Safeguarding AI-Based Software Development and Verification using Witnesses (Position Paper)

Dirk Beyer

LMU Munich, Munich, Germany

Abstract. This position paper accompanies a presentation and discussion at AISoLA 2023 on the topic of how (generative) AI influences software engineering, namely, the development and verification of software programs. We outline a few opportunities and challenges that are posed by the application of AI. AI-based techniques can be an efficient method to produce software code. Not only this, AI can also be efficient in producing invariants that help proving correctness of software programs. But unfortunately, the results generated by AI-based approaches are often still imprecise or wrong: Code produced with the help of AI often does not satisfy the specification, and AI-generated invariants are often not helpful to prove the correctness of the program. The solution is to safeguard the process by independently checking the results using verification witnesses and witness validation. The goal of this paper is to convince readers that software witnesses are important and that independent result validation is of utmost importance to ensure correctness.

Keywords: Software Verification · Software Development · Generative AI · Machine Learning · Formal Methods · Software Correctness · Invariants · Verification Witnesses · Error Reports · Exchange Formats · Explainability

1 Motivation

Generative AI has applications in almost all areas of software engineering, from requirements engineering over code generation to quality assurance. This discussion at ISoLA focused on three applications of AI (1–3 below) that are all important since they affect the correctness of the software, have a large potential for improvement through AI, and, on the negative side, currently still suffer from imprecision of the AI techniques. The three approaches share another commonality: Their negative impact on the correctness can be addressed by the same solution: witness-based validation of result.

(1) Generative AI can be used to suggest software code. It seems that the more integrated this technique is in the software-development process, the more productive it becomes. Recent empirical studies show that approaches like GitHub Copilot can significantly improve the productivity, especially if developers are trained on how to use them. There is a large body of literature on this topic

describing the state of the art [1, 2]. The problem is that generated code often contains bugs that are sometimes obvious but sometimes quite subtle and therefore missed by developers.

(2) Whenever software has to solve an intractable problem, that is, a problem for which no efficient algorithm is known, heuristics are applied that work well in certain circumstances [3]. Often, there are many different heuristics from which the developer or user can choose. For example, in software verification, the tools PeSCo [4, 5] and Graves-CPA [6, 7] use an ML-based algorithm selection to choose the most promising configuration from the software-verification framework CPAchecker [8, 9]. The problem here is that the best configuration selected by the AI might turn out to be the wrong choice and produce a wrong result and another configuration producing the correct result remains unused. For example, the AI may select a configuration without pointer analysis (because it might have been fast on other programs without pointers during training) for a program that has pointers.

(3) AI can be used to generate invariants that accelerate the construction of a proof of correctness. Machine learning has been used to infer loop invariants for programs [10]. Also, termination arguments [11] have been derived by using neural networks to represent ranking functions. Generative AI can be used as code pilots for interactive theorem proving [12]. A recent study showed that generative AI (ChatGPT) can be used to generate loop invariants that help Frama-C prove the correctness of a program [13]. The problem is that the suggested predicates might not be valid invariants, might not be inductive, or might not aid in proving the safety property.

> Imprecise AI-based techniques should be safeguarded by techniques that verify the result, like witness-based result validation.

2 Solution

We would like to advocate verification witnesses as a means towards solving the above-mentioned problems, that is, by safeguarding software development and verification with the help of verification witnesses.

2.1 Related Work

Traditionally, verifiers returned claims of the form *true* (system satisfies the specification) or *false* (system violates the specification) and the user is left alone with the result. Any kind of useful information that *explains* the result would improve the situation. The formal-methods community has established the practice of witness validation, in the area of software verification [14, 15], termination checking [16], SAT solving [17, 18], SMT solving [19, 20], and hardware model checking [21, 22]. Witnesses were also used for graph algorithms [23]. The technique of execution reports [24] was investigated to provide analysis results

in a more structured way, and the format SARIF [25] is supported by some program-analysis tools. Information exchange also enables the integration of tools, for example, the Evidential Tool Bus (ETB) [26, 27, 28] supported tool integration by producing claims that are supported by evidence, and the Electronic Tools Integration (ETI) [29, 30, 31] offered a platform and web service for integrating model-checking tools. Also cooperative verification [32, 33, 34] requires information in standard exchange formats and tools should be collected using a standard format with information about the tools [35].

2.2 Verification Witnesses

Nowadays, software verifiers produce witnesses to certify their result [14, 15, 36]. The competition on software verification (SV-COMP) uses verification witnesses since 2015 [37]. The most recent advancement is that the competition on software verification [38] has for its 2023 edition introduced a new track for the evaluation of tools for witness validation [39]. Furthermore, to make verification witnesses more human-readable and more concise, and semantically well-defined, the community developed a new version 2.0 of the widely supported exchange format for verification witnesses [40]. This new standard format was immediately adopted in the competition in its 2024 edition [41].

Software verification is the process of producing, for a given program P and specification φ, a verdict (from *true*, *false*, and *unknown*) and a verification witness w (a correctness witness for verdict *true* and a violation witness for verdict *false*). The verdict *true* means that φ holds for P and the verification tool has constructed a proof of correctness π (denoted as $\pi : P \models \varphi$). In this case, the *correctness* witness contains program invariants aiding the construction of the correctness proof. The verdict *false* means that φ is violated by P and the verification tool has constructed a counterexample to the proof of correctness π (denoted as $\pi : P \not\models \varphi$). In this case, the *violation* witness describes at least one error path through P that violates φ, and for the proof of violation it suffices to analyze the semantics of the program along the described error paths.

Software validation in this context is the process of reestablishing a verdict, for a given program p, specification φ, and verification witness w. To reconstruct a proof of correctness $\pi' : P \models \varphi$, the validator takes the invariants stored in w and checks if they hold. If this is the case, then the validator can use the invariants as lemmata in its own proof of correctness π'. If the invariants in the witness hold and the program satisfies the specification, then the witness is *valid for verdict true*. To reestablish a verdict *false*, the validator explores the paths described by the witness and checks if they are feasible. If a feasible path is found, then the validator can use the path to check whether the strongest post-condition of the operations along the path leads to a specification violation. If a feasible path described by the witness violates the specification, then the witness is *valid for verdict false*. We refer to the literature [15, 40] for more details.

Program:

```
#include <assert.h>
extern unsigned char
↪ __nondet_uchar(void);

int main() {
  unsigned char n =
  ↪ __nondet_uchar();
  if (n == 0) {
    return 0;
  }
  unsigned char v = 0;
  unsigned int  s = 0;
  unsigned int  i = 0;
  while (i < n) {
    v = __nondet_uchar();
    s += v;
    ++i;
  }
  assert(s >= v);
  assert(s <= 65025);
  return 0;
}
```

Specification:

All assertions in the program must hold.

Witness:

```
- entry_type: invariant_set
  metadata:
    format_version: "2.0"
    producer:
      name: "CPAchecker"
  content:
    - invariant:
        type: loop_invariant
        location:
          file_name: "inv-a.c"
          line: 12
          column: 3
          function: main
        value: "s <= i*255 &&
        ↪ 0 <= i && i <= 255 &&
        ↪ n <= 255"
        format: c_expression
```

Fig. 1: Example C program similar to inv-a.c (left, adopted from [15]), satisfying the given specification (top right), and correctness witnesses in format 2.0 (right, metadata shortened, with a single nontrivial invariant, adopted from [40])

2.3 Example

For illustration, we show an example from the literature [40] in Fig. 1 for a verification witness that contains a loop invariant: The figure shows a C program (left) and a specification (top right), together with a correctness witness in format 2.0 (right). The program mainly consists of a loop in which `n` values are read into variable `v` and summed up in variable `s`. The specification is to ensure that no assertion is violated. The first assertion requires that the sum `s` is at least as large as the last read value `v`. The second assertion requires that the sum `s` is less than or equal to 65025.

The creative task for program verification is now to come up with a loop invariant. For example, let us consider $s \leq i * 255 \wedge 0 \leq i \wedge i \leq 255 \wedge n \leq 255$. This predicate captures the knowledge that the upper bound of variable `s` is `i` times the largest possible value for `v`. Since variable `v` has type `unsigned char`, its largest possible value is 255. It further tells us that variable `i` has only values from 0 to 255, because it starts with 0, is counted up by 1, and its largest value is the largest value of `n`, which is 255. The invariant also tells us that the value of `n` is bound by 255, because it is of type `unsigned char`.

Table 1: Witness Validators in SV-COMP 2024, with literature references, the language they support, since when the tools exist, and which formats they support ('viol' short for violation witnesses, 'corr' short for correctness witnesses)

Validator	Reference	Lang.	Since	Supported Formats			
				1.0-viol	1.0-corr	2.0-viol	2.0-corr
CPAchecker	[14, 36, 42]	C	2015	✓	✓	✓	✓
UAutomizer	[14, 36]	C	2016	✓	✓		✓
CPA-witness2test	[43]	C	2018	✓			
CProver-witness2test	[43]	C	2018	✓			
MetaVal	[44]	C	2020	✓	✓		
NITWIT	[45]	C	2020	✓			
WitnessLint	[40]	C	2021	✓	✓	✓	✓
Dartagnan	[46]	C	2022	✓			
GWIT	[47]	Java	2022	✓			
Symbiotic-Witch	[48]	C	2022	✓			
WIT4JAVA	[49]	Java	2022	✓			
ConcurWitness2Test	[50]	C	2024	✓			
Goblint	[51]	C	2024				✓
JCWIT		Java	2024		✓		
LIV	[52]	C	2024		✓		
Mopsa	[53]	C	2024				✓
Witch	[40, 54]	C	2024			✓	

This predicate has three interesting properties: (a) It is a *loop invariant*, because it holds at every iteration before the evaluation of the loop head (beginning of the loop). (b) It is *inductive*, because if assumed at the loop head, it holds again at the next visit of the loop head. (c) It is *safe*, because it implies that the safety specification holds. The latter is true because both assertions hold. The first assertion holds because after `v` (with a value in the interval $[0, 255]$) is added to `s`, the value of `s` is at least as large as the value of `v`. The variable `s` cannot overflow because it is of type `unsigned int`, which is sufficiently large and `s` does not grow larger than 65025 (see next assertion). The second assertion holds because `s` is always less than or equal to the product of the largest possible values for variables `n` and `v`, which are both bound by 255.

No matter how a verification tool came up with such an invariant, witness validation is available to safeguard the verification result: A witness-based result validator takes as input the program, the specification, and the witness, and checks whether the claimed invariant really holds and the program fulfills the specification.

Figure 1 shows a correctness witness on the right. Besides the invariant, the witness format captures all necessary and useful information, such as the precise location at which the invariant holds, the format in which the invariant is specified, that the invariant is a loop invariant, and metadata about the producer of the witness and the verification task.

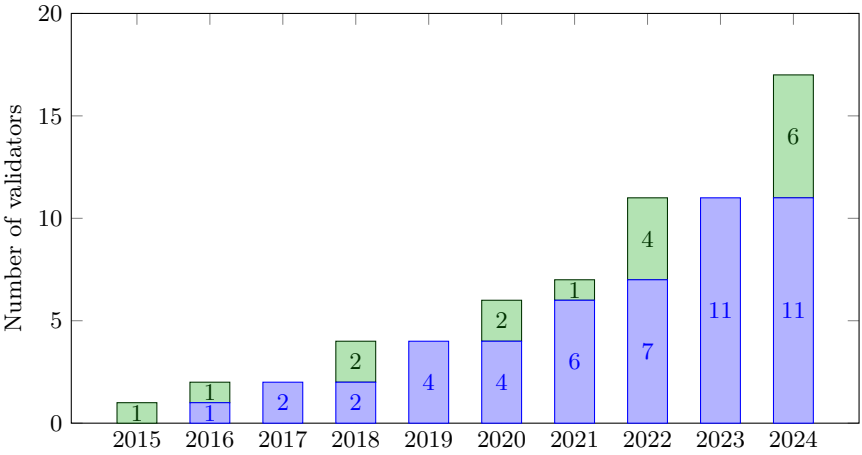

Fig. 2: Number of witness validators evaluated in SV-COMP for each year (first-time participants on top), taken from 2024 report [41, page 317]

2.4 Available Witness Validators

Validation of verification results becomes more and more important, as the research on witnesses and their validation matures [39]. The most recent competition on software verification has compared 17 validators [41] (including a syntax checker WitnessLint). There are currently two kinds of verification witnesses considered: correctness witnesses and violation witnesses. Also, there are two versions of the format for verification witnesses: The first format, version 1.0, is from 2015 [14] and based on GraphML (XML) [55]. It describes a witness automaton. The second format, version 2.0, is from 2023 [40] and based on YAML format. In sum, there are four combinations of witness kind and witness format, version 1.0 for violation ('1.0-viol'), version 1.0 for correctness ('1.0-corr'), version 2.0 for violation ('2.0-viol'), and version 2.0 for correctness ('2.0-corr'). Table 1 lists all validators, their references, supported languages, and supported formats. Figure 2 shows how the number of available witness validators developed over the last ten years. There is strong interest in developing tools for validation of verification results, which is a valuable enabling factor for the use of imprecise techniques in verification tools that compute invariants or error paths.

2.5 Safeguarding Software Development

Now we are equipped with witnesses and can address the three problems outlined in the motivation Sect. 1. First, programs should be accompanied by behavioral specifications and verification should be applied to ensure that the program fulfills the specification. Ideally, the programs are annotated with assertions [56], which are an in-code form of specification. If code is AI-generated, the assertions (and other annotations [57]) should also be generated, or manually added as the contract

between the generated code and its context. As the volume of AI-generated code will grow in the near future, but the precision of the code will still be suboptimal, the correctness of the software must be ensured by verification.

Second, if a particular verification tool is selected or automatically configured (for example by an AI-based selection), than the result of the verifier —that is, the verification witness— must be validated by a witness validator. This way, the verification process is less vulnerable to the risk that a verification tool was wrongly configured and produced a wrong result.

Third, and finally, there is no risk in using AI-generated invariants *if* the invariants are validated by a witness validator. That is, the generated invariants are put into a correctness witness and then given together with the program to the validator to be checked for validity. Note that it is not important for the validation process whether the invariants are annotated in the program or given as a correctness witness: both are interchangeable [58].

3 Conclusion

Imprecise approaches (such as AI-based code generation and AI-based invariant generation) can safely be used in the development and verification of software systems if the results are independently checked for correctness. That is, generated code should be analyzed to make sure it adheres to the specification, and generated invariants should be checked by witness-based result validators. The competition on software verification (SV-COMP 2024) has evaluated 16 validators for software witnesses (correctness witnesses and violation witnesses) and shown that their quality is very good. In conclusion, it seems that imprecise AI-based techniques can be empowered by techniques that safeguard the result, like witness-based result validation.

Funding Statement. My work on this topic was supported in part by the Deutsche Forschungsgemeinschaft (DFG) – 418257054 (COOP).

Acknowledgements. I would like to thank M. Dangl, D. Dietsch, M. Heizmann, T. Lemberger, A. Stahlbauer, and M. Tautschnig for the collaboration on developing the concepts, format, and tools for the verification witnesses in format 1.0 [14, 15], and P. Ayaziová, M. Lingsch-Rosenfeld, M. Spiessl, J. Strejček, and N. Weise for the collaboration on developing the new witness format 2.0 [40]. I am very grateful that the idea of verification witnesses became widely adopted in the community of software verification [59] (more than 60 verifiers generate witnesses) and the developers of the validators (16 validators were created, see Table 1) to support witness validation.

References

1. Wang, S., Geng, M., Lin, B., Sun, Z., Wen, M., Liu, Y., Li, L., Bissyandé, T.F., Mao, X.: Natural language to code: How far are we? In: Proc. FSE. pp. 375–387. ACM (2023). https://doi.org/10.1145/3611643.3616323

2. Shin, J., Nam, J.: A survey of automatic code generation from natural language. J. Inform. Processing Systems **17**(3), 537–555 (June 2021). https://doi.org/10.3745/JIPS.04.0216
3. Rice, J.R.: The algorithm selection problem. Adv. Comput. **15**, 65–118 (1976). https://doi.org/10.1016/S0065-2458(08)60520-3
4. Richter, C., Hüllermeier, E., Jakobs, M.C., Wehrheim, H.: Algorithm selection for software validation based on graph kernels. Autom. Softw. Eng. **27**(1), 153–186 (2020). https://doi.org/10.1007/s10515-020-00270-x
5. Richter, C., Wehrheim, H.: PeSCo: Predicting sequential combinations of verifiers (competition contribution). In: Proc. TACAS (3). pp. 229–233. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_19
6. Leeson, W., Dwyer, M.B.: Algorithm selection for software verification using graph neural networks. arXiv/CoRR **2201**(11711) (January 2022). https://doi.org/10.48550/arXiv.2201.11711
7. Leeson, W., Dwyer, M.: Graves-CPA: A graph-attention verifier selector (competition contribution). In: Proc. TACAS (2). pp. 440–445. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_28
8. Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: Proc. CAV. pp. 184–190. LNCS 6806, Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_16
9. Baier, D., Beyer, D., Chien, P.C., Jakobs, M.C., Jankola, M., Kettl, M., Lee, N.Z., Lemberger, T., Lingsch-Rosenfeld, M., Wachowitz, H., Wendler, P.: Software verification with CPAchecker 3.0: Tutorial and user guide. In: Proc. FM. LNCS, Springer (2024)
10. Si, X., Dai, H., Raghothaman, M., Naik, M., Song, L.: Learning loop invariants for program verification. In: Proc. NeurIPS. pp. 7762–7773. Curran Associates (2018), https://dl.acm.org/doi/pdf/10.5555/3327757.3327873
11. Giacobbe, M., Kröning, D., Parsert, J.: Neural termination analysis. In: Proc. ESEC/FSE. pp. 633–645. ACM (2022). https://doi.org/10.1145/3540250.3549120
12. Song, P., Yang, K., Anandkumar, A.: Towards large language models as copilots for theorem proving in Lean. In: Proc. MATH-AI (2023), https://mathai2023.github.io/papers/4.pdf
13. Janßen, C., Richter, C., Wehrheim, H.: Can ChatGPT support software verification? In: Proc. FASE. pp. 266–279. LNCS 14573, Springer (2024). https://doi.org/10.1007/978-3-031-57259-3_13
14. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Stahlbauer, A.: Witness validation and stepwise testification across software verifiers. In: Proc. FSE. pp. 721–733. ACM (2015). https://doi.org/10.1145/2786805.2786867
15. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Lemberger, T., Tautschnig, M.: Verification witnesses. ACM Trans. Softw. Eng. Methodol. **31**(4), 57:1–57:69 (2022). https://doi.org/10.1145/3477579
16. Sternagel, C., Thiemann, R.: The certification problem format. In: Proc. UITP. pp. 61–72. EPTCS 167, EPTCS (2014). https://doi.org/10.4204/EPTCS.167.8
17. Heule, M.J.H.: The DRAT format and drat-trim checker. CoRR **1610**(06229) (October 2016), https://arxiv.org/abs/1610.06229
18. Wetzler, N., Heule, M.J.H., Jr., W.A.H.: Drat-trim: Efficient checking and trimming using expressive clausal proofs. In: Proc. SAT. pp. 422–429. LNCS 8561, Springer (2014). https://doi.org/10.1007/978-3-319-09284-3_31
19. Bury, G.: Dolmen: A validator for SMT-LIB and much more. In: Proc. SMT Workshop. CEUR Workshop Proceedings, CEUR (2021), https://ceur-ws.org/Vol-2908/short4.pdf

20. Bury, G., Bobot, F.: Verifying models with Dolmen. In: Proc. SMT Workshop. CEUR Workshop Proceedings, CEUR (2023), https://ceur-ws.org/Vol-3429/short9.pdf
21. Yu, E., Biere, A., Heljanko, K.: Progress in certifying hardware model checking results. In: Proc. CAV. pp. 363–386. LNCS 12760, Springer (2021). https://doi.org/10.1007/978-3-030-81688-9_17
22. Ádám, Z., Beyer, D., Chien, P.C., Lee, N.Z., Sirrenberg, N.: Btor2-Cert: A certifying hardware-verification framework using software analyzers. In: Proc. TACAS (3). pp. 129–149. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_7
23. McConnell, R.M., Mehlhorn, K., Näher, S., Schweitzer, P.: Certifying algorithms. Computer Science Review **5**(2), 119–161 (2011). https://doi.org/10.1016/j.cosrev.2010.09.009
24. Castaño, R., Braberman, V.A., Garbervetsky, D., Uchitel, S.: Model checker execution reports. In: Proc. ASE. pp. 200–205. IEEE (2017). https://doi.org/10.1109/ASE.2017.8115633
25. OASIS: Static analysis results interchange format (sarif) version 2.0 (2019), https://docs.oasis-open.org/sarif/sarif/v2.0/csprd02/sarif-v2.0-csprd02.html
26. Rushby, J.M.: An Evidential Tool Bus. In: Proc. ICFEM. pp. 36–36. LNCS 3785, Springer (2005). https://doi.org/10.1007/11576280_3
27. Cruanes, S., Hamon, G., Owre, S., Shankar, N.: Tool integration with the Evidential Tool Bus. In: Proc. VMCAI. pp. 275–294. LNCS 7737, Springer (2013). https://doi.org/10.1007/978-3-642-35873-9_18
28. Cruanes, S., Heymans, S., Mason, I., Owre, S., Shankar, N.: The semantics of Datalog for the Evidential Tool Bus. In: Specification, Algebra, and Software. pp. 256–275. Springer (2014). https://doi.org/10.1007/978-3-642-54624-2_13
29. Margaria, T., Nagel, R., Steffen, B.: Remote integration and coordination of verification tools in jETI. In: Proc. ECBS. pp. 431–436 (2005). https://doi.org/10.1109/ECBS.2005.59
30. Steffen, B., Margaria, T., Braun, V.: The Electronic Tool Integration platform: Concepts and design. STTT **1**(1-2), 9–30 (1997). https://doi.org/10.1007/s100090050003
31. Margaria, T.: Web services-based tool-integration in the ETI platform. Software and Systems Modeling **4**(2), 141–156 (2005). https://doi.org/10.1007/s10270-004-0072-z
32. Beyer, D., Wehrheim, H.: Verification artifacts in cooperative verification: Survey and unifying component framework. In: Proc. ISoLA (1). pp. 143–167. LNCS 12476, Springer (2020). https://doi.org/10.1007/978-3-030-61362-4_8
33. Beyer, D., Kanav, S.: CoVeriTeam: On-demand composition of cooperative verification systems. In: Proc. TACAS. pp. 561–579. LNCS 13243, Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_31
34. Beyer, D., Kanav, S., Wachowitz, H.: CoVeriTeam Service: Verification as a service. In: Proc. ICSE, companion. pp. 21–25. IEEE (2023). https://doi.org/10.1109/ICSE-Companion58688.2023.00017
35. Beyer, D.: Conservation and accessibility of tools for formal methods. In: Proc. Festschrift Podelski 65th Birthday. Springer (2024), https://www.sosy-lab.org/research/pub/2024-Podelski65.Conservation_and_Accessibility_of_Tools_for_Formal_Methods.pdf
36. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M.: Correctness witnesses: Exchanging verification results between verifiers. In: Proc. FSE. pp. 326–337. ACM (2016). https://doi.org/10.1145/2950290.2950351

37. Beyer, D.: Software verification and verifiable witnesses (Report on SV-COMP 2015). In: Proc. TACAS. pp. 401–416. LNCS 9035, Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_31
38. Beyer, D.: Competition on software verification and witness validation: SV-COMP 2023. In: Proc. TACAS (2). pp. 495–522. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_29
39. Beyer, D., Strejček, J.: Case study on verification-witness validators: Where we are and where we go. In: Proc. SAS. pp. 160–174. LNCS 13790, Springer (2022). https://doi.org/10.1007/978-3-031-22308-2_8
40. Ayaziová, P., Beyer, D., Lingsch-Rosenfeld, M., Spiessl, M., Strejček, J.: Software verification witnesses 2.0. In: Proc. SPIN. Springer (2024)
41. Beyer, D.: State of the art in software verification and witness validation: SV-COMP 2024. In: Proc. TACAS (3). pp. 299–329. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_15
42. Beyer, D., Friedberger, K.: Violation witnesses and result validation for multi-threaded programs. In: Proc. ISoLA (1). pp. 449–470. LNCS 12476, Springer (2020). https://doi.org/10.1007/978-3-030-61362-4_26
43. Beyer, D., Dangl, M., Lemberger, T., Tautschnig, M.: Tests from witnesses: Execution-based validation of verification results. In: Proc. TAP. pp. 3–23. LNCS 10889, Springer (2018). https://doi.org/10.1007/978-3-319-92994-1_1
44. Beyer, D., Spiessl, M.: MetaVal: Witness validation via verification. In: Proc. CAV. pp. 165–177. LNCS 12225, Springer (2020). https://doi.org/10.1007/978-3-030-53291-8_10
45. J. Švejda, Berger, P., Katoen, J.P.: Interpretation-based violation witness validation for C: NitWit. In: Proc. TACAS. pp. 40–57. LNCS 12078, Springer (2020). https://doi.org/10.1007/978-3-030-45190-5_3
46. Ponce-De-Leon, H., Haas, T., Meyer, R.: Dartagnan: Smt-based violation witness validation (competition contribution). In: Proc. TACAS (2). pp. 418–423. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_24
47. Howar, F., Mues, M.: GWit (competition contribution). In: Proc. TACAS (2). pp. 446–450. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_29
48. Ayaziová, P., Strejček, J.: Symbiotic-Witch 2: More efficient algorithm and witness refutation (competition contribution). In: Proc. TACAS (2). pp. 523–528. LNCS 13994, Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_30
49. Wu, T., Schrammel, P., Cordeiro, L.: Wit4Java: A violation-witness validator for Java verifiers (competition contribution). In: Proc. TACAS (2). pp. 484–489. LNCS 13244, Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_36
50. Bajczi, L., Ádám, Z., Micskei, Z.: ConcurrentWitness2Test: Test-harnessing the power of concurrency (competition contribution). In: Proc. TACAS (3). pp. 330–334. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_16
51. Saan, S., Erhard, J., Schwarz, M., Bozhilov, S., Holter, K., Tilscher, S., Vojdani, V., Seidl, H.: Goblint Validator: Correctness witness validation by abstract interpretation (competition contribution). In: Proc. TACAS (3). pp. 335–340. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_17
52. Beyer, D., Spiessl, M.: LIV: A loop-invariant validation using straight-line programs. In: Proc. ASE. pp. 2074–2077. IEEE (2023). https://doi.org/10.1109/ASE56229.2023.00214
53. Monat, R., Milanese, M., Parolini, F., Boillot, J., Ouadjaout, A., Miné, A.: Mopsa-C: Improved verification for C programs, simple validation of correctness witnesses

(competition contribution). In: Proc. TACAS (3). pp. 387–392. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_26
54. Ayaziová, P., Strejček, J.: Witch 3: Validation of violation witnesses in the witness format 2.0 (competition contribution). In: Proc. TACAS (3). pp. 341–346. LNCS 14572, Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_18
55. Brandes, U., Eiglsperger, M., Herman, I., Himsolt, M., Marshall, M.S.: GraphML progress report. In: Graph Drawing. pp. 501–512. LNCS 2265, Springer (2001). https://doi.org/10.1007/3-540-45848-4_59
56. Turing, A.: Checking a large routine. In: Report on a Conference on High Speed Automatic Calculating Machines. pp. 67–69. Cambridge Univ. Math. Lab. (1949), https://turingarchive.kings.cam.ac.uk/publications-lectures-and-talks-amtb/amt-b-8
57. Baudin, P., Cuoq, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C specification language version 1.17 (2021), available at https://frama-c.com/download/acsl-1.17.pdf
58. Beyer, D., Spiessl, M., Umbricht, S.: Cooperation between automatic and interactive software verifiers. In: Proc. SEFM. p. 111–128. LNCS 13550, Springer (2022). https://doi.org/10.1007/978-3-031-17108-6_7
59. Beyer, D., Podelski, A.: Software model checking: 20 years and beyond. In: Principles of Systems Design. pp. 554–582. LNCS 13660, Springer (2022). https://doi.org/10.1007/978-3-031-22337-2_27