

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

---

Zaiyu Cheng, Tong Wu, Peter Schrammel, Norbert Tihanyi,  
Eddie B. de Lima Filho, Lucas C. Cordeiro

# Introduction

- **Witness** is a format where verifiers can exchange information.
  - **Witness validation** is a formal verification method to independently validate software verification tool result.
  - The formal software verification work is usually performed by **untrusted verification engines** or **static analysis tools** with the risk of **false positives**.
    - Witness validation is therefore particularly important.
- 
- Correctness witness validation
  - Violation witness validation

# Motivation

- There are several violation validation tools for the programming language **Java**.
  - GWIT
  - Wit4Java

There is **no** dedicated correctness witness validator!

- There were **seven** Java **verifiers** in SV-COMP 2023.
- Before **JCWIT** was born, there were only **two** Java **validators** in SV-COMP 2023.

```

JavaOverall:
properties:
  - assert_java
categories:
  - assert_java.ReachSafety-Java
verifiers:
  - coastal
  - gdart
  - java-ranger
  - jayhorn
  - jbmc
  - jdart
  - mlb
  - spf
  - swat
validators:
  - wit4java-violation-1.0
  - gwit-violation-1.0
    
```

# Principles

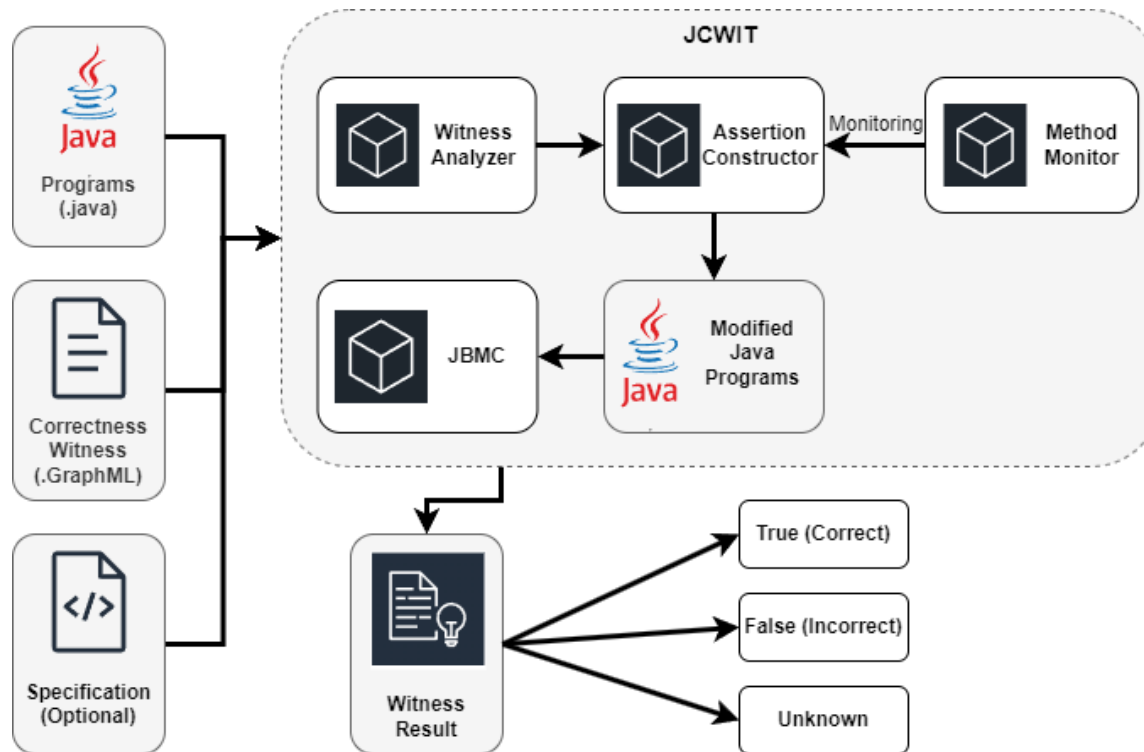
Drawing inspiration from the existing methodology  
**validation via verification**

- Transform the program such that the invariants are **asserted**.
- A standard **verification engine** is then asked to verify that the **transformed program** satisfies the original **specification** and all **assertions** added to the program hold.

The core idea of this approach is to use **existing verification tools** to confirm the **accuracy** and **reliability** of the **evidence** generated by the verification process

# Architecture

We implemented an open source<sup>1</sup> tool for  
Java Correctness Witness



1. <https://github.com/Chriszai/JCWIT>

# Witness Analyzer

- There are **three** data elements types in the witness.
  - Graph Data for Witness Automaton
  - Node Data for Automaton States
  - Edge Data for Automaton Transitions
- The data elements are accompanied by **different** annotations (or sub-elements) based on their **respective categorizations**.

# Assertion Constructor

1. Match and extract the values of value-type variables or class identifiers of reference-type variables from the invariants in the witness.
2. Assert the extracted invariants at the appropriate positions based on the meta-information.

```

<node id="n0">
  <data key="invariant">
    java.lang.Object.@class identifier="java::Example";
  </data>
</node>
<edge source="n0" target="n1">...</edge>
<node id="n1"><data key="invariant">anonlocal::1i = 5;</data></node>

```

The value and class identifier are stored in the witness in the above form

# Method Monitor

When a particular **method** is executed **multiple** times, inserted assertions that are **not** pertinent to the current method invocation will be **inadvertently executed**

## Solution:

1. Introduce **method counters** to record the **frequency** of method calls during program execution.
2. Upon **each** invocation, the corresponding method's counter is **incremented** accordingly.
3. The method monitor maintains a **static method** internally so that the different **invariants** corresponding to the same execution statement will be asserted **separately** rather than **simultaneously**.

The principle is to allow programs to be able to selectively execute assertions based on the current number of times a method is being invoked



# Example

```
public static void main(String[] args) {  
    Example.isRightTriangle(3,4,5);  
    Example.isRightTriangle(6,8,10);}  
static void isRightTriangle(int a, int b, int c){  
    int x =max(max(a,b),c);  
    int y= a * a + b * b + c * c;  
    if(y == 2* x * x) assert true;  
    else assert false;}
```

# Example

```
public static void main(String[] args) {
    Example.isRightTriangle(3,4,5);
    Example.isRightTriangle(6,8,10);}
static void isRightTriangle(int a, int b, int c){
    int x =max(max(a,b),c);
    int y= a * a + b * b + c * c;
    if(y == 2* x * x) assert true;
    else assert false;}
```



The static method are inserted

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

# Example

```
public static void main(String[] args) {
    Example.isRightTriangle(3,4,5);
    Example.isRightTriangle(6,8,10);}
static void isRightTriangle(int a, int b, int c){
    int x =max(max(a,b),c);
    int y= a * a + b * b + c * c;
    if(y == 2* x * x) assert true;
    else assert false;}
```



The static method are inserted

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

- On the first invocation, the invariants  $x = 5$  and  $y = 50$  are asserted. The method's counter is then incremented, and the invariants  $x = 10$  and  $y = 200$  are asserted on the second invocation.



# Verification

- When everything is ready, JCWIT compiles the transformed program again as a `.class` file, and is then used as `input` for verification by the `Java Bounded Model Checker (JBMC)`.
- The results of the verification will be divided into `three` types.
  - True
    - A successful verification report indicates that all invariants within a correctness witness automaton are `valid`.
  - False
    - A report informing verification failure means that a witness automaton contains `erroneous` program execution `trajectories` or that the `invariants` within the execution states are `incorrect`.
  - Unknown
    - A report informing an unknown verification result indicates that JCWIT has encountered `impediments` during the verification process of a `.class` file.

# Evaluation

- 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.

JCWIT participated in JavaOverall category adhered to a single specification called ReachSafety in SV-COMP 2024.

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

# Future Work

- **Extend this work for the new witness format.** Last year, a new format was introduced for the competition, we could extend this work to support this new format.
- **Incorporate LLMs to produce invariants for validation.** The witness validation could be used as a method to check whether invariants produced by an LLM are valid.

# Reference

1. Howar, F., Mues, M. (2022). **GWIT: A Witness Validator for Java based on GraalVM (Competition Contribution)**. In: Fisman, D., Rosu, G. (eds) Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2022. Lecture Notes in Computer Science, vol 13244. Springer, Cham. [https://doi.org/10.1007/978-3-030-99527-0\\_29](https://doi.org/10.1007/978-3-030-99527-0_29)
2. Wu, T., Schrammel, P., Cordeiro, L.C. (2022). **Wit4Java: A Violation-Witness Validator for Java Verifiers (Competition Contribution)**. In: Fisman, D., Rosu, G. (eds) Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2022. Lecture Notes in Computer Science, vol 13244. Springer, Cham. [https://doi.org/10.1007/978-3-030-99527-0\\_36](https://doi.org/10.1007/978-3-030-99527-0_36)
3. Beyer, D., Spiessl, M. (2020). **MetaVal: Witness Validation via Verification**. In: Lahiri, S., Wang, C. (eds) Computer Aided Verification. CAV 2020. Lecture Notes in Computer Science(), vol 12225. Springer, Cham. [https://doi.org/10.1007/978-3-030-53291-8\\_10](https://doi.org/10.1007/978-3-030-53291-8_10)
4. Cheng, Z., Wu, T., Schrammel, P., Tihanyi, N., de Lima Filho, E. B., & Cordeiro, L. (Accepted/In press). **JCWIT: A Correctness-Witness Validator for Java Programs based on Bounded Model Checking**. In The ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA).



The University of Manchester

**Thank you !**