GWIT: A Witness Validator for Java based on GraalVM (Competition Contribution)*

Falk Howar (✉)[1,2] and Malte Mues[1]

[1] TU Dortmund University, Dortmund, Germany
{falk.howar, malte.mues}@tu-dortmund.de
[2] Fraunhofer ISST, Dortmund Germanny

Abstract. GWIT is a validator for violation witnesses produced by Java verifiers in the SV-COMP software verification competition. GWIT weaves assumptions documented in a witness into the source code of a program, effectively restricting the part of the program that is explored by a program analysis. It then uses the GDart tool (dynamic symbolic execution) to search for reachable errors in the modified program.

1 Introduction

Software verification tools, like any other software, can contain bugs. Given their intended use, i.e., proving the absence of errors in programs, however, bugs in verification tools are particularly problematic. On the other hand, verification tools can generate certificates for computed verdicts (e.g., counterexamples) that can be used to validate verification results. In the SV-COMP competition on software verification *violation witnesses* and *correctness witnesses*, based on annotated abstract control-flow automata have been established as a standardized representation of such certificates [1, 2]. Participating verifiers are expected to produce witnesses for verdicts and *witness validators* are used for confirming verdicts based on these witnesses.

In this paper, we present GWIT (as in "**G**uess **W**hat **I**'m **T**hinking" or as in **G**Dart-based **wit**ness validator), a validator of violation witnesses for Java programs, based on the GDart tool ensemble [6]. GWIT validates violation witnesses by weaving the assumptions documented in a witness into the original program under analysis and checks the restricted program with dynamic symbolic execution.

2 Witness Validation in GWIT

We illustrate the operation of GWIT for the small example shown in Figure 1: In the program, a String value is created nondeterministically before asserting that the value of this String value should not be "whoopsy". This program contains a reachable error: in case the value "whoopsy" is returned by the call to `Verifier.nondetString()`, an assertion violation will be triggered.

* This work has been partially founded by an Amazon Research Award

```
1  public static void main(String[] args) {
2    String s = Verifier.nondetString();
3    assert !s.equals("whoopsy")
4  }
```

Fig. 1: Small program with reachable error.

Java verifiers will generate a violation witness in such a case. In SV-COMP, witnesses are produced in a standardized format, conceptually based on control-flow automata and technically realized as models in the *GraphML* format [2]. Figure 2 shows an excerpt of such a witness for the above example. The witness makes an assumption on the state of the program when executing line 2 of the example program, namely that variable `s` has value "whoopsy". As discussed, execution paths on which this assumption holds, will lead to an error.

GWIT weaves the assumptions from the witness into the original program, restricting the number of program paths that have to be explored for finding the error. Figure 3 shows the result for our example: a call to `Witness.assume(...)` is generated from the assumption from the witness in Figure 2. The assume method wraps potentially many calls to the `Verifier.assume(...)` method, enabling multiple assumptions on the same line of code (e.g., due to execution of that line in a loop). The `counters` array keeps statistic on assumptions per line. The `Verifier.assume(...)` method is used by GDart to stop analysis on paths that violate the corresponding assumption.

Figure 4, finally, shows the effect of weaving the witness into the code on the obtained constraints-trees. In the left of the figure, the tree computed by GDart for the original program is shown. The tree has two satisfiable paths, branching on the condition of the assert statement. The right of the figure shows the tree for the modified program. This tree contains a node for the assumption, one path that is not executed after the violation of the assumption, one path that is not feasible after the assumption for the assert statement, and one path leading to an error (i.e., assertion violation). In this small example, the tree for the modified program is more complex than the tree for the original program, but it has fewer complete execution paths. In more complex programs, assumptions will typically remove multiple execution paths, making the validation task significantly easier than the original verification task.

```
<edge source="n0" target="n1">
  <data key="originfile">Main.java</data>
  <data key="startline">2</data>
  <data key="threadId">0</data>
  <data key="assumption">s.equals("whoopsy")</data>
  <data key="assumption.scope">...</data>
</edge>
```

Fig. 2: Excerpt of violation witness produced by GDart or JBMC.

```
1  static int[] counters = new int[] { 0 };
2  public static void assume(int id, boolean ... assumptions) {
3    int idx = counters[id];
4    counters[id]++;
5    Verifier.assume(assumptions[idx]);
6  }
7
8  public static void main(String[] args) {
9    String s = Verifier.nondetString();
10   Witness.assume(0, s.equals("whoopsy"));
11   assert !s.equals("whoopsy")
12 }
```

Fig. 3: Program with assumption from witness weaved into the code.

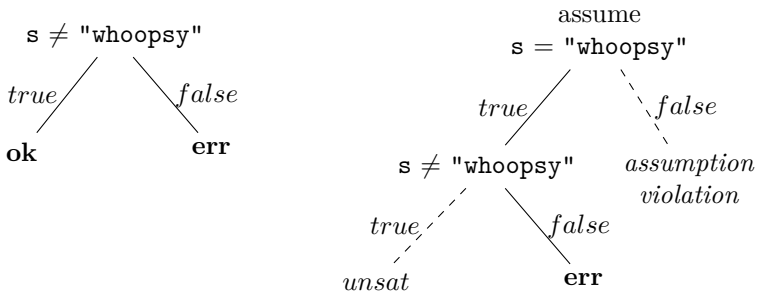

Fig. 4: Constraints-tree for original program (left) and modified program (right).

3 Performance and Limitations

While the approach of GWIT is sound for violation witnesses, the current implementation still has limitations, validating roughly half of the witnesses provided by verifiers.

Soundness. GWIT is sound: weaving a witness into the code adds additional decision nodes to the constraints-tree. In the sub-tree rooted at such a new node, some paths become unsatisfiable and will not be explored. Every complete path ψ in the modified tree has an equivalent path ϕ in the original constraints-tree such that $\psi \implies \phi$. If an error is reached in the modified tree, it is also reachable in the original program.

Performance. For programs with few decisions, the modified program may actually be more complex than the original program, but GDart does only explore more paths than in the original program in cases where the initial value along some path does not satisfy an assumption. Comparing the CPU times of GDart used as a verifier and used through GWIT, using almost identical configuration

options (only difference: GWIT does not produce witnesses), complexity is reduced for most benchmark instances that do not fail due to syntactic errors during weaving (see below).

Two extreme examples are the `BellmanFord-FunSat02` for which weaving a witness with 13 assumptions increases CPU time more than twice, leading to a timeout during validation and the `nanoxml_eqchk/prop2` instance for which the CPU time required for validation is less than 14% of the CPU time needed for the original verification task.

Overall, GWIT successfully validates 301 of 614 witnesses provided by GDart and JBMC [3] (the only Java verifiers that currently produce witnesses). In 286 cases, validation failed with inconclusive verdicts due to currently unsupported features of witness. In 15 cases, incorrect weaving (see below) prevented validation of witnesses. For 12 witnesses, validation attempts exhaust resource limits.

Limitations. First, GWIT currently only supports violation witnesses. In principle, it should be possible to validate verification witnesses by weaving assertions into the program code, but it is not obvious that such an approach makes the validation of witnesses a simpler problem than the original verification task. Second, since weaving witnesses is done on the source code, it only works correctly on proper blocks, delimited with braces, and with one statement per line. While this does not affect soundness, it makes the validation of witnesses impossible in some cases.

4 Tool Setup

GWIT is shipped as a git repository with sub-projects delivering all required components. Checking out the repository and initializing all sub-projects pulls in all required source code. For building the SPouT component, the mx build system[3] maintained by the GraalVM [7] team is required. Other components are built with maven. Once all build systems are available, the `./compile-all.sh` script builds GWIT. The `./run-gwit.sh` is used to validate witnesses, taking the witness file and source folders of a benchmark instance as parameters. GWIT currently does not expose any other configuration parameters.

5 Software Project

The GWIT tool is available on GitHub[4]. GWIT's scripts are licensed under the Apache 2.0 license. The sub-project bring their own license as follows: DSE[5] is available under the Apache 2.0 license, JConstraints[6] [4] as well, and SPouT[7] is available under the GPL v2 license. The components of GWIT and GWIT itself are currently developed at TU Dortmund by the group led by Falk Howar.

[3] https://github.com/graalvm/mx
[4] https://github.com/tudo-aqua/gwit
[5] https://github.com/tudo-aqua/dse
[6] https://github.com/tudo-aqua/jconstraints
[7] https://github.com/tudo-aqua/spout

6 Data Availability Statement

The GWIT archive used for SV-COMP 2022 is available at Zenoodo [5].

References

1. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M.: Correctness witnesses: Exchanging verification results between verifiers. In: Proc. FSE. p. 326337. FSE 2016, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2950290.2950351
2. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Stahlbauer, A.: Witness validation and stepwise testification across software verifiers. In: Proc. FSE. p. 721733. ESEC/FSE 2015, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2786805.2786867
3. Cordeiro, L., Kroening, D., Schrammel, P.: JBMC: Bounded model checking for Java bytecode. In: Beyer, D., Huisman, M., Kordon, F., Steffen, B. (eds.) Proc. TACAS. pp. 219–223. Springer International Publishing, Cham (2019). https://doi.org/10.1007/978-3-030-17502-3_17
4. Howar, F., Jabbour, F., Mues, M.: JConstraints: A library for working with logic expressions in Java. In: Models, Mindsets, Meta: The What, the How, and the Why Not?, pp. 310–325. Springer (2019). https://doi.org/10.1007/978-3-030-22348-9_19
5. Howar, F., Mues, M.: Gwit artifact for sv-comp 2022 (Feb 2022). https://doi.org/10.5281/zenodo.5956885
6. Mues, M., Howar, F.: GDart: An ensemble of tools for dynamic symbolic execution on the java virtual machine (competition contribution). In: Proc. TACAS (2). Springer (2022)
7. Würthinger, T., Wimmer, C., Wöß, A., Stadler, L., Duboscq, G., Humer, C., Richards, G., Simon, D., Wolczko, M.: One VM to rule them all. In: Proc. SPLASH. pp. 187–204 (2013)