Sun, and H. Fan, “Satisfiability modulo ordering consistency theory for multi-threaded program verification,” S. N. Freund and E. Yahav, Eds. ACM, 2021, pp. 1264–1279.
- [9] Z. Sun, H. Fan, and F. He, “Consistency-preserving propagation for SMT solving of concurrent program verification,” *Proc. ACM Program. Lang.*, vol. 6, no. OOPSLA2, pp. 929–956, 2022.
- [10] H. Rocha, R. S. Barreto, L. C. Cordeiro, and A. D. Neto, “Understanding programming bugs in ANSI-C software using bounded model checking counter-examples,” ser. Lecture Notes in Computer Science, J. Derrick, S. Gnesi, D. Latella, and H. Treharne, Eds., vol. 7321. Springer, 2012, pp. 128–142.
- [11] H. S. Hong, I. Lee, O. Sokolsky, and S. D. Cha, “Automatic test generation from statecharts using model checking,” in *Workshop on Formal Approaches to Testing of Software (FATES)*, 2001, pp. 15–30.
- [12] L. Bajczi, Z. Ádám, and Z. Micskei, “Concurrentwitness2test: Test-harnessing the power of concurrency (competition contribution),” ser. Lecture Notes in Computer Science, B. Finkbeiner and L. Kovács, Eds., vol. 14572. Springer, 2024, pp. 330–334.
- [13] T. A. Henzinger, R. Jhala, R. Majumdar, G. C. Necula, G. Sutre, and W. Weimer, “Temporal-safety proofs for systems code,” ser. Lecture Notes in Computer Science, E. Brinksma and K. G. Larsen, Eds., vol. 2404. Springer, 2002, pp. 526–538.
- [14] A. Iliasov, “Generation of certifiably correct programs from formal models,” R. Pietrantuono and N. Silva, Eds. IEEE Computer Society, 2011, pp. 43–48.
- [15] E. Ghassabani, A. Gacek, and M. W. Whalen, “Efficient generation of inductive validity cores for safety properties,” T. Zimmermann, J. Cleland-Huang, and Z. Su, Eds. ACM, 2016, pp. 314–325.
- [16] D. Beyer, M. Dangl, D. Dietsch, and M. Heizmann, “Correctness witnesses: exchanging verification results between verifiers,” T. Zimmermann, J. Cleland-Huang, and Z. Su, Eds. ACM, 2016, pp. 326–337.
- [17] L. Bajczi, C. Telbisz, M. Somorjai, Z. Ádám, M. Dobos-Kovács, D. Szekeres, M. Mondok, and V. Molnár, “Theta: Abstraction based techniques for verifying concurrency (competition contribution),” ser. Lecture Notes in Computer Science, B. Finkbeiner and L. Kovács, Eds., vol. 14572. Springer, 2024, pp. 412–417.