Software Verification Witnesses for Weak Memory

Levente Bajczi
Budapest University of Technology and Economics
Department of Artificial Intelligence and Systems Engineering

Marian Lingsch-Rosenfeld
Ludwig-Maximilians-Universität München
Department of Computer Science

***Abstract*—In software verification, correctness and violation witnesses are essential for exchanging information between tools, ensuring the trustworthiness of verdicts through validation, and explaining the results to verification engineers. Despite advances in memory-model-aware verification, there is no standardized format for witnesses with weak memory semantics, hindering easy validation of verdicts. This paper proposes a way to standardize witnesses for weak memory semantics, facilitating broader interoperability and enhancing trust in verification tools.**

I. Introduction

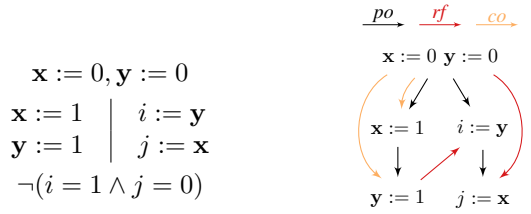
In software verification, *correctness* and *violation* witnesses serve as a way to exchange information between verification tools [1]. Such exchange of knowledge is vital to ensure the trustworthiness of verdicts, as witnesses can support a binary *safe* or *unsafe* verdict with auxiliary information, which can help an easier reproduction of the same result using other tools, in a process called *validation* [2]. However, validators may not use the same internal program representation or verification algorithms and techniques, therefore a standardized way of representing witnesses is crucial in this process.

The format of verification witnesses has evolved alongside the software verification competition (SV-COMP), where all participating verification tools produce witnesses. If the witness cannot be validated, no points will be awarded for a correct verdict. Currently, the format version 2.0 [1] is being phased in for verification witnesses, which ties the semantics of witnesses to the program itself (as opposed to an automata-based approach in previous versions [3]), and uses a more lightweight, YAML-based syntax as opposed to the previous GraphML format [3]. These changes have received approval from tool developers for both the verification and validation.

Memory-model-aware verification – where the memory access semantics of the target architecture also needs to be taken into account for verification [4] – has been seeing widespread adoption in recent years [5], [6]. However, until now, no standardized format for witnesses has been proposed for either *counterexamples* or *proofs*, and therefore, their verdicts cannot be easily validated. In this paper, we aim to bridge this gap by proposing a standardized format for witnesses with weak memory semantics, building on the existing witness infrastructure, and extending the novel format version 2.0.

II. Execution Semantics for Weak Memory

We assume similar semantics to most memory-model-aware verification techniques [4], [6], [5], [7], where program execution is defined by a directed and edge-labeled graph, where nodes correspond to *memory events* (read- and write accesses and fences) in some program path, and edges correspond to relations over these memory events. Given a global order of same-variable write events (*coherence-order, co*) and exactly one incoming *read-from (rf)* relation for each read event, the ordering of instructions in the program's source (*program order, po*) can identify exactly one *candidate execution* of the program. Then, the memory model can check for consistency and decide if the execution is *allowed* on the target architecture.



(a) Program with property (b) Counterexample candidate

Fig. 1: A multi-threaded program and a candidate execution. **Bold** variables are global, *italicized* variables are local. The program is safe under SC and TSO but not under PSO

Verification techniques typically aim to generate such execution graphs where a safety property is violated. If any of such executions is found to be consistent, the program is faulty. As an example, see Figure 1: a candidate execution violating the safety property is generated (in Figure 1b), using *co*, *rf*, and *po* edges over memory accesses in the program. If this candidate is consistent with a memory model's constraints, then the safety violation can happen on the corresponding architecture. For instance, Sequential Consistency (SC) and Total Store Ordering (TSO) will preserve safety, while Partial store Ordering (PSO) will allow the execution to happen [8].

III. Violation Witnesses

To encode a violation witness for a *counterexample*[1] of a program given a memory model, we need to encode the paths of the program threads leading to an error location, as well as the *co* and *rf* relations of the memory accesses. However, we cannot assume that these relations are readily available for all verification tools because not every memory-model-aware verification technique necessitates keeping track of such relations. In contrast, simply reasoning about the order

[1] We assume reachability properties (can any thread reach an error location).

https://doi.org/00.0000/00000000

Thread 0	waypoint type	value	line	column
	assume	$\backslash at(\mathbf{x}, 0) = 0$	0	*middle*
	assume	$\backslash at(\mathbf{y}, 0) = 0$	0	*end*
	thread_start	1, 2	1	0
Thread 1				
	assume	$\backslash at(\mathbf{x}, 1) = 1$	1	*end*
	assume	$\backslash at(\mathbf{y}, 1) = 1$	2	*end*
Thread 2				
	assume	$i = \backslash at(\mathbf{x}, 1)$	1	*end*
	assume	$j = \backslash at(\mathbf{y}, 1)$	2	*end*
	target	-	2	*end*

Table I: A violation witness, encoding a violation under PSO for Figure 1b

invariant type	value	line	column
location	$\backslash at(\mathbf{x}, 0) = 0$	0	*middle*
location	$\backslash at(\mathbf{y}, 0) = 0$	0	*end*
location	$\backslash at(\mathbf{x}, 1) = 1$	1 *(left)*	*end*
location	$\backslash at(\mathbf{y}, 1) = 1$	2 *(left)*	*end*
location	$\exists a : a \in \{0, 1\}$ $i = \backslash at(\mathbf{x}, a)$	1 *(right)*	*end*
location	$\exists a, b : a, b \in \{0, 1\}$ $j = \backslash at(\mathbf{y}, a)$ $i = \backslash at(\mathbf{x}, b)$ $b = 1 \implies a = 1$	2 *(right)*	*end*
location	$\neg(i = 1 \wedge j = 0)$	2 *(right)*	*end*

Table II: A correctness witness, encoding a proof over SC for Figure 1a

of instruction executions should be intuitive for all algorithms, given the nature of memory models.

Therefore, we rely on a variation of the `\at(<var>, <label>)` construct of the ANSI/ISO C Specification Language (ACSL) [9] to provide a way to express the global ordering of write events (similar to *co*), by constraining the `<label>` parameter to be a natural number. Thus, we can exploit the ordering capabilities of integers and assign the following semantics to the construct (x is a variable, $W^c_{\mathbf{x}}$ is a write to $\mathbf{x}$ with a value of c):

$$\forall i, j \in \mathbb{N}. \quad \begin{array}{l} i < j \wedge \\ \backslash at(\mathbf{x}, i) = c_i \wedge \\ \backslash at(\mathbf{x}, j) = c_j \end{array} \implies W^{c_i}_{\mathbf{x}} \prec_{co} W^{c_j}_{\mathbf{x}}$$

Informally, declaring `\at(x, i) = c` is equivalent to saying that *a write to x with a value of c is globally the i^{th} write to x*, and therefore, if two writes to the same variable have different indices, then the natural order of the indices define a *co*-relation. Furthermore, we can use the *assumption* waypoints for encoding the *rf*-relation by reducing it to an expression about the value received from the *read* instruction. For the candidate execution in Figure 1b, we could formulate the statements in Table I. A potential validator (employing execution graph-based validation) could check that the encoded *co* and *rf* relations are consistent with a memory model, and furthermore, the provided waypoints do take the execution to the target waypoint.

IV. Correctness Witnesses

In contrast to violation witnesses, correctness witnesses need to encode *invariant properties* of the program to provide over-approximations of the program behavior, which will help prove the program correct. Informally, for a safe program given a memory model, we need to encode that ordering constraints stemming from the memory model prohibit the establishment of a *consistent* execution graph that results in a target location. To achieve this, we can rely on `\at(<var>, <label>)`, the same as for *violation witnesses*.

As an example, assuming Sequential Consistency (SC), we can encode the safety proof of Figure 1a in Table II. Informally, we encoded that the first read can read either from the 1^{st} or 2^{nd} write, and the second read can read from either write normally, but only from the 2^{nd} write if the previous read was also from its corresponding 2^{nd} write. Notice we do not use thread IDs for correctness witnesses: we expect the proofs to include true location invariants, meaning the invariant shall hold in any reachable state corresponding to the program location in any thread.

V. Application to Sequential Consistency

Although the main use-case of the presented witness format is to allow memory-model-aware verifiers to exchange information with each other, it is also possible to use the technique above to represent witnesses for sequentially consistent (naively concurrent) programs. One advantage of this technique is that it does not rely on (nor allows) invariants or assumptions over global variables relying only on snapshots of their values. This is important, since the value of a global variable could have hard-to-interpret semantics due to the concurrent nature of their modifications.

To transform an interleaving-based *violation witness* to this format, one must only add `\at(x, i) = c` waypoints with increasing i values per variable to each global write in some ordering of the threads that causes the violation to occur, and instead of assumptions over the values of global variables, include the `\at(x, i)` construct with the last i value.

To transform an interleaving-based *correctness witness* to this format, one shall also add `\at(x, i) = c` invariants with increasing i values (either as literals or as symbolic values depending on the previous program paths) per variable to each global write, and instead of invariants over the values of global variables, include a universally quantified `\at(x, i)` construct with all possible i values at the location.

References

[1] P. Ayaziová, D. Beyer, M. Lingsch-Rosenfeld, M. Spiessl, and J. Strejček, "Software verification witnesses 2.0," in *Proc. SPIN*. Springer, 2024.

[2] D. Beyer and J. Strejček, "Case study on verification-witness validators: Where we are and where we go," in *Proc. SAS*, ser. LNCS 13790. Springer, 2022, pp. 160–174.

[3] 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.

[4] J. Alglave, L. Maranget, and M. Tautschnig, "Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory," *ACM Trans. Program. Lang. Syst.*, vol. 36, no. 2, jul 2014. [Online]. Available: https://doi.org/10.1145/2627752

[5] N. Gavrilenko, H. Ponce-de León, F. Furbach, K. Heljanko, and R. Meyer, "BMC for Weak Memory Models: Relation Analysis for Compact SMT Encodings," in *Computer Aided Verification*, I. Dillig and S. Tasiran, Eds. Cham: Springer International Publishing, 2019, pp. 355–365.

[6] M. Kokologiannakis and V. Vafeiadis, "GenMC: A Model Checker for Weak Memory Models," in *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part I*. Berlin, Heidelberg: Springer-Verlag, 2021, p. 427–440. [Online]. Available: https://doi.org/10.1007/978-3-030-81685-8_20

[7] F. He, Z. Sun, and H. Fan, "Satisfiability modulo ordering consistency theory for multi-threaded program verification," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. Pldi 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 1264–1279. [Online]. Available: https://doi.org/10.1145/3453483.3454108

[8] H. Fan, Z. Sun, and F. He, "Satisfiability modulo ordering consistency theory for sc, tso, and PSO memory models," *ACM Trans. Program. Lang. Syst.*, vol. 45, no. 1, pp. 6:1–6:37, 2023. [Online]. Available: https://doi.org/10.1145/3579835

[9] P. Baudin, P. Cuoq, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto, "ACSL: ANSI/ISO C specification language version 1.17," 2021, available at https://frama-c.com/download/acsl-1.17.pdf.