JCWIT: A Correctness-Witness Validator for Java Programs based on Bounded Model Checking

Zaiyu Cheng
University of Manchester
Manchester, United Kindom
zaiyucheng.cs@gmail.com

Tong Wu
University of Manchester
Manchester, United Kindom
wutonguom@gmail.com

Peter Schrammel
University of Sussex
Sussex, United Kindom
ps372@sussex.ac.uk

Norbert Tihanyi
Eotvos Lorand University
Hungary
ntihanyi@inf.elte.hu

Eddie B. de Lima Filho
TPV Technology
Brazil
eddie.filho@tpv-tech.com

Lucas C. Cordeiro
University of Manchester
Manchester, United Kindom
lucas.cordeiro@manchester.ac.uk

ABSTRACT

Witness validation is a formal verification method to independently verify software verification tool results, with two main categories: *violation* and *correctness* witness validators. Validators for violation witnesses in Java include *Wit4Java* and *GWIT*, but no dedicated correctness witness validators exist. To address this gap, this paper presents the Java Correctness-Witness Validator (JCWIT), the first tool to validate correctness witnesses in Java programs. JCWIT accepts an original program, a specification, and a correctness witness as inputs. Then, it uses invariants of each witness's execution state as conditions to be incorporated into the original program in the form of assertions, thus instrumenting it. Next, JCWIT employs an established tool, Java Bounded Model Checker (JBMC), to verify the transformed program, hence examining the reproducibility of correct witness results. We evaluated JCWIT in the SV-COMP *ReachSafety* benchmark, and the results show that JCWIT can correctly validate the correctness witnesses generated by Java verifiers.

CCS CONCEPTS

• **Software and its engineering → Formal software verification**.

KEYWORDS

Correctness, Witness Validation, Java Programming, Bounded Model Checking.

ACM Reference Format:
Zaiyu Cheng, Tong Wu, Peter Schrammel, Norbert Tihanyi, Eddie B. de Lima Filho, and Lucas C. Cordeiro. 2024. JCWIT: A Correctness-Witness Validator for Java Programs based on Bounded Model Checking. In *Proceedings of the 33nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24).* ACM, New York, NY, USA, 5 pages. https://doi.org/XXXXXXX.XXXXXXX

ISSTA '24, September 16–20, 2024, Vienna, Austria

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Ensuring software reliability and integrity in software development is paramount [1]. As software systems become increasingly complex and multifaceted, robust verification techniques turn into imperative assets [3, 7, 8, 19]. Consequently, formal software verification emerged as a powerful methodology aiming to ensure correctness for software systems through rigorous mathematical reasoning and logical inference [15]. Indeed, it has evolved into a cornerstone approach for addressing the challenges of software complexity and functional requirements.

However, the formal software verification work is usually performed by untrusted verification engines or static analysis tools with the risk of false positives [11]. As a response, witness validation has therefore emerged as a technique aiming to independently check the accuracy and reproducibility of results provided by software verification tools [1].

Witness validation uses a combination of programs, specifications, verification results, and a generated witness [7] to create a piece of evidence, during verification processes, capable of validating outcome veracity [17]. Its objective is to improve the credibility of software verifiers by ensuring verification result precision and replicability by third-party entities [6].

Witness validators encompass various methodologies and are usually categorized into two main classes: violation witness validators and correctness witness validators [8]. The former focuses on validating instances where a software program fails to meet its specifications, using a witness demonstrating the associated violations [18]. In contrast, the latter confirms that a program adheres to its specifications, with a witness as evidence of correct behavior.

There are several validators for violation witnesses in Java programs, such as Wit4Java [20] and the GDart-based witness validator (GWIT) [16]. However, we found no validator dedicated to correctness witnesses, leaving a void in this area. To address this gap, we propose the Java Correctness-Witness Validator (JCWIT), the first tool to validate correctness witnesses within Java programs. It is equipped with a comprehensive framework, accepting input programs, specifications, and a correctness witness as parameters, where any deviations prompt it to report potential errors.

Drawing inspiration from the existing methodology *validation via verification* [9], the core concept revolves around asserting invariants extracted from witnesses and seamlessly integrating them into original programs. Subsequently, using the Java Bounded

Model Checker (JBMC) [12–14], a tool based on bounded model checking (BMC) [10] for Java programs, conducts a rigorous re-verification of the transformed programs, ensuring both adherence to the original specifications and the validity of assertions.

We have evaluated JCWIT using the dataset SV-COMP *ReachSafety* benchmark [1]. The output of the validation result manifested in three possible scenarios: successful validation, indicating that all assertions held and adhered to the original specifications; validation failure, indicating discrepancies between the transformed program and original verification results; and unknown validation, signaling either the inability to validate or unknown validation outcomes.

2 RELATED WORK

The primary tools used to validate violation witnesses for Java programs are GWIT [16] and Wit4Java [20]. Both extract counterexamples from a witness and reweave these assumptions into an original program. This modification ensures that a program's execution follows the execution path described by a witness, ultimately reaching the violated property. In SV-COMP 2022 [2], Wit4Java and GWIT successfully validated 140 and 150 out of 302 violation witnesses, respectively, for the category *ReachSafety* in Java programs.

MetaVal [9], another tool for witness validation, tackles correctness and violation witnesses for C programs. In the case of a violation witness, a program is adapted by pruning segments deemed irrelevant to its counterexample path as indicated by a witness. Conversely, to assert invariants, it modifies programs according to correctness witnesses, thus ensuring whether they indeed hold, and integrates them into a reconstructed proof of correctness. Subsequently, a standard verification engine confirms that the transformed program aligns with its original specifications. In SV-COMP 2024 [4], MetaVal confirmed 30219 out of 73256 correctness witnesses for the category *ReachSafety* in C programs.

3 JCWIT OVERVIEW

Figure 1 illustrates the JCWIT's architecture and execution process. It accepts three parameters: a program to be validated or a directory, the latter indicating that all programs within it will be included in the validation scope, a specification, and a correctness witness. It is noteworthy that JCWIT does not support the simultaneous independent validation of multiple Java programs. Indeed, such a condition may happen due to the potential inclusion of references to other Java programs within the one to be validated. Besides, optional parameters are possible, such as the one asking for the current tool version.

The primary Java program to be validated should be placed first among all programs provided as input or as the first element within a directory. JCWIT assumes that a user has already adapted all input programs. With such resources in hand, the validation process of JCWIT is divided into four phases: *witness analysis*, *method monitoring*, *assertion construction*, and *verification*.

3.1 The Witness Analyzer

Indeed, the exchange formats based on the Graph Markup Language (GraphML), for violation and correctness witnesses, have become highly standardized [5]. According to the current specifications [5, 7], witnesses are presented in the form of witness automatons. Different types of Extensible Markup Language (XML) data elements in GraphML are employed to enrich edges and nodes of a graph representing witness automatons.

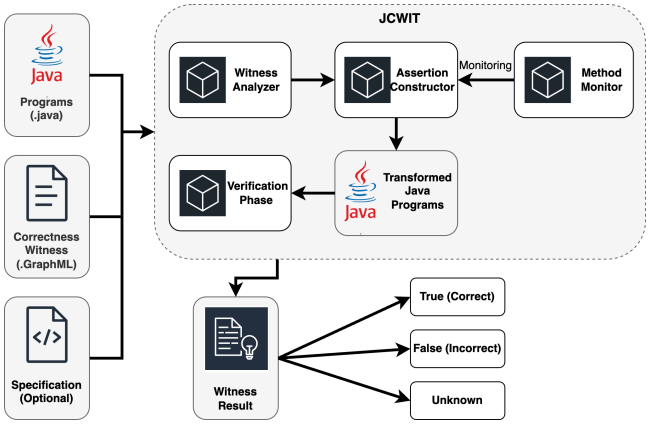

Figure 1: The JCWIT's architecture, where the specification input is optional.

In the context of witnesses, data elements are categorized into three types: graph data for witness automatons, node data for automaton states, and edge data for automaton transitions. Such elements are accompanied by different annotations (or sub-elements) based on their respective categorizations. According to the same standards, these annotations adhere to specific format requirements, optional value ranges, and whether an annotation is mandatory, encompassing three essential criteria.

JCWIT accepts a correctness witness in GraphML format [5]. Its witness analyzer examines the annotations of data elements under different classifications against these standards, thus ensuring compliance with the specified criteria. Suppose these standards dictate specific format requirements for a given annotation. In that case, the analyzer converts these format requirements into regular expressions and checks whether the content of the corresponding annotation matches the defined pattern. These optional values are compiled into a set for annotations with requirements, and the analyzer examines whether specific annotation content matches any unique optional value in this set. Furthermore, any annotations marked as mandatory must appear within their respective data elements while the analyzer searches for their presence. Any deviations from the requirements outlined in the standards are promptly identified and reported as errors.

3.2 The Assertion Constructor

In the assertion-construction phase, JCWIT transforms the invariants associated with each state, as delineated within a witness, into evaluative predicates. The predicates must be compatible with the Java runtime environment in terms of syntax and semantics to be asserted as conditions.

Initially, invariants associated with an original program are extracted from a correctness witness. They encapsulate precise values of value-type variables or identifiers of class instances associated

[1] https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks

with reference-type variables. Indeed, our procedural framework extracts and documents the definitive values associated with value-type variables and the corresponding class identifiers for reference-type ones, using regular expressions. For example in Listing 1, the determinant value 5 in line 4 was extracted from the corresponding XML element *<data key="invariant">anonlocal::1i = 5;</data>* in the correctness witness. Concurrently, detailed metadata about a program's execution state to which these invariants belong is documented. This encompasses the line number, in an original program, to which the execution state belongs, the applicability scope of the respective invariant, and the file to which the execution state belongs.

Subsequently, leveraging the mentioned metadata, the execution statement within an original program, corresponding to an invariant's execution state, is identified. Then, JCWIT embarks on formulating judgment conditions, which are bifurcated based on the nature of the variables involved in the execution statement: value or reference type. For value-type variables, JCWIT crafts a judgment condition predicated on the equivalence between their specific values and the names of the corresponding variables within an original program. In the case of reference-type variables, the keyword `instanceof` is employed to establish a judgment condition linking their corresponding variable names to the class identifiers of the reference-type variables. Upon the construction of these judgment conditions, they are subsequently asserted. Indeed, assertions are strategically inserted following the execution statements that determine them, thus ensuring the continuity and integrity of a verification process. As shown in the example program in Listing 1, from one execution statement to another, we load assertions to check whether conditions generated from a correctness witness can hold during verification. It can be seen in lines 5 and 7.

```
1 public static void main(String[] args) {
2     isRightTriangle(3,4,5);}
3 static void isRightTriangle(int a, int b, int c){
4     int x = max(max(a,b),c);
5     assert x == 5; //Invariant x = 5 is asserted.
6     int y = a * a + b * b + c * c;
7     assert y == 50; //Invariant y = 50 is asserted.
8     if(y == 2 * x * x) assert true;
9     else assert false;}
```

Listing 1: Example program with loaded assertions.

3.3 The Method Monitor

A notable challenge in validation processes is that not all assertions inserted after the same execution statement, in a method, can be universally applicable simultaneously. In a Java program, a particular method can be called more than once, leading to the same statement appearing in the witness with various distinct states. Consequently, multiple invariants corresponding to these states are simultaneously asserted and inserted after the same execution statement. Thus, during runtime or when employing JBMC, carelessly inserted assertions that are not pertinent to the current method invocation will be inadvertently executed.

The JCWIT's method monitor prevents the occurrence of such phenomena. It consists of counters (static variables) and a static method named *assertionSelection*, as shown in Listing 2. The rationale of method monitor is to enable JCWIT to selectively execute assertions based on the current number of times a method is being invoked. This way, multiple invariants in the witness resulting from multiple executions of the same statement are correctly asserted.

The application of the method monitor details three distinct stages. Initially, we introduce method counters to record the frequency of method calls during program execution. Employing the existing tool javalang[2], a Java program is parsed into an Abstract Syntax Tree (AST). Filtering is applied to the AST to extract all method declarations, encompassing details such as the class to which they pertain, their names, parameter types, and return value types. Following this, individual counters are instantiated for each method as static variables, as in line 2 of Listing 2, introducing a novel naming format for them:

`className_methodName_parameterTypes_returnValueType.`

Subsequently, upon each invocation, the corresponding method's counter is incremented accordingly, as in line 3 of Listing 3. Using these counters, JCWIT can determine which conditions should be asserted and executed by a Java program based on the method's invocation counts.

Finally, JCWIT's method monitor internally maintains a static method named *assertionSelection* in Listing 2, which only runs in method code blocks other than the main method. The first parameter *index* corresponds to the index of the counter of the method to which the execution statement belongs. The second parameter *conditions* represents the conditions formed by the invariants of all states corresponding to the execution statement in a witness.

In Listing 3, JCWIT injects this static method *assertionSelection* in lines 7 and 9 after executing statements within the method. On the first invocation to the method *isRightTriangle*, the invariants $x = 5$ and $y = 50$ are asserted. Subsequently, the method's counter is incremented, and the invariants $x = 10$ and $y = 200$ are asserted on the second invocation. This way, invariants corresponding to different states of the same execution statement will be asserted separately rather than simultaneously.

```
1 //The counter of the example program is:
2 static int Test_isRightTriangle_III_V = 0;
3 public static void assertionSelection (int index,
      boolean ... conditions){
4     assert conditions[index];}
```

Listing 2: The static method that selectively executes assertions.

```
1 public static void main(String[] args) {
2     isRightTriangle(3,4,5);
3     Test_isRightTriangle_III_V++;//Counter increments.
4     isRightTriangle(6,8,10);}
5 static void isRightTriangle(int a, int b, int c){
6     int x = max(max(a,b),c);
7     assertionSelection(Test_isRightTriangle_III_V, x ==
        5, x == 10);
8     int y = a * a + b * b + c * c;
9     assertionSelection(Test_isRightTriangle_III_V, y ==
        50, y == 200);
10    if(y == 2 * x * x) assert true;
11    else assert false;}
```

Listing 3: Example program with loaded assertions where methods are invoked multiple times.

[2] https://github.com/c2nes/javalang

3.4 Verification Phase

Following the previously mentioned stages, the transformed Java programs are now instrumented for verification. Moreover, given that JBMC only supports verification of `.class` files, JCWIT compiles the transformed source code into this file type, which is then used as input for verification.

The results of the witness validation are categorized into three types. A successful verification report indicates that all invariants within a correctness witness automaton are valid and that the transformed program's execution trajectories and states align consistently with those of the original one. A report informing verification failure means that a correctness witness automaton contains erroneous program execution trajectories or that the invariants within the execution states are incorrect, rendering them unsuitable as cues to guide verifiers in their proofs. Finally, a report informing an unknown verification result indicates that JCWIT has encountered impediments during the verification process of a `.class` file, which obstructed a conclusive assessment. Such impediments may include assertion construction failures, exceeding predefined time limits, and program compilation errors.

4 EVALUATION

JCWIT participated in the 13th International Software Verification Competition (SV-COMP 2024)[4], completing its submission with comprehensive evaluation results focused on validating the correctness witnesses. All datasets and validation tools utilized in the competition have been made publicly available, thus facilitating inspection and enabling the replication of our research.

Experimental environment. In SV-COMP 2024, each verification run was initiated on machines equipped with the GNU/Linux operating system (x86_64-linux Ubuntu 22.04) and featuring an Intel Xeon E3-1230 v5 Computer Processing Unit (CPU) (3.40 GHz) with eight processing units. Each verification task was subject to three specific resource constraints: a memory cap of 7 GB, a CPU runtime limit of 15 minutes, and a restriction to 2 CPU processing units.

Benchmark tasks. In SV-COMP 2024, the benchmark verification tasks for the *JavaOverall* category adhered to a single specification called *ReachSafety*, which refers to verifying whether a given program can reach a specified program location, safety property, or state during execution without violating any safety constraints. The verdicts of these verification tasks can be categorized as *true*, *false*, and *unknown*, corresponding to the generation of a correctness witness, a violation witness, and the absence of any witness, respectively.

4.1 Results

Table 1 reveals that JCWIT confirmed 67 out of 105 correctness witnesses. Moreover, 38 witness validation tasks were regarded as *unknown*. Indeed, our results demonstrate that JCWIT effectively captured most values or class instance identifiers from single or multiple invariants and injected these as assertion statements into original programs.

During JCWIT's validation process, 32 tasks encountered errors related to loading and compiling assertions. Specifically, in 13 of them, JCWIT could not extract valid invariants due to its current lack of support for extracting array-related invariants from witnesses. This limitation prevents the recognition of deterministic values at different array indices, resulting in its failure to assert them. Additionally, in 19 validation tasks, most compilation errors triggered by syntax errors in assertions occurred in code blocks containing string variables. This is because the correctness witnesses provided by various verifiers failed to capture operations (assignments) related to strings. Consequently, the respective variable values in invariants were considered missing or mistakenly recorded as *null*. In addition, the time limit for validating witness correctness was set at a relatively high level of 15 minutes, which resulted in only three witness verification tasks experiencing timeouts.

Table 1: Results of Correctness Witness Validation.

Result Type	Quantities	Comment
True	67	Witness Confirmed
Unknown	13	Invariant Extraction/Insertion Failed
(38	19	Program Compilation Error
Correctness	3	Validation timeout
Witnesses)	3	IO Exception

5 CONCLUSION

This paper proposes JCWIT, a novel tool for validating correctness witnesses within Java programs that distinguishes itself by its specialized focus. It addresses an existing gap in Java program validation by facilitating rigorous validation against reachability safety specifications. We believe this contribution improves the accuracy of results verified by third-party tools, thereby increasing developers' confidence regarding the reliability and safety of software systems.

However, JCWIT still lacks effective handling of invariants obtained by operations on string-type and array-type variables, which prevents it from achieving better performance.

Therefore, we plan to implement possible solutions to address these limitations in future work. For array-related invariants, one strategy involves exploring effective regular expressions to match the indices and their corresponding values in the invariants, thereby constructing the required conditions for assertions. For string-related invariants, a possible solution is for JCWIT to flag invariants with missing string-type values and skip asserting such invariants during the assertion construction phase based on these flags. The feasibility of these strategies will be verified in subsequent work.

6 TOOL AVAILABILITY

JCWIT is publicly available under the Apache 2.0 license. The latest version of JCWIT can be downloaded at GitHub[3]; the packaged JCWIT archive is published separately on Zenodo[4]; the demo of the tool in its current version is published on YouTube[5].

[3] https://github.com/Chriszai/JCWIT
[4] https://doi.org/10.5281/zenodo.12605446
[5] https://youtu.be/w2rIAdOPyE8

REFERENCES

[1] Jade Alglave, Alastair F. Donaldson, Daniel Kroening, and Michael Tautschnig. 2011. Making Software Verification Tools Really Work. In *Automated Technology for Verification and Analysis*, Tevfik Bultan and Pao-Ann Hsiung (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 28–42.

[2] Dirk Beyer. 2022. Progress on Software Verification: SV-COMP 2022. In *Tools and Algorithms for the Construction and Analysis of Systems*, Dana Fisman and Grigore Rosu (Eds.). Springer International Publishing, Cham, 375–402.

[3] Dirk Beyer. 2023. Competition on Software Verification and Witness Validation: SV-COMP 2023. In *Tools and Algorithms for the Construction and Analysis of Systems*, Sriram Sankaranarayanan and Natasha Sharygina (Eds.). Springer Nature Switzerland, Cham, 495–522.

[4] Dirk Beyer. 2024. State of the Art in Software Verification and Witness Validation: SV-COMP 2024. In *Tools and Algorithms for the Construction and Analysis of Systems*, Bernd Finkbeiner and Laura Kovács (Eds.). Springer Nature Switzerland, Cham, 299–329.

[5] Dirk Beyer, Matthias Dangl, Daniel Dietsch, and Matthias Heizmann. 2016. Correctness witnesses: exchanging verification results between verifiers. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 326–337. https://doi.org/10.1145/2950290.2950351

[6] Dirk Beyer, Matthias Dangl, Daniel Dietsch, Matthias Heizmann, Thomas Lemberger, and Michael Tautschnig. 2022. Verification Witnesses. *ACM Trans. Softw. Eng. Methodol.* 31, 4, Article 57 (sep 2022), 69 pages. https://doi.org/10.1145/3477579

[7] Dirk Beyer, Matthias Dangl, Daniel Dietsch, Matthias Heizmann, and Andreas Stahlbauer. 2015. Witness validation and stepwise testification across software verifiers. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 721–733. https://doi.org/10.1145/2786805.2786867

[8] Dirk Beyer, Matthias Dangl, Thomas Lemberger, and Michael Tautschnig. 2018. Tests from Witnesses. In *Tests and Proofs*, Catherine Dubois and Burkhart Wolff (Eds.). Springer International Publishing, Cham, 3–23. https://doi.org/10.1007/978-3-319-92994-1_1

[9] Dirk Beyer and Martin Spiessl. 2020. MetaVal: Witness Validation via Verification. In *Computer Aided Verification*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer International Publishing, Cham, 165–177.

[10] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, and Yunshan Zhu. 2009. Bounded model checking. *Handbook of satisfiability* 185, 99 (2009), 457–481.

[11] Maria Christakis and Christian Bird. 2016. What developers want and need from program analysis: an empirical study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE '16)*. Association for Computing Machinery, New York, NY, USA, 332–343. https://doi.org/10.1145/2970276.2970347

[12] Lucas Cordeiro, Pascal Kesseli, Daniel Kroening, Peter Schrammel, and Marek Trtik. 2018. JBMC: A Bounded Model Checking Tool for Verifying Java Bytecode. In *Computer Aided Verification*, Hana Chockler and Georg Weissenbacher (Eds.). Springer International Publishing, Cham, 183–190. https://doi.org/10.1007/978-3-319-96145-3_10

[13] Lucas Cordeiro, Daniel Kroening, and Peter Schrammel. 2019. JBMC: Bounded Model Checking for Java Bytecode. In *Tools and Algorithms for the Construction and Analysis of Systems*, Dirk Beyer, Marieke Huisman, Fabrice Kordon, and Bernhard Steffen (Eds.). Springer International Publishing, Cham, 219–223. https://doi.org/10.1007/978-3-030-17502-3_17

[14] Lucas C. Cordeiro, Daniel Kroening, and Peter Schrammel. 2019. Benchmarking of Java Verification Tools at the Software Verification Competition (SV-COMP). *SIGSOFT Softw. Eng. Notes* 43, 4 (jan 2019), 56. https://doi.org/10.1145/3282517.3282529

[15] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. 2008. A Survey of Automated Techniques for Formal Software Verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27, 7 (2008), 1165–1178. https://doi.org/10.1109/TCAD.2008.923410

[16] Falk Howar and Malte Mues. 2022. GWIT: A Witness Validator for Java based on GraalVM (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems*, Dana Fisman and Grigore Rosu (Eds.). Springer International Publishing, Cham, 446–450.

[17] R.M. McConnell, K. Mehlhorn, S. Näher, and P. Schweitzer. 2011. Certifying algorithms. *Computer Science Review* 5, 2 (2011), 119–161. https://doi.org/10.1016/j.cosrev.2010.09.009

[18] Herbert Rocha, Raimundo Barreto, Lucas Cordeiro, and Arilo Dias Neto. 2012. Understanding Programming Bugs in ANSI-C Software Using Bounded Model Checking Counter-Examples. In *Integrated Formal Methods*, John Derrick, Stefania Gnesi, Diego Latella, and Helen Treharne (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 128–142. https://doi.org/10.1007/978-3-642-30729-4_10

[19] D.R. Wallace and R.U. Fujii. 1989. Software verification and validation: an overview. *IEEE Software* 6, 3 (1989), 10–17. https://doi.org/10.1109/52.28119

[20] Tong Wu, Peter Schrammel, and Lucas C. Cordeiro. 2022. Wit4Java: A Violation-Witness Validator for Java Verifiers (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems*, Dana Fisman and Grigore Rosu (Eds.). Springer International Publishing, Cham, 484–489. https://doi.org/10.1007/978-3-030-99527-0_36