Wit4Java: A Violation-Witness Validator for Java Verifiers (Competition Contribution)

Tong Wu[1], Peter Schrammel[2], and Lucas C. Cordeiro[1](✉)

[1] University of Manchester, Manchester, United Kingdom
[2] University of Sussex, Brighton, and Diffblue Ltd, Oxford, United Kingdom
lucas.cordeiro@manchester.ac.uk

Abstract. We describe and evaluate a violation-witness validator for Java verifiers called Wit4Java. It takes a Java program with a safety property and the respective violation-witness output by a Java verifier to generate a new Java program whose execution deterministically violates the property. We extract the value of the program variables from the counterexample represented by the violation-witness and feed this information back into the original program. In addition, we have two implementations for instantiating source programs by injecting counterexamples. Experimental results show that Wit4Java can correctly validate the violation-witnesses produced by JBMC and GDart in a few seconds.

Keywords: Witness Validation · Software Verification · Java Bytecode.

1 Overview

Witness validation is the process of checking whether the same results can be reproduced independently according to the given program, specification, verification result, and the generated witness, improving the trust level of the software verifiers [2].

Here, we describe and evaluate a new violation-witness validator for Java programs called Wit4Java. We take an approach similar to Rocha et al. [5] and Beyer et al. [1] for C programs and apply it to Java programs. As a result, we implement Wit4Java as a Python script that creates a new Java program or a unit test case using Mockito with the program variable values extracted from the counterexample. As input, Wit4Java uses the violation-witness in the GraphML format to extract the value of the non-deterministic variables in Java programs. Lastly, Wit4Java runs the new created program using the Java Virtual Machine (JVM) to check the *assert* statements.

There are some validators for C programs in the literature [6,12]. For example, NitWit is an interpretation-based witness validator that can execute each statement step-by-step without compiling the entire program [12]. The concept of MetaVal is to generate a new program based on the input and then use any checker to check for specifications [6]. CPA-witness2test and FShell-witness2test are execution-based validators for C programs that can process the witness in

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13244, pp. 484–489, 2022.
https://doi.org/10.1007/978-3-030-99527-0_36

GraphML format and generate a test harness that drives the program to the specification violation [1]. Rocha et al. focus on the counterexample produced by ESBMC [4] while CPA-witness2test and FShell-witness2test can process GraphML files. However, witness validation for SV-COMP's Java track [7] is still at an early stage. GWIT is another validator that uses assumptions to prune the search space for dynamic symbolic execution, limiting the analysis to paths where a given assumption holds [10,11].

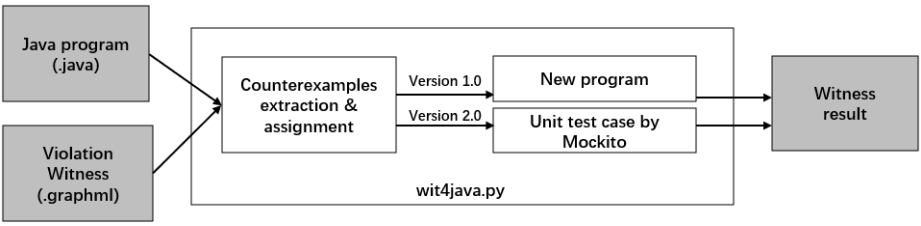

Fig. 1. Wit4Java Architecture. The grey boxes represent the inputs and outputs, and the white boxes represent the validation process.

2 Validation Approach

The architecture of Wit4Java is illustrated in Fig. 1. First, Wit4Java takes the Java program and the witness as input. Then, it uses the Python package NetworkX to read the graph content of the witness and extracts the counterexample values of the variables corresponding to the source program from the violation-witness and saves them. After that, it generates new programs that contain the witness's assumptions. Finally, the validation process is performed by the JVM (using the `-ea` option) to check whether the execution of the generated program exhibits the detected assertion failure.

There are two implementations (Wit4Java 1.0 and Wit4Java 2.0) to extract and use counterexamples. The first version is to save them as tuples (*linenum*, *counterexample*). Then it reads the source program and replaces the variables of the program statements with counterexamples if the line number and variable in the program match the tuple, thus generating a new created Java program. In comparison, the second version records the data types and values of the counterexamples and saves them sequentially into two lists. Moreover, only the assumptions made in the witness for the non-deterministic variables (as determined by Verifier.nondet) are recorded. Then, it builds a unit test case and employs the Mockito framework to mock the Verifier.nondet calls in the source program to make them return deterministic counterexample values from the lists. This makes the execution of the source program follow the path described in the witness and eventually reach the violated property.

Listing 1.1. Analyzed program

```
int v1 = Verifier.nondetInt();
int v2 = Verifier.nondetInt();
assert v1 == v2;
```

Listing 1.2. Output of Wit4Java 1.0

```
int v1 = 1;
int v2 = 0;
assert v1 == v2;
```

We show examples for both implementations in Listings 1.1 to 1.4. Wit4Java 1.0 (the naive version) saves the counterexamples in witness in line number order. It directly replaces the variable values in the source program, thus generating a new program (cf. Listing 1.2). Wit4Java 2.0 (We name it the Mockito version) generates a test case that returns the counterexample value when the mocked function is called (cf. Listing 1.4).

Listing 1.3. Violation witness

```
<edge source="203.167" target="207.186">
      <data key="originfile">
      Main.java
      </data>
      <data key="startline">
      13
      </data>
      <data key="assumption">
      v1 = 1;
      </data>
</edge>
 <edge source="207.186" target="252.201">
      <data key="originfile">
      Main.java
      </data>
      <data key="startline">
      14
      </data>
      <data key="assumption">
      v2 = 0;
      </data>
 </edge>
```

Listing 1.4. Output of Wit4Java 2.0

```
List_type = [int, int];
List_value = [1, 0];
Mockito.mockStatic(Verifier.class);
int n = List_type.length;
OngoingStubbing <Integer>
  stubbing_int = Mockito.
    when(Verifier.nondetInt());
for (int i = 0; i < n; i++) {
 if ("int".equals(List_type[i])) {
   stubbing_int = stubbing_int.
     thenReturn(Integer.
       parseInt(List_value[i]));
 }
}
Main.main(new String[0]);
```

3 Discussion of Strengths and Weaknesses

Fig. 2 on the left compares the validation results of the two validation tools Wit4Java and GWIT. The former is based on version 1.0 (naive version). The latter is based on violation-witnesses produced by GDart. The results indicate that Wit4Java has successfully validated 140 out of 302 witnesses, while GWIT correctly validates 150 results. Version 2.0 handles counterexamples with different values for each iteration within a loop better than version 1.0. This is because version 1.0 skips the counterexamples before the last iteration. However, version 2.0 can fully use the counterexamples generated by each iteration. Fig. 2 on the right compares the validation results of the two versions of Wit4Java, which shows that version 2.0 (Mockito version) has a better validation ability (168 out of 302), thereby outperforming both version 1.0 and GWIT. However, the tool can only handle witnesses with concrete counterexamples. There are two main reasons why Wit4Java shows the result *unknown*: JBMC [3,8] produces an empty witness, or the witness does not contain a counterexample for a non-deterministic value. Besides, the validation for strings is not supported yet, which occurs in almost half of witnesses because JBMC does not yet output counterexample values for strings. Thus we were not able to test it. Generally, there are not enough

witnesses of high quality for testing the witness validator yet because JBMC sometimes correctly terminates without producing a witness in SV-COMP. The witness support in the Java verifiers requires further development work so that they are able to produce complete violation witnesses whenever they terminate with verdict false.

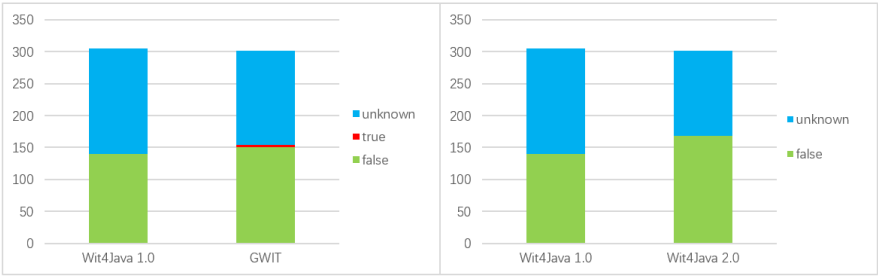

Fig. 2. Validation results based on 302 witnesses. The x-axis represents the names of the two tools and the y-axis represents the number of witnesses. A green "false" indicates a confirmed correct result.

4 Tool Setup and Configuration

The competition submission is based on Wit4Java version 1.0 (naive version).[3] For the competition [9], Wit4Java is called by executing the script wit4java.py. It reads .java source files and corresponding witnesses in the given benchmark directories. The answer would be *false* if the assertion failure was found. As an example, we can validate the witness by executing the following command:

./wit4java.py -witness <path-to-sv-witnesses>/witness.graphml <path-to-sv-benchmarks>/java/jbmc-regression/return2

where witness.graphml indicates the witness to be validated, and return2 indicates the benchmark name. The Benchexec tool info module is called wit4java.py and the benchmark definition file wit4java-validate-violation-witnesses.xml. NetworkX should be installed separately in the SV-COMP machines. If a validation task does not find a property violation, it will return *unknown*.

5 Software Project and Contributors

Tong Wu maintains Wit4Java. It is publicly available under a BSD-style license. The source code is available at https://github.com/Anthonysdu/wit4java, and instructions for running the tool are given in the README file.

[3] https://github.com/Anthonysdu/wit4java

Acknowledgment

The work in this paper is partially funded by the EPSRC grants EP/T026995/1, EP/V000497/1, EU H2020 ELEGANT 957286, and Soteria project awarded by the UK Research and Innovation for the Digital Security by Design (DSbD) Programme.

References

1. Beyer et al. “Tests from Witnesses - Execution-Based Validation of Verification Results”. In: Tests and Proofs - 12th International Conference, TAP@STAF. Vol. 10889. Lecture Notes in Computer Science. 2018, pp. 3–23. https://doi.org/10.1007/978-3-319-92994-1_1.
2. Beyer et al. “Witness validation and stepwise testification across software veri- fiers”. In: ESEC/FSE. 2015, pp. 721–733. https://doi.org/10.1145/2786805.2786867.
3. Cordeiro et al. “JBMC: A Bounded Model Checking Tool for Verifying Java Bytecode”. In: CAV. Vol. 10981. LNCS. 2018, pp. 183–190. https://doi.org/10.1007/978-3-319-96145-3_10.
4. Gadelha et al. “ESBMC v6.0: Verifying C Programs Using k-Induction and Invariant Inference - (Competition Contribution)”. In: TACAS: vol. 11429. LNCS. 2019, pp. 209–213. https://doi.org/10.1007/978-3-030-17502-3_15.
5. Rocha et al. “Understanding Programming Bugs in ANSI-C Software Using Bounded Model Checking Counter-Examples”. In: IFM. Vol. 7321. LNCS. 2012, pp. 128–142. https://doi.org/10.1007/978-3-642-30729-4_10.
6. Dirk Beyer and Martin Spiessl. “MetaVal: Witness Validation via Verification”. In: CAV Part II. Vol. 12225. LNCS. 2020, pp. 165–177. https://doi.org/10.1007/978-3-030-53291-8_10.
7. Lucas C. Cordeiro, Daniel Kroening, and Peter Schrammel. “Benchmarking of Java Verification Tools at the Software Verification Competition (SV-COMP)”. In: ACM SIGSOFT Softw. Eng. Notes 43.4 (2018), p. 56. https://doi.org/10.1145/3282517.3282529.
8. Lucas C. Cordeiro, Daniel Kroening, and Peter Schrammel. “JBMC: Bounded Model Checking for Java Bytecode - (Competition Contribution)”. In: TACAS. Vol. 11429. LNCS. 2019, pp. 219–223. https://doi.org/10.1007/978-3-030-17502-3_17.
9. D. Beyer. “Progress on Software Verification: SV-COMP 2022”. In: Proc. TACAS. Springer, 2022.
10. Falk Howar and Malte Mues. “GWIT (Competition Contribution)”. In: Proc. TACAS (2). Springer, 2022.
11. Falk Howar and Malte Mues. Tudo-Aqua/gwit. https://github.com/tudo-aqua/gwit.
12. Jan Svejda, Philipp Berger, and Joost-Pieter Katoen. “Interpretation-Based Violation Witness Validation for C: NITWIT”. In: TACAS. Vol. 12078. LNCS. 2020, pp. 40–57. https://doi.org/10.1007/978-3-030-45190-5_3.