Dartagnan: SMT-based Violation Witness Validation (Competition Contribution)

Hernán Ponce-de-León[1] (✉) *, Thomas Haas[2], and Roland Meyer[2]

[1]Bundeswehr University Munich, Munich, Germany
[2]TU Braunschweig, Braunschweig, Germany
hernan.ponce@unibw.de, t.haas@tu-braunschweig.de, roland.meyer@tu-bs.de

Abstract. The validation of violation witnesses is an important step during software verification. It hides false alarms raised by verifiers from engineers, which in turn helps them concentrate on critical issues and improves the verification experience. Until the 2021 edition of the Competition on Software Verification (SV-COMP), CPAchecker was the only witness validator for the *ConcurrencySafety* category. This article describes how we extended the Dartagnan verifier to support the validation of violation witnesses. The results of the 2022 edition of the competition show that, for witnesses generated by different verifiers, Dartagnan succeeds in the validation of witnesses where CPAchecker does not. Our extension thus improves the validation possibilities for the overall competition. We discuss Dartagnan's strengths and weaknesses as a validation tool and describe possible ways to improve it in the future.

1 Introduction

Most software verification tools report witnesses to property violations. Since SV-COMP 2015, there is a common format in which witnesses are represented by automata [4]. Each edge of such an automaton is annotated with data that can be used to match program executions. A data annotation can be, e.g., *"assumption"* specifying constraints on values of variables in a given state, *"control"* specifying the outcome of a branch condition, or *"startline"* specifying a concrete line in the source code. More details about data annotations and their semantics can be found in the exchange format documentation [1].

A witness validator checks that a violation can be reproduced using the information provided by the witness. Automata-based verifiers can easily be converted into validators by analyzing the synchronized product of the program with the witness automaton. In this setting the witness automaton guides the verifier. If none of the outgoing edges on the program state match the next edge of the witness automaton, then the verifier cannot explore the current path further. If the edge on the program state matches, then the witness automaton and the program proceed to the next state, eventually leading to a violation.

* Jury member.

While this idea allows one to easily convert any automata-based verifier into a validator, not all verifiers are automata-based.

Dartagnan is an SMT-based verifier. In the next section, we explain how to convert it into a validator. The idea is to extract information from the witness and use it to reduce the search space explored by the backend SMT solver.

2 Validation Approach

Given a concurrent program and a specification in the form of assertions, Dartagnan generates an SMT formula $\varphi_{\text{Ver}} = \varphi_{\text{Cf}} \wedge \varphi_{\text{Df}} \wedge \varphi_{\text{Sc}} \wedge \varphi_{\text{🐞}}$ which is satisfiable if and only if some assertion fails [16,15]. The formulas φ_{Cf} and φ_{Df} encode (respectively) the control flow and the data flow of the program. Formula φ_{Sc} encodes scheduling constraints. Finally, $\varphi_{\text{🐞}}$ expresses that at least one assertion must fail. If the formula is satisfiable, then a violation exists. The goal of Dartagnan (as a verifier) is to find such a violation. This amounts to finding an appropriate scheduling among the threads. Such a scheduling is encoded as a *happens-before relation* between the instructions. Dartagnan thus searches the space of all viable happens-before relations to find a violation or prove that none exists.

We now explain how to extend Dartagnan into a violation witness validator. The idea is to extract from the violation witness a formula $\varphi_{\text{👁}}$ that we conjoin to the rest of Dartagnan's encoding, resulting in $\varphi_{\text{Val}} = \varphi_{\text{Ver}} \wedge \varphi_{\text{👁}}$. The extra constraints in $\varphi_{\text{👁}}$ reduce the search space for the SMT solver. For the verification of concurrent programs taking inputs from the environment, there are two sources of non-determinism: the *data* coming from the input (which might influence the control flow) and the *scheduling*. The purpose of $\varphi_{\text{👁}}$ is to reduce this non-determinism. Extending the SMT encoding as described in φ_{Val} is conceptually easy. The interesting question is *"what information from the witness shall we use?"* The less information we use, the more we move from pure validation to full verification.

While automata-based validators can use some information in a straightforward manner, this is not the case for Dartagnan.

1. A violation witness can contain cycles to represent infinitely many executions. However, SMT-based tools unroll cycles and perform bounded verification, thus only part of this information is helpful.
2. Since Dartagnan (as many other BMC tools) does not keep an explicit notion of state, using state information is not trivial.

The exchange format for violation witnesses allows for expressing information about state assumptions, the control flow, and the scheduling. We abstract out from the former two and only use scheduling information. We assume that witness automata represent a single path and that the edges contain *"startline"* data corresponding to read or write instructions[1]. Those are the only instructions

[1] Our validator accepts witnesses that do not satisfy the second assumption, but it filters out the corresponding edges.

that can affect our happens-before relation. While we do not explicitly encode the outcome of control-flow instructions, certain control-flow information is implicitly encoded based on which instructions are executed. We explain the reason behind these design decisions and assumptions, discuss its limitations, and describe how we plan to improve this in the future in Section 3. Despite these limitations, and as we show in Section 4, our validator performs well in practice.

Let (S, E) be a witness automaton with states S and edges E. For each $e \in E$, function `e2i(e)` returns the set of read or write instructions coming from the *"startline"* in the C file that corresponds to the given edge. Since witnesses represent single paths, they can be seen as a word over S. Let $w \in S^*$ be a witness, we define the *witness-to-formula* function which constructs $\varphi_{\faEye}$ as

$$\texttt{w2f}(w) = \begin{cases} \mathit{true} & \text{if } w = \epsilon \\ \texttt{w2f}(w') \wedge \bigvee\limits_{\substack{i_1 \in \texttt{e2i}((\_,s)) \\ i_2 \in \texttt{e2i}((s,\_))}} \textsf{happens-before}(i_1, i_2) & \text{if } w = s \cdot w' \end{cases}$$

3 Strengths and Weaknesses

The main strengths of our validation approach are simplicity and modularity. The approach just requires to add a new sub-formula to the SMT encoding used for verification. The validator is modular in the sense that using more or different information from the witness does not change the validation approach. For example, adding information from the witness about the control flow just requires adding more constraints to $\varphi_{\faEye}$.

Our validation approach assumes that witness automata represent single paths. This is a limitation not imposed by the exchange format. However, verifiers tend to stop as soon as they find one violation and thus generate witnesses representing a single violation path. A second limitation is that we do not explicitly consider control-flow information. This might impact the performance of the validation since not all non-determinism is removed and the search space might still be large. Converting such control-flow information into SMT is simple in principle. However, since Dartagnan internally converts the C program into Boogie [14], matching conditionals with the corresponding assembly-like jumps requires some work. A second consequence of not extracting control-flow information from the witness is that we might validate witnesses that do not lead to a violation. This is because we over-approximate the paths of the program represented by the witness and thus our approximation might include the path leading to the violation even if the witness did not.

4 Validation Results

We inspected the results of SV-COMP 2022 [5] to answer the following questions

RQ1: What percentage of the witnesses can Dartagnan validate?
RQ2: What percentage can Dartagnan not validate and why?

RQ3: Can Dartagnan validate witnesses that CPAchecker cannot?
RQ4: Can CPAchecker validate witnesses that Dartagnan cannot?

From the 20 verifiers in *ConcurrencySafety*, we selected five tools implementing different verification approaches. We consider them good representatives of the whole category: *(i)* **CBMC** [12] (used as a backend by Deagle [8] and Lazy-CSeq [10]), *(ii)* **CPAchecker** [6] (used as a backend by CPA-Lockator [3] and Graves [13]), *(iii)* **EBF** [2] (combines BMC with fuzzing, a very effective technique to find bugs), *(iv)* **Dartagnan** [16] (only tool where the memory model, here sequential consistency, is taken as an input), and *(v)* **GemCutter** [11] (shares the codebase with UTaipan [7] and UAutomizer [9]).

Table 1 presents the results of the validation in SV-COMP 2022. We report the number of witnesses generated by each verifier ("Witnesses"). For each of the validators (columns "Dartagnan" and "CPAchecker"), we report the number of cases where the validation conclusively finished (i.e., it returned True or False), whether the violation was confirmed (left of "/") or not (right of "/"), and the number of correct validations by one tool where the other did not report a result (columns "Dart \ CPA" and "CPA \ Dart", respectively).

Tool	Witnesses	Dartagnan	CPAchecker	Dart \ CPA	CPA \ Dart
CBMC	305	193/0	95/0	117	19
CPAchecker	256	0/0	256/0	0	256
Dartagnan	273	245/1	35/6	204	0
EBF	290	219/0	57/0	177	15
GemCutter	299	18/237	262/1	15	28

Table 1. Results of the validation in SV-COMP 2022.

For the SMT-based verifiers CBMC and EBF, Dartagnan has 63.28% resp. 75.52% success rate in the validation (against 31.15% resp. 19.66% success rate for CPAchecker). Unfortunately, it did not validate any of the witnesses generated by CPAchecker. This was due to a bug in the witness parser that has been identified and fixed after the competition. CPAchecker validated all the witnesses that it generated as a verifier. Dartagnan validated 89.74% of its own witnesses while CPAchecker only validated 12.82%. For GemCutter, the validation success of Dartagnan is only 6.02%. This is because, due to another bug, it wrongly marked 237 witnesses as not validated. The fixed version of Dartagnan is able to validate all such cases. Despite this, from the 18 witnesses that Dartagnan validated, 15 of them were not validated by CPAchecker, thus improving the validation possibilities for the overall competition.

5 Software Project and Configuration

The project home page is https://github.com/hernanponcedeleon/Dat3M. To run Dartagnan as a validator, use the following command:

```
$ Dartagnan-SVCOMP.sh -witness <witness> <property> <program>
```

References

1. Exchange Format for Violation Witnesses and Correctness Witnesses. https://github.com/sosy-lab/sv-witnesses.
2. Fatimah Aljaafari, Lucas C. Cordeiro, Mustafa A. Mustafa, and Rafael Menezes. EBF: A hybrid verification tool for finding software vulnerabilities in iot cryptographic protocols. *CoRR*, abs/2103.11363, 2021.
3. Pavel S. Andrianov, Vadim S. Mutilin, and Alexey V. Khoroshilov. cpalockator: Thread-modular analysis with projections - (Competition Contribution). In *TACAS (2)*, volume 12652 of *Lecture Notes in Computer Science*, pages 423–427. Springer, 2021. doi:10.1007/978-3-030-72013-1_25.
4. Dirk Beyer. Software verification and verifiable witnesses - (report on SV-COMP 2015). In *TACAS*, volume 9035 of *Lecture Notes in Computer Science*, pages 401–416. Springer, 2015. doi:10.1007/978-3-662-46681-0_31.
5. Dirk Beyer. Progress on software verification: SV-COMP 2022. In *TACAS (2)*. Springer, 2022.
6. Dirk Beyer and M. Erkan Keremoglu. CPAchecker: A tool for configurable software verification. In *CAV*, volume 6806 of *Lecture Notes in Computer Science*, pages 184–190. Springer, 2011. doi:10.1007/978-3-642-22110-1_16.
7. Daniel Dietsch, Matthias Heizmann, Alexander Nutz, Claus Schätzle, and Frank Schüssele. Ultimate Taipan with symbolic interpretation and fluid abstractions - (Competition Contribution). In *TACAS (2)*, volume 12079 of *Lecture Notes in Computer Science*, pages 418–422. Springer, 2020. doi:10.1007/978-3-030-45237-7_32.
8. Fei He, Zhihang Sun, and Hongyu Fan. Deagle: An SMT-based verifier for multi-threaded programs (Competition Contribution). In *TACAS (2)*. Springer, 2022.
9. Matthias Heizmann, Yu-Fang Chen, Daniel Dietsch, Marius Greitschus, Jochen Hoenicke, Yong Li, Alexander Nutz, Betim Musa, Christian Schilling, Tanja Schindler, and Andreas Podelski. Ultimate Automizer and the search for perfect interpolants - (Competition Contribution). In *TACAS (2)*, volume 10806 of *Lecture Notes in Computer Science*, pages 447–451. Springer, 2018. doi:10.1007/978-3-319-89963-3_30.
10. Omar Inverso, Ermenegildo Tomasco, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Lazy-CSeq: A lazy sequentialization tool for C - (Competition Contribution). In *TACAS*, volume 8413 of *Lecture Notes in Computer Science*, pages 398–401. Springer, 2014. doi:10.1007/978-3-642-36742-7_46.
11. Dominik Klumpp, Daniel Dietsch, Matthias Heizmann, Frank Schüssele, Marcel Ebbinghaus, Azadeh Farzan, and Andreas Podelski. Ultimate GemCutter and the axes of generalization (Competition Contribution). In *TACAS (2)*. Springer, 2022.
12. Daniel Kroening and Michael Tautschnig. CBMC - C bounded model checker - (Competition Contribution). In *TACAS*, volume 8413 of *Lecture Notes in Computer Science*, pages 389–391. Springer, 2014. doi:10.1007/978-3-642-54862-8_26.
13. William Leeson and Matthew Dwyer. GraVeS: Graph-based verifier selector (Competition Contribution). In *TACAS (2)*. Springer, 2022.
14. K. Rustan M. Leino. This is Boogie 2. 2008. URL: https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/.
15. Hernán Ponce de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. Portability analysis for weak memory models. PORTHOS: One tool for all models. In *SAS*, volume 10422 of *LNCS*, pages 299–320. Springer, 2017. doi:10.1007/978-3-319-66706-5_15.

16. Hernán Ponce de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. Dartagnan: Bounded model checking for weak memory models (Competition Contribution). In *TACAS (2)*, volume 12079 of *LNCS*, pages 378–382. Springer, 2020. doi:10.1007/978-3-030-45237-7_24.