

# Develop and Evaluate a Security Analyzer for Finding Vulnerabilities in Java programs



The University of Manchester

A dissertation submitted to The University of Manchester for the degree of Master of  
Science in the Faculty of Science and Engineering

2023

By

Zaiyu Cheng(10510215)

Department of Computer Science

# Contents

Contents .....	2
Abstract .....	4
Acknowledgments .....	5
Chapter 1 Introduction .....	6
1.1 Problem Description .....	7
1.2 Aims and objectives .....	8
1.3 Contribution .....	8
1.4 Organisation of Dissertation .....	9
Chapter 2 Background .....	10
2.1 Security Vulnerabilities .....	10
2.2 Software Testing .....	12
2.2.1 Software Verification Competition .....	14
2.3 Bounded model checking .....	16
2.3.1 Symbolic model checking .....	17
2.3.2 Bounded model checking .....	17
2.3.3 Java Bounded Model Checking .....	18
2.4 Witness Validation .....	21
2.4.1 Witness validation for Java .....	22
2.5 Other Java verification tools .....	25
2.6 Summary of background .....	26
Chapter 3 Research Methodology .....	28
3.1 System Architecture .....	28
3.2 Algorithm .....	30
3.2.1 Python script .....	30
3.2.2 Directed graph checking .....	33
3.2.3 Validation harness creation .....	37
3.2.4 Analysis about the Algorithms .....	39
3.3 Illustrative Examples .....	40
3.3.1 Int Example .....	40
3.3.2 Long Example .....	42

3.3.3 Short Example .....	44
3.3.4 Float Example .....	47
3.3.5 Double Example .....	49
3.3.6 Boolean Example .....	51
3.3.7 Char Example .....	53
Chapter 4 Experimental Evaluation .....	55
4.1 Setup .....	55
4.1.1 Environment installation for Java .....	55
4.1.2 Environment installation for Python .....	56
4.1.3 JBMC installation .....	56
4.1.4 Mockito installation .....	57
4.1.5 Proposed extension .....	57
4.1.6 Benchmarks .....	57
4.1.7 Environment versions .....	58
4.1.8 Running the tests .....	58
4.2 Objectives of the evaluation .....	59
4.3 Results .....	59
4.4 Threats to validity .....	65
Chapter 5 Related Work .....	66
Chapter 6 Conclusion .....	68
Bibliography .....	70

# Abstract

The purpose of this work is to comprehend how software validation is used to find security flaws, to learn about the tools that have been built for software validation and their expansions, and to build on those tools for development and evaluation. A bounded model checker called Java Bounded Model Checking (JBMC) enables the verification of Java programmes. Its objective is to precisely check and examine the correctness of Java software in accordance with a specified specification as an effective software verification tool. Our attention is on how to go about proving the accuracy witness provided by JBMC since the specification in this study refers to the correctness property or the violation property and earlier implementations have extended conflict witnesses in more comprehensive ways. Although software verifiers are very accurate in their operations, there may still be a chance of disagreements among correctness witness in a given circumstance, therefore verifying witnesses is required to raise public confidence in JBMC.

The focus of this study is testimonial verification of Java programmes as we continue to play about with testimonials in GraphML format. The Verification Toolkit, Python, and Mockito, a Java simulation framework, were used to do this. We have developed a built-in random class based on Python that can translate arbitrary random numbers into random numbers adapted to the various types of java to aid us with our testing due to some of the validation restrictions of JBMC. It is necessary to validate the witnesses in order to boost public confidence in JBMC since, despite the software validator's high level of accuracy, there is still a chance that the correctness witnesses will disagree in a particular circumstance.

# Acknowledgments

Firstly, I would like to thank my Masters project supervisor, Dr Lucas Cordeiro. He gave me a lot of help and guidance in completing my thesis, and it was helpful to have his comments on my assignments during the weekly group meetings.

Secondly, I would also like to thank my parents for supporting me in completing my MSc. In addition, I would like to thank the University of Manchester for giving me the opportunity to complete my MSc and creating a very excellent learning environment for me. 17 years of study have come to an end, with a lot of frustration and bystander help along the way. The process was painful at times, but the rewards of persevering were very gratifying.

# Chapter 1 Introduction

Java is a widely used object-oriented programming language for cross-platform development, introduced in 1995 by Sun Microsystems (now a subsidiary of Oracle Corporation). The Java language was originally created as a solution to the problem of developing for embedded devices, and it can run on different operating systems, requiring only the installation of the Java virtual machine on different platforms. This cross-platform nature has made Java a widely used language for cross-platform development.

Generally speaking, we usually think of the JDK is the core of Java, which includes the JRE (Java Runtime Environment), some Java tools (javac/java/jdb, etc.) and Java class libraries. JVM is the core runtime environment of the Java language, it is a virtual computer for the execution of Java byte code and is also part of the JRE. The JVM is the core runtime environment of the Java language, it is a virtual computer used to execute Java bytecode is also part of the JRE. Java source code in the compiler is compiled into bytecode (a kind of intermediate code), rather than compiled directly into the machine code associated with a particular hardware platform. This allows Java programs to run on any platform that supports the JVM without the need for recompilation.

The JIT compiler is an integral part of the JVM. The JIT compiler has two main modes: interpreted mode and compiled mode. Interpretation mode will be in the programme execution line by line interpretation of the byte code, and immediate execution of the corresponding operation. This mode starts fast, but the execution speed is relatively slow. The compiled mode JIT will monitor the execution of the program, if a method is frequently called, it will be the method of the byte code compiled into the local machine code, and optimised [1]. This allows the optimised machine code to be run directly when the method is subsequently executed, thus increasing execution speed. With the JIT compiler, the performance of Java programs can be comparable to, or in some cases even better than, traditional compiled languages such as C++, while retaining the cross-platform nature of Java.

Thus, the process of compiling a Java program from compilation to execution can be divided into the following steps [2]:

- **Compiling Java source code:** Java source code needs to be compiled into Java bytecode, which is usually stored as a .class file. The compilation process can be done using a Java compiler.
- **Loading bytecode files by the class loader:** The Java Virtual Machine (JVM) needs to load bytecode files into memory at runtime. The class loader is responsible for loading the .class file into the JVM and also for parsing the class information in the bytecode file.
- **Bytecode verification:** After loading the bytecode file, the JVM verifies the bytecode file to ensure that the bytecode file conforms to the JVM specification and does not contain any security vulnerabilities.
- **Bytecode interpretation or JIT compilation:** When executing bytecode, the JVM can choose to either interpret the bytecode directly via an interpreter or compile the bytecode into native machine code for execution via a Just-In-Time (JIT) compiler.

## 1.1 Problem Description

One of the languages that is currently most often used worldwide is Java. It is used in a wide range of projects and applications where a seemingly insignificant programming error or weakness can have a significant impact on costs associated with maintenance and repair. The Java Development Kit (JDK) itself features a security architecture that consists of a range of Application Programming Interfaces (APIs) and tools in order to prevent security vulnerabilities. However, Java applications continue to be plagued by an increasing number of security vulnerabilities, such as bugs and weaknesses that are rarely fully detected by traditional modelling and testing methodologies. We therefore propose model checking approaches to automatically check software for defects by examining all realisable states.

The Java Boundary Model Checking Tool (JBMC), a verification tool for locating bugs in Java software, is highlighted in this thesis. JBMC effectively verifies Java bytecode by inspecting every realisable state to automatically examine software being written for flaws [3]. However, in some circumstances, the JBMC benchmark results generate some false results, including false validation results. Because of this, the results of running JBMC must be extended and checked, leading to the emergence of violation witnesses and correctness witnesses. In this thesis, we will concentrate on the verification of correctness

witnesses.

## 1.2 Aims and objectives

This project focuses on building on the Java Bounded Model Checking tool (JBMC), evaluating existing validation strategies and implementing a suitable extension that performs witness verification based on GraphML witness files generated by the JBMC verifier.

The specific objectives of this paper are to:

- Be able to understand the JBMC tool well enough to understand the principles and mechanisms used to discover security vulnerabilities.
- Evaluate available strategies for finding security vulnerabilities in JBMC.
- Research and implemented suitable JBMC extensions in Python and Java, primarily targeting correctness witness.
- Discuss the accuracy and effectiveness of the implemented extensions.

## 1.3 Contribution

The contribution of this MSc project can be broadly categorised into two main implementations respectively Python scripts and validation harness. These Python scripts are used to extract eligible instances from the validation file and inject them into the validation harness. An appropriate simulation framework, Mokito, is then used to assign these extracted instances to the newly generated Java program, re-verifying the program using Java with the role of whether or not the verification results can be reproduced. This is partly based on the algorithm in the previous implementation, and we need to modify and optimise some details to compensate for its initial shortcomings. The other part of the project was to verify the correctness and completeness of the witness file, which consisted of some Python scripts. In order to determine whether the relationship between nodes and edges is correct, algorithms are used to complete the tests for the presence of loops in the witness file and for the correct order of program execution.

In contrast to witness verification based on the C language, which has been expanded and



implemented by numerous academics and organisations, witness verification based on the Java language has received less attention and is still in its early stages. Examining the possibility for witness verification techniques in extended Java programmes built on JBMC and earlier versions is another contribution of this research. We try to propose witness verification as a fresh approach for validating Java programme vulnerabilities using our improved verification tool. Witness verification has a prospective use in the verification of other programmes, according to our research and application.

## **1.4 Organisation of Dissertation**

The whole thesis is divided into six main chapters. The first chapter is a general introduction to the whole project, in which the problem to be solved and the purpose of the research are described. In Chapter 2, we discuss the current security pitfalls of Java and the background of bounded model checking. Chapter 3 deals with the methodology proposed in this project. We discuss and analyse in detail the design and implementation process of the algorithm, showing part of the code for explanation. In Chapter 4, the evaluation chapter, the tested methodology and key results are shown and discussed. In Chapter 5, some related research results are discussed and compared. Finally, in Chapter 6, the results and outcomes of the thesis are summarised, reflected upon and discussed, and some ideas and suggestions for future work are given.

# Chapter 2 Background

This chapter's purpose is to give my thesis the proper and vital foundation it needs, including knowledge and ideas crucial to my issue. In order to fully explain the definitions of some of the proper nouns and to give the most recent advancements on the subject, the majority of the information has been taken from published papers and websites. Three pieces are present. Background information on security flaws and their implications for contemporary society is given in Section 2.1. The idea of software testing is covered in Section 2.2, along with how it can be used to lessen the possibility of vulnerabilities and exploits in various applications and systems. Software model checking is covered in Section 2.3, where the software model checker is introduced and its architecture and execution method are further explained using JBMC as an example. After that, in subchapter 2.4, which describes the extended architecture and algorithmic implementation method based on prior implementations, witness verification is discussed and given some insight as the foundation for this thesis research. The limits of the current witness verifiers for Java are thoroughly covered in Chapter 2.5, which also introduces a few Java verification tools. The key findings are summarised in Chapter 2.6.

## 2.1 Security Vulnerabilities

The extensive usage of digital technology is one of the primary reasons why people are becoming more and more concerned about security breaches. As the use of the Internet, smartphones, and other digital technology increases, people's lives are becoming more and more dependent on online services and applications. Security risks are associated with this digital lifestyle's convenience, with security breaches being one of the key issues.

Due to the popularity of services like financial services, healthcare apps, smart home devices, etc., the amount of data on the Internet is growing tremendously. In general, these services and apps include large volumes of sensitive data and personal information. Security flaws could allow hackers or other malicious individuals to obtain this data and utilise it for illegal purposes including identity theft, financial fraud, and privacy invasion. Additionally, malware can propagate via security flaws, and many of these IoT devices are

vulnerable to attack. As a result, they can serve as entry points for cyberattacks that pose substantial risks to users' devices and data. More than 50% of databases had at least one vulnerability in 2002, according to a study by the FBI and the Computer Security Institute (CSI), each of which could cost users around \$4 million[4] in losses. More than 7.9 billion data records were unintentionally compromised, according to a different estimate from DiityForce, between January and September 2019 [5]. Software vulnerabilities are typically the most prominent culprit, along with a few management-related factors.

A security vulnerability is a fault or weakness that could allow an attacker to enter a system or network without authorization, steal data, or harm an application. These vulnerabilities can arise in hardware, software, or human actions and be brought on by a variety of things, such as programming errors, configuration problems, design flaws, or insufficient security measures. The increasing reliance of our society on technology and the internet has led to a considerable increase in security vulnerabilities. As more systems and gadgets are connected to the internet, the potential attack surface for hackers and cybercriminals is growing. The Top 25 Most Dangerous Software Weaknesses (CWE Top 25) of the Common Weaknesses Enumeration (CWETM) provides information on issues that regularly arise each year. Buffer Boundary Violation, which is defined as what occurs when a programme attempts to write more data to a buffer (for example, an array or buffer) than its preallocated size, has risen to the top spot in the 2023 CWE Top 25 compared to 2020. Additionally, since 1999, the Common flaws and Exposures (CVE) system has served as a resource for publicly known information security flaws and exposures. Charts and graphs are used to depict and analyse various levels, types, and frequencies of software vulnerabilities.

"A weakness in an asset or control that can be exploited by one or more threats" is how ISO27000 defines a vulnerability. Vulnerability[6] can be caused by technical causes, such as software bugs, or even human sources, such as the use of weak passwords because they are easier to remember. Exploiting vulnerabilities, whether on purpose or accidentally, typically leads to organisational disaster. Java is a type-safe programming language thanks to its numerous safety features, which include automatic memory management and garbage collection [7]. These qualities ensure programme robustness while, to a certain extent, lowering the probability of errors. The compiler also transforms Java programmes into bytecode, which is subsequently checked by a verifier before execution [8]. Java

application programmers have tools to increase security through the usage of Java Secure Application Programming Interfaces (APIs). However, the CVE database has 673 JRE-related vulnerabilities, 39 of which were disclosed the year before (2020). Despite the fact that the Java programming language has many security protections built in, vulnerabilities occasionally appear in Java programmes.

## **2.2 Software Testing**

Modern software systems are becoming increasingly complex, involving multi-layered architectures, large amounts of code and components, and complex data interactions. The increased complexity of software systems makes errors and defects in software more difficult to detect and resolve. In addition, with the proliferation of the Internet, software systems face an increasing number of security threats and attacks. Hackers and malicious users exploit software vulnerabilities to attack and intrude, posing a threat to personal privacy and data security. Therefore, with the rapid development of computer technology and the wide application of software in daily life, industry and business, software testing becomes more and more important.

Software testing is the systematic verification and evaluation of a software system during the software development process to identify potential defects, bugs, and problems, and to ensure that the software quality meets expectations. Software testing is a critical step in the software development lifecycle, designed to ensure that software functionality, performance, security, and stability meet expectations.

During software testing, testers use different testing techniques and methodologies to design and execute a series of test cases that simulate different situations and scenarios to validate various aspects of the software. Test cases include input data, expected outputs and execution steps to check whether the software behaves as expected under different conditions.

In general, the main purpose of software testing is to ensure that the software system has high quality, reliability and good performance to meet the needs and expectations of the users. The following three points are the most important purposes of software testing:

- **Finding Defects and Bugs:** The main objective of software testing is to find bugs, defects and problems in the software system. By continuously executing test cases, testers can find and document various problems in the software, including functional errors, interface problems, security vulnerabilities, etc.
- **Verifying Functionality and Performance:** Software testing is designed to verify that the functionality of a software system has been developed in accordance with the specifications and user requirements, and that the performance meets expectations. Testers run a variety of test cases to check whether the software behaves as expected in different scenarios.
- **Reduce Maintenance Costs:** Identifying and fixing problems early in the software development process can reduce maintenance costs after the software is released. The cost of fixing problems usually increases significantly later in the software development process, and software testing helps to identify problems early and reduce maintenance costs.
- **Software flaws frequently appear in high-quality programmes after numerous version updates.** Software flaws could result in a range of financial losses and a decline in client confidence. This does not imply that businesses are not serious or responsible about the software development process; rather, it just highlights how challenging it is to properly account for all potential flaws given the growing size and complexity of software. When we attempt to fix a bug, it could confuse the programme and create additional possible bugs [8]. As a result, it is typical to handle commercial-grade software with a very thorough testing strategy in order to raise the product's trustworthiness.

During software testing, testers use different testing techniques and methodologies to design and execute a series of test cases that simulate different situations and scenarios to validate various aspects of the software. Test cases include input data, expected outputs and execution steps to check whether the software behaves as expected under different conditions. When the completed system is handed over from developers to customers or users, the overall software will be accepted. The goal is to build trust in the functionality of the system, not to discover flaws in the system. Only by building a reliable testing framework that includes both functional and non functional testing methods can we ensure the creation of high-quality software. In return, our end users can deliver and successfully adopt it. Generally speaking, software testing can be divided into several levels and types, including but not limited to the following:

- Unit testing: testing against the smallest units of functionality in the software (usually functions or methods) to ensure that they operate as expected.
- Integration testing: testing the interaction between different modules or components to verify that they work together correctly when integrated.
- Functionality testing: verifies that the software meets the specifications and user requirements to ensure that the functionality works as expected.
- Performance testing: Tests the performance and response time of the software under different load conditions to evaluate its throughput, latency and resource utilization.
- Security testing: Evaluates the security of a software system and identifies potential vulnerabilities and security risks. Regression testing: After making modifications or updates, re-run the previously passed test cases to ensure that the modified parts have not introduced new bugs or broken the original functionality.

### **2.2.1 Software Verification Competition**

SV-COMP (Software Verification Competition) is a competition for software verification tools held at TACAS to promote and evaluate the development of static and dynamic software verification techniques. It addresses the problem that there is no widely distributed suite of benchmark verification tasks and that most concepts are proven only in research prototypes.

The competition provides a standard benchmark test set with a series of challenging programs to evaluate the performance and capabilities of different verification tools. This helps researchers and developers understand the strengths and weaknesses of various tools and how they perform on different types of programs.

SV-COMP consists of several tasks, each covering different verification techniques and application scenarios. These tasks can be categorized into the following main categories:

- C Task: this task focuses on the verification of C language programs. Participants submit tools that statically analyze or simulate the execution of a set of C programs to verify their correctness.
- Java Task: Similar to the C task, the Java task focuses on the verification of Java language programs. Participants submit a tool that verifies a set of Java programs.

- Horn Task: This task involves verification techniques for predicate abstraction and is primarily used to verify concurrent and parallel systems.
- ReachSafety task: In this task, the verification tool needs to determine whether a given program satisfies the Reachability Property, i.e., whether it is possible to reach a particular state from the starting state of the program.
- Concurrency task: this task involves the verification of concurrent programs, where participants need to verify the correctness of multi-threaded programs.

The purpose of the competition is to provide a platform for researchers in the field of software verification to experiment and compare the performance of verification tools through [9]. By participating in the competition, academia and industry can share their experiences and work together to promote the development and application of verification techniques. In other words, developers are encouraged to improve and optimize their tools by adopting verification methods and their performance in the software development process. Another advantage is through the provision of standardized benchmark test sets containing a series of challenging procedures covering different verification problems and application scenarios.

The evaluation process of the SV-COMP competition consists of two main phases: the prediction phase and the verification phase. In the prediction phase, participants' tools need to be run on a set of test programs to generate predictions of the verification results. During the validation phase, these predictions are compared and evaluated against the actual validation results to determine the performance and accuracy of the tool.

Similar to CBMC, JBMC can only handle bounded programs that contain known upper bounds. In SV-COMP 2021, JBMC scored 603 out of 693 and completed 423 out of 473 benchmark tasks, which means 24 It remains a powerful verification tool on Java programs [10].

## 2.3 Bounded model checking

Model checking tools are built as a complement to software testing because not all defects in complex software can be quickly discovered by creating test cases. Thus, model checking methods can take into account all potential behaviors of a finite system, unlike traditional software testing which can only build a limited number of test cases [11]. Our focus is on tools for bounded model checking, which potentially have the power to fundamentally ensure the dependability and integrity of a programme. We examine their ideas and tenets and discuss Java Bounded Model Checking (JBMC), a brand-new formal verification tool for Java programmes built on their foundation.

Model Checking is a formal verification technique for automated verification of design models of systems. It can be used to verify the design of many types of systems such as hardware systems, software systems, and parallel systems. In model checking, the design of a system is abstracted into a finite state model, usually represented using a Finite State Automaton or a Finite State Transition System. The model describes the states of the system and the transition relationships between states [12].

States, transitions, and requirements or qualities expressed as logical formulae make up the programme model. The values of all programme counters, stack and heap settings, and programme variable values are calculated in one state. Transitions describe how a programme is moved from one state to the next. To determine compliance, it is feasible to examine every potential state in a programme. The programme will potentially come to an end in a specific length of time if the state space is limited. In contrast, a counterexample is produced (signifying an improper execution trajectory) if a state is discovered to violate the correctness property. Utilising model checking approaches, features that are only partially stated, such as safety or activity, are examined. The security attribute describes the inaccessibility of bad state, such as assertion conflicts, the occurrence of null pointer dereferences or buffer overflows, or API usage contracts, such as the order of function calls [13].



### **2.3.1 Symbolic model checking**

Symbolic Model Checking is a formal verification technique for checking the nature and specification of computational systems. It uses symbolic representations when exploring the state space of a system, with the aim that it can significantly reduce the need for computational and storage resources.

Traditional model checking methods may encounter the state explosion problem when traversing the state space, i.e., the number of system states grows exponentially with the size of the system. Symbolic model checking mitigates the state explosion problem to some extent by using symbolic representations for states and operations rather than explicitly enumerating and storing all states.

Representing a system's states and operations as symbolic expressions is the fundamental concept behind symbolic model checking. The system's list of states is represented by some Boolean functions after some Boolean variables decide some of the system's states. BDDs (Binary Decision Diagrams) can be used to represent the collection of states and state transitions. The fundamental concept behind BDDs is to express Boolean functions as a directed acyclic graph, where each node denotes a Boolean variable and the edges denote the relationships between the variables' assignments. The provided state space is traversed at the logical level using a model checking procedure to see whether the attributes are true. Then, by performing logical operations on these symbolic expressions, symbolic formulas about the properties of the system can be generated. These formulas can be further passed to a logic solver or model checker for automatic verification.

### **2.3.2 Bounded model checking**

Model checking is the foundation for bounded model checking (BMC). Early in the 1990s, the idea of bounded model checking was initially put forth. In his doctoral dissertation from 1999, Armin Biere suggested using the SAT (satisfiability problem solver) in bounded model testing. An approach for resolving Boolean satisfiability issues is known as a SAT solver. Because it only concentrates on states that may be achieved in a finite number of steps and when the depth (number of steps) of the system's state is finite, or

verified within a specific bound, it is called a bounded state. This cap is typically specified by the user, either as a predetermined number of steps or as an upper limit on how much system capacity (such as memory) may be used. Therefore, systems having finite states are a natural fit for this method. For instance, a confirmed design must be unwrapped  $k$  times with an attribute and presented to the SAT solver in order to produce a propositional formula. If there is an unwrapped system of length  $k$  that contradicts the attribute, the formula is valid. Many errors that could otherwise go unnoticed are discovered by BMC. However, because of the short state depth, some flaws or characteristics might be missed.

### 2.3.3 Java Bounded Model Checking

An addition to the C Bounded Model Checker (CBMC), Java Bounded Model Checking (JBMC) is a bounded model checker for Java programmes [14]. It is based on the satisfiability module theory (SMT) and Boolean satisfiability (SAT). It is a development on top of the CProver framework for the C Bounded Model Checker (CBMC). It consists of a front-end for parsing Java bytecode, a Goto program that transforms parsed and type-checked Java programs into internal representations, and a SAT/SMT back-end. JBMC can identify a range of errors in a programme, including null pointer references, array out-of-bounds, concurrent access issues, and other types of function and runtime errors because it supports many Java language features, such as object-oriented programming, inheritance, polymorphism, and exception handling. Overall, JBMC primarily focuses on the verification of Java programmes and offers helpful tools for ensuring that Java code is valid. However, it is constrained by the memory explosion problem, which affects all formal verification tools. For instance, in the 10th Software Verification Contest, there were still 50 inaccurate or unidentified verification findings [15].

JBMC can be executed on the command line interface (CLI) using the following commands.

```
jbmc <filename> <additional properties (optional)>
```

As we know from the official JBMC manual, it does not accept Java source code as input, but only class files or jar files. Of course, it allows to specify other options, for example, if

you want to present the validation results of jbmc as GraphML, you need to add `--graphml-witness filename` to the subsequent properties.

```
<some-directory>$ <path-to-jbmc>/jbmc my.petty.examples.Simple
--unwind 5 --classpath <path-to-jbmc>/core-models.jar:.
```

If an error is found after validation is complete, the output will either show "Validation failed" or "Validation succeeded", indicating that there were no errors in the program. Alternatively, if an option is used with an incorrect format or path, "Usage error!" will be displayed, indicating that the validation did not complete and that there was an error in the command line statement. The following common commands can be used to run JBMC from the command line interface [16].

Option	Description
<code>--show-properties</code>	show the properties, but don't run analysis
<code>--symex-coverage-report f</code>	generate a Cobertura XML coverage report in f
<code>--property id</code>	only check one specific property
<code>--stop-on-fail</code>	stop analysis once a failed property is detected
<code>--trace</code>	give a counterexample trace for failed properties
<code>--function name</code>	set the function name as a custom entry point the the program (instead of the standard main function
<code>--unwind nr</code>	unwind nr times
<code>--graphml-witness filename</code>	write the witness in GraphML format to filename
<code>--unwinding-assertions</code>	generate unwinding assertions

Table 2.1: Optional properties of JBMC

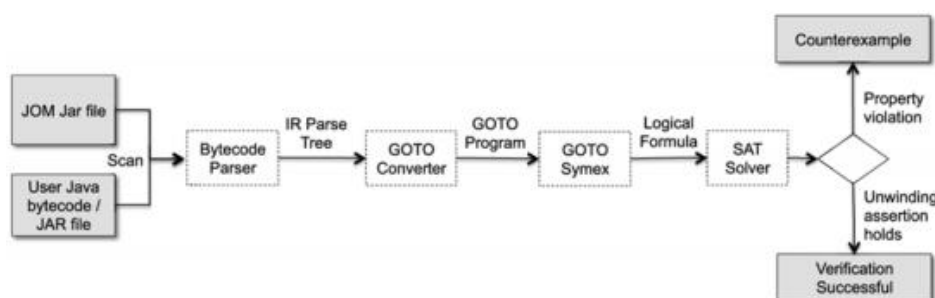


Figure 2.1: JBMC Operation Process

The JBMC architecture, which has three basic components, is depicted in the above diagram. The input is indicated by the grey rectangle on the diagram's left side. The JBMC verification stages are shown by the white rectangle in the centre. The output verification results are shown in the grey rectangle on the right.

JBMC first accepts an input of a JAR file or class file. The input file is subsequently parsed into a parse tree using a bytecode parser. The GOTO converter is then used to turn the parse tree into a GOTO programme. This conversion lowers the control flow of exceptions while also streamlining the Java bytecode representation. It then uses GOTO Symex to translate the GOTO programme into a Boolean logic formula, extending the loop and the calls to the recursive functions in accordance with the upper bound  $K$  (if the user sets the extension limit  $k$ ). The output of this operation is delivered to the SAT or SMT solver, which will check for mistakes. As a result, at the conclusion of the verification lifecycle, it produces an output [17]. The output so signifies the end of the file validation life cycle. JBMC's "validation successful" answer denotes that no issues were found. On the other side, if a programming mistake is found, JBMC states "validation failed" and provides a counterexample that shows the specific traits of the wrong state that may be reached in  $K$  steps.

JBMC won the gold prize in the SV-COMP '19 software verification competition, demonstrating the strength of its verification capabilities. JBMC still has flaws in a number of areas, though. First off, due to the restrictions of limited model checking, it is unable to check the state beyond the upper bound  $k$ . The route explosion problem might arise if the model checking threshold is surpassed, which would need a substantial amount of computing time and resources for the model checking tool to finish the research. Second, JBMC might not fully support all Java language features, libraries, and frameworks. This might make it less able to analyse some programmes, especially those that employ unusual or sophisticated features. Third, JBMC typically is unable to enable multi-threaded programme verification efficiently [18].

```

1 package utils;
2
3 public class test {
4     public static void main(String[] args){
5         recursion(i: 8);
6
7         assert false;
8     }
9     2 usages
10    public static void recursion(int i){
11        if(i<=0){
12            assert false;
13        }
14        if(i>0){
15            recursion(i: i-1);
16        }
17    }

```

Figure 2.2: A recursive program example

The recursive function in the programme will be run 8 times recursively in this code from Figure 2.2, and the programme will stop when the value of  $i$  is decreased to zero. Line 11 of the programme has a manually set assertion error. According to BMC theory, an assertion error is caused by the condition  $x=8$  if the recursive function is enlarged 5 times or fewer, but if it is expanded 10 times or less, an assertion error is recognised.

The results of the test were as expected, when the recursive function is run only 5 times, no program assertion errors are detected, while when the recursive function is run 10 times, program assertion errors are detected. Thus, experiments show that the program itself may be incorrect even if it is correct when extended  $k$  times.

## 2.4 Witness Validation

In comparison to software validation and software model checking, the notion of witness validation is quite recent. In all, six verifiers (including the two that were first submitted in 2015) were submitted to SVCOMP between 2015 and 2020. The process of verifying a witness produced by a software model checker that a particular programme violates that property is known as witness validation. This is done in order to prove that a property has been broken by giving a counterexample.

The objective of the study in this project is to identify software tools that do witness verification, whereas witness producers are software model checkers like CBMC, JBMC,

and other tools for creating witnesses [19]. Witness validation intends to employ an interchangeable format for representing witness data and raise the degree of confidence in validation findings produced by software verifiers.

There are two types of witnesses: witnesses to violations and witnesses to correctness. In violation witnesses, improper pathways or violations are described together with counterexamples that the software verifier can identify. Correctness witnesses, on the other hand, include witness information that outlines a proof of correctness and represents a potential event in which a violation might have happened but the software verifier missed it. This opens the door to the use of combinations of different verification methods and enables the verification findings to be independently validated with a standard witness format.

GraphML is a suggested format for witness validation that is interchangeable. In GraphML, an XML-based format, the states and behaviours of the attributes are represented by edges and nodes that are connected to one another to form a strictly directed graph [20]. The relative ease of reading and writing GraphML files up and down is one of the key reasons that GraphML was chosen as the preferred interchangeable file format. Client data can be defined and kept since GraphML can be extended. This enables the expression and storage of specific witness data, such as error routes. However, witnesses are not guaranteed to be absolutely valid because the verifier may produce incorrect results and the complexity of the format of the counterexamples, which are difficult to read by subsequent programs, drives the need to do witness validation.

### **2.4.1 Witness validation for Java**

Previously, since SV-COMP did not provide a stable witness verifier for Java programs, Vi attempted to validate witnesses for Java programs by developing a tool to validate the witnesses generated by the Java verifier. However, this tool had many limitations and drawbacks, and the tool was later improved by Tong.

The now widely used method for validating violating witnesses is wit4j developed by tong. the validation method is to check counterexamples, in particular to validate witnesses generated by JBMC [21]. Figure 2.10 shows the extended architecture, which incorporates

the processes of JBMC validation and witness verification.

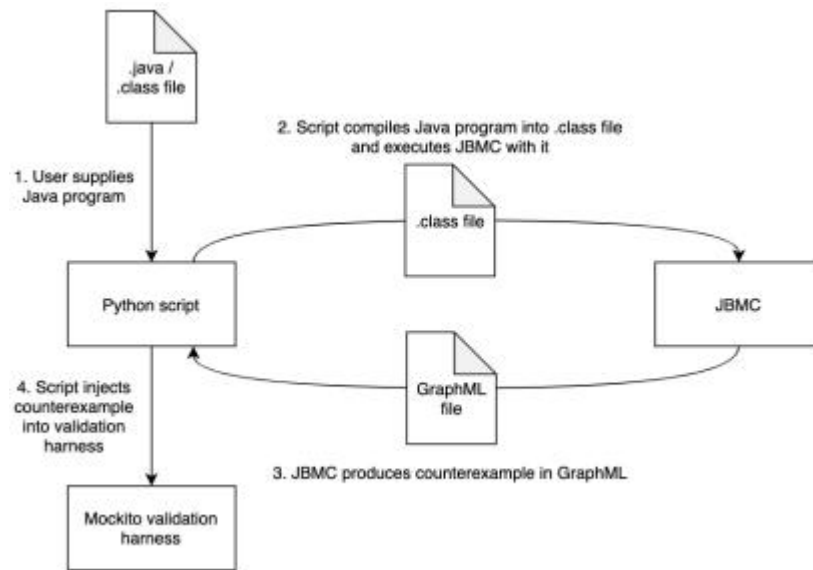


Figure 2.3: Vi's Project Operating Principles

In Figure 2.3, the core of this program run is a script written in Python. It accepts either a Java file or a compiled Java class file as input to the JBMC runtime, with subsequent actions varying by parameter type. Next, JBMC generates a complete GraphML file, and the Python script attempts to extract a counterexample from the witness formatted as GraphML. After obtaining the type and specific value of the counterexample, the script injects it into the validation harness to generate a Java unit test case. The final step is to execute the harness program to validate the counterexample through the machine control framework and then display the results at the terminal [22]. Specifically, in Java unit testing, the unit testing framework Mockito is introduced, which is able to simulate the methods in java classes ensuring that the user's desired values are returned when the generated methods of various 0 random data types are called. Here, the counterexample value is used as the return value of the function that generates the random value, thus assigning the counterexample value to the source program variable.

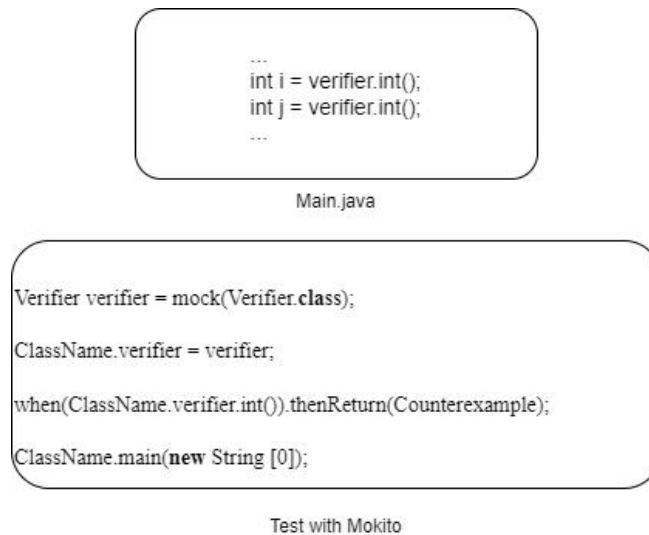


Figure 2.4: Validation Harness Injection Example

Figure 2.4 illustrates an example of a verification algorithm using Mockito. In the main.java program, *i* and *j* are variables of type *int*, and the verifier will call a function that can be a function that returns a random *int* type. When JBMC finds a violated property, the witness will give a counterexample for *i* or *j*. In witness verification, this class of methods is modeled by Mockito's *mock()* function, which returns the value of the counterexample in the witness when the *int()* function of this verifier is called. Thus, the assignment of the counterexample is equal to the *int i/j = counterexample* value of the java statement. The Java unit test programme is then executed using Mockito; if it is discovered to have the same error characteristics as those discovered by JBMC, it indicates that the counterexample is accepted and the validation is successful.

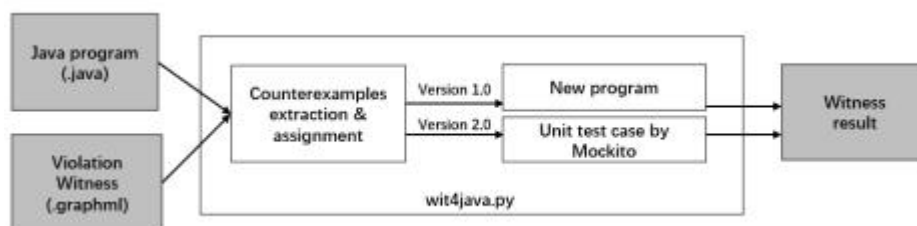


Figure 2.5: Wit4j version Comparison

Tong shows examples of both implementations in wit4j. Wit4Java 1.0 (kindergarten version) saves the counterexamples in witness programs in line number order. It generates a new program by directly replacing variable values in the source program. The disadvantages of this approach are that the code robustness is too weak, the code is not modularized well, and test failures occur from time to time. The use of Mockito is



encapsulated in Wit4Java 2.0 (which we named Mockito version) to validate the java file by generating test cases and return the counterexamples when the simulated function is called.

## 2.5 Other Java verification tools

In addition to JBMC there are many other Java verification tools capable of verifying java files, most notably Java Ranger, Java Pathfinder (JPF), Jdart, etc. One of the key objectives of JBMC when it was first developed was to exceed the performance of these tools. This section provides background on various Java verification tools and briefly discusses how they work and perform in software verification contests. The following figure shows a line graph of the scores obtained by each of the tools for verifying Java in running the benchmarks and the time taken to run them.

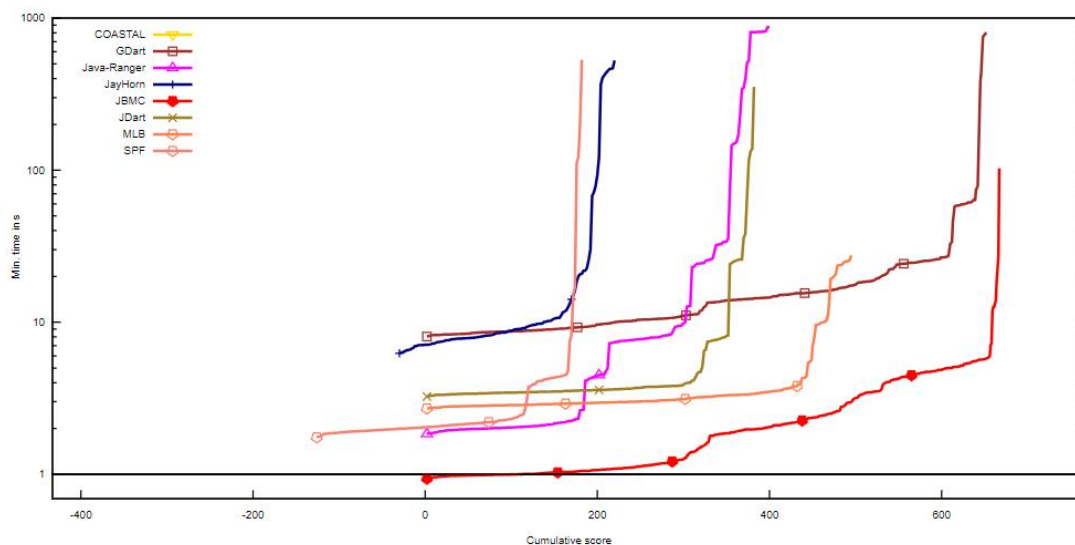


Figure 2.6: Effects of different bounded model checkers

JDart simulation's adaptability is leveraged by GDart, which also incorporates COASTAL's modular structure. The symbolic decision engine (DSE), the concolic actuator (SPouT), and the SMT solver backend (JConstraints) are the three separate parts of the Dynamic Symbolic Execution Engine, which enables meta-strategies to solve SMT issues. Through a recently developed communication protocol, the symbolic decision-making component is loosely connected to the actuators. On SV-COMP 2022, GDART completed 471 of the 586 jobs and discovered 302 more accurate wrong results than real right results. It placed fourth.

The Dynamic Symbol Enforcement tool in the SV-COMP benchmark tended to be more effective at identifying attribute violations than at confirming that none were present. This is somewhat intentional, since some of the questions were designed to assess how well candidates could manage broad and challenging state spaces. Overall, GDart completed 35 fewer tasks than JBMC and 5 more than Java Ranger.

The well-known Symbolic Pathfinder program's Java Ranger expansion broadens the verification methods for symbolic Java bytecode execution. Java Ranger outperforms SPF in nine benchmarks by reducing the runtime and the quantity of paths executed by 38% and 71%, respectively [23]. The SPF configuration is fairly similar to the Java Ranger setup. Given that the Java Ranger catalogue is merely an expansion of Symbolic PathFinder—the only configuration change—it can be said to be a legal addition to SPF. At a major theoretical conference, Java Ranger competed against the most advanced Java verifiers in a competition for static verification. With 630 points out of a possible 693 and 427 jobs completed out of a possible 473, Java Ranger won the competition's Java verification track [24].

At CMU and NASA Ames Research Centre, the JDart Java dynamic symbolic analysis framework has been in development. The main goal of JDart development is to offer a dynamic symbolic analysis tool for use in large-scale software, such as intricate NASA systems. The two most important parts of JDart are the executor and resource manager. The executor runs the software that has been through analysis and monitors any symbolic limitations on the values of the data. It is currently implemented using the Java PathFinder framework plug-in. The resource manager selects the exploration approach. It uses the JConstraints constraint library as an abstraction layer to efficiently encode symbolic path constraints and provides interfaces to several constraint solvers [25]. JDart was second only to Java Ranger in the SV-COMP 2021 with a score of 623 (out of 693) and proved its superiority by finishing 437 out of 473 task solvers in the Java track.

## **2.6 Summary of background**

In comparison to traditional software testing, the use of effective bounded checking models can, overall, play a crucial role in verifying software security-related elements.

Additionally, model checkers can be used in conjunction with conventional software testing techniques to identify software flaws and test scenarios that are challenging to manually recreate. The main obstacle to using bounded model checkers in practise, particularly for witness verification of Java programmes, is determining the reliability of the witnesses submitting their verification results. Currently, methods for confirming contradictory witnesses have also been put out by Tong et al.

# Chapter 3 Research Methodology

This chapter describes tools that extend the Java Validator to implement the Java Validator Correctness Witness. We add previously available algorithms and retool the implementation with appropriate framework choices in order to validate witnesses for Java programs using Python and Mockito. We describe the overall system architecture in Section 3.1. We will introduce the role that each part plays in the overall tool through flowcharts and text. Then, in Section 3.2, we will continue to delve into the algorithms implemented in the extension, including improvements to some of the algorithms already implemented by our predecessors and some new additions. In the last section of the chapter, Section 3.3, we will provide some examples to illustrate how the implemented extensions work.

## 3.1 System Architecture

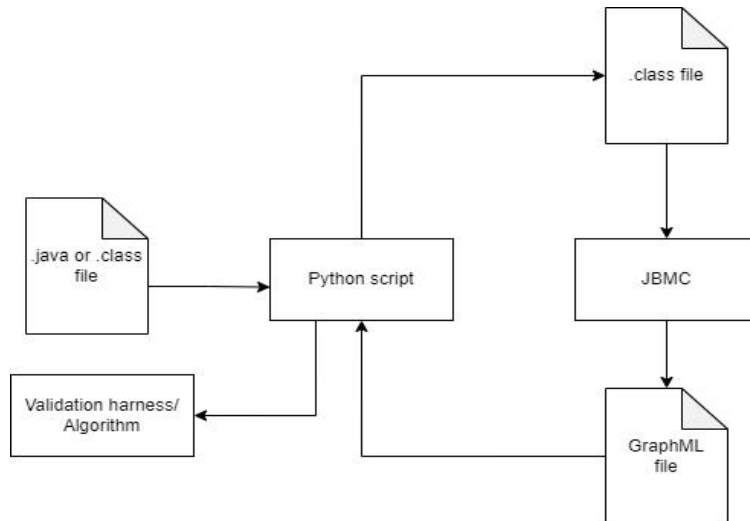


Figure 3.1 : Architecture of the proposed extension

Figure 3.1 shows the general structure and flow of events in the form of a flowchart. Through the terminal or console, the user will provide JBMC with parameters in the form of a .java file or a .class file, which is an important parameter for running JBMC, and JBMC will not run without the specific file. The Python script will oversee the entire operation, including the internal algorithms, build the validation harnesses, and report to the user the results of the accuracy witnesses, including a determination of whether or not

they are legitimate. The program's structure will be thoroughly explained in the sections that follow, along with the functions of each component and the precise steps involved in its execution.

In the first stage, the user needs to provide the Python script with the Java program to be tested. This step can be either a .java file or a .class file, depending on the user's preferences, there is no hard and fast rule. Depending on JBMC's requirements for input file types, the script needs to ensure that the Java program is in the required .class format. So, if the user submits a .java file, the python script will automatically compile it to create the required .class file. Therefore, just make sure that the file type entered in the next step is bytecode to satisfy JBMC's needs. The script will then execute JBMC via the CLI and run it using system commands.

JBMC can generate a format file called GraphML, which is an XML-based graphics file format, through an option. The inner core of such a file is a directed graph connecting all nodes, which can be displayed graphically. The commands are invoked directly from the command line interface, and we use python scripts to run the commands. The variables to which the bounded model check is applied get two arguments: the name of the bytecode file they were received in the previous phase, and the type of their raw data. The command line has three outcomes: if it returns "VERIFICATION FAILED," JBMC has determined that the programme is vulnerable; if it returns "VERIFICATION SUCCESSFUL," JBMC has determined that the programme is error-free; and the last outcome is an exception, execution timeout is the final outcome. In order to determine whether the programme has any assertion breaches, an automated Python script examines the output [26].

If the assertion is not violated, the Python script performs a series of validations using the GraphML file to determine if it is indeed a valid correctness witness. Here are the specific ideas for the validation and the order in which they are performed.

- Based on the nodes and edges in the GraphML file, determine if the flow from the entry node to the sink node is correct and if the edges and nodes are connected correctly. Due to the nature of directed graphs, every edge and adjacency to an edge should correspond.
- Determine if there are any loops in the generated GraphML, because if there are loops, the program is stuck in an infinite loop and will not terminate. Although there may be a

special case where the console outputs "validation successful" due to a limit on the number of times  $k$  can be expanded, the presence of a loop in itself is a sign that the program is vulnerable.

- If no violations are discovered, the design's validation tool is injected with one or more GraphML invariants. The major components of the validation tool are a Java-based validation harness and the Java class-simulation-capable Mockito simulation framework. The verification harness is used to execute the original programme after first simulating it. Additionally, we must rely on additional testing tools like Python to check whether the variables are correct or have the necessary values in order to identify any issues in the programme. The primary topic of study for this project is the GraphML file of the correctness witnesses since we are primarily concerned with validating the examples in the witness.

## **3.2 Algorithm**

The many algorithms used in this work are described in this section. The implemented algorithms and the inner workings of the Python scripts are described in depth in Section 3.2.1, and the usage of the algorithms to create a runnable validation harness and the use of Mockito to retrieve the actual file are also described in Section 3.2.2. The complexity, completeness, and resilience of the algorithms given in Subsections 3.2.1 and 3.2.2 are finally covered in Subsection 3.2.3.

### **3.2.1 Python script**

In this project, the first operation that will be performed is to run JBMC via python script to validate the Java programs. After checking the necessary Java programmes, it will start making every effort to carry out witness verification. The witness file associated with the Java programme will be read by the validation tool. It will carry out the witness verification if it reads a correctness witness. The validation findings will be output on the console at the end once all the witnesses have been verified, and they will be compared to those provided by JBMC. We use a programme called the starting script to automate this procedure. The startup script is a python programme that may be launched by typing the following command in a terminal window on Ubuntu or Windows:

If the user provides a .class file:  
python3 script.py File.class int

If the user provides a .java file:  
python3 script.py File.java

Both different approaches can be verified by running a script through python, and it is important to note that all Python files utilized in this project are Python3. The parameters that must be entered are separated by a space. Exec.py stands for the Python script that has to be executed, and File.java for the Java file that needs to be validated. The user has the option of providing either a .class file or a .java file for the third argument. The original data type of the variable to which the bounded model check is applied must be supplied as the fourth argument if the user opts to submit a .class file. This is necessary since it is currently challenging to acquire the contents of bytecode that has been compiled by the JVM using a python script, necessitating manual input from the user.

```
classnameArray = sys.argv[1].split('.')
classname = classnameArray[0]
if len(classnameArray) == 2 and classnameArray[1] == 'java':
    subprocess.Popen(['javac', sys.argv[1]]).wait()

if(classnameArray[1] == 'class'):
    type = sys.argv[2].lower()
```

Listing 3.1: Excerpt of script to get the file type

The code snippet in Listing 3.1 is able to get the class name of the program as it is required by JBMC. As mentioned earlier in subsection 3.2.1, this code segment will be able to handle both .java files and .class files. The user input parameters are used to identify whether the input file type is a .java file or a .class file. In the first case, the script will automatically translate it into a bytecode file that JBMC may use. For instance, if the user gives the script a file, say File.java, it will automatically compile that file into File.class, and if the file is a .class file, it will be validated by the script, it will also automatically capture the type of the provided value through a JBMC call.

```
cmd = 'jbmc ' + classname + ' --stop-on-fail --graphml-witness witness'
try:
    result = subprocess.check_output(cmd, shell = True)
except subprocess.CalledProcessError as e:
    result = e.output
```

Listing 3.2: Excerpt of script to execute JBMC

The argument `--graphml-witness` on the command line informs the system to produce the witness file in GraphML format, and the output witness's filename is the second parameter. The following stage is to determine whether there is any violation in the output result, mostly using the witness file created in the previous phases to determine whether the violation occurs or not. Usually, we read the witness file using the function `nx.read_graphml` from the special library to see if there is a conflict based on the nodes in the file.

```
witnessFile = nx.read_graphml("witness")
violation = False
for violationKey in witnessFile.nodes(data=True):
    if 'isViolationNode' in violationKey[0]:
        violation = True

if(violation == False):
    # This is the part we're focusing on.
else:
    print('Violation founded')
    exit(1)
```

Listing 3.3: Excerpt of script to check for violation

The GraphML witness file retrieved from the JBMC will be opened by the code in Listing 3.3. The script will go through each node for keys to violations as data is particularly represented in GraphML as nodes and keywords related conflicts will be written as node attributes. If a violation key is present, it means that a violation really took place. If, however, there was no violation, it means that the violation key was not included in the witness file. Because of this, we have to set the violation flag's default value to false; but, if the violation attribute is absent, as is the case in line 5, the violation flag will be set to a boolean value. Line 7 is where we decide whether or not the violation value from lines 1 through 5 is true. The script will immediately terminate the programme and notify the user that a conflict has been found in the case of a violation. If no violation is found, the script snippets in Sections 3.2.2 through 3.2.3 will still be executed even if the Boolean value will be "false". The details of these subsections are discussed in their respective subsections.



### 3.2.2 Directed graph checking

This section describes how to implement the first two methods in subsection 3.1 using algorithms, which may involve some classical algorithms. If we need to determine whether there is a loop in all nodes in GraphML, the first step we need to extract the data in GraphML. From the data structure in GraphML, it can be clearly seen that each node is linked through the edge, and the nodes corresponding to the source attribute and the target attribute in the edge represent the pointing relationship between the two nodes. First, extract all the nodes in the directed graph, and save their ids in an array or set (because there will be no duplicate ids). A key-value data structure should then be used to store the values that correspond to the source and target properties after extracting all of the edges using a loop.

A common way to determine whether a directed graph has a cycle is to use topological sorting. Topological sorting can sort a directed acyclic graph (DAG), that is, arrange all nodes in the graph into a linear sequence, so that for any directed edge  $(u, v)$ , the source node  $u$  is ranked in the target in the sequence in front of node  $v$ .

If there is a cycle in the graph, topological sorting cannot be performed, because the nodes on the ring depend on each other, and the order between them cannot be determined. Therefore, if there are nodes whose sequence cannot be determined when topologically sorting a directed graph, it means that there is a cycle in the graph.

In terms of specific implementation, topological sorting can be performed using depth-first search (DFS). First create a for loop to traverse each node. For each node, first mark it as "visiting", indicating that the node and its child nodes are currently being traversed. Then for each child node pointed to by this node, if the child node has been marked as "visiting", then there is a ring. If the child node has not been visited, the child node is traversed recursively. Finally, mark the node as "visited", indicating that the traversal of the node and its child nodes has been completed. When traversing other nodes, if it is found that the node has been marked as "visited", the node is skipped because the node has been traversed in DFS.

```
def CollatingData(file):  
    edgeDict = {}
```

```

num = 0
for node in file.nodes(data = True):
    if(node[1].get('isEntryNode') == True ):
        entryNode = node[0]
for data in file.edges(data = True):
    if(edgeDict.get(data[0]) == None):
        edgeDict.update({data[0]:data[1]})
    else:
        str = edgeDict.get(data[0]) + ',' + data[1]
        edgeDict.update({data[0]:str})
num = num + 1
return CheckIntegrity(entryNode,edgeDict,num,0)

```

Listing 3.4: Functions for organising data

The data set in the aforementioned code snippet Listing 3.4 is done in preparation for a subsequent integrity check. The initial node in the whole GraphML witness file, which is the entrance point of the complete witness, is sought after during the first traversal of all nodes. Next, a global dictionary is used to scan the edges in the GraphML witness file and record the nodes associated with each edge as key-value pairs. In directed networks, it is typical for one node to correspond to several nodes, which is a specific situation. As it can be seen here, the multiple nodes are divided by the "," sign before the update method updates the global dictionary with the associated key-value pairs.

```

def CheckIntegrity(node, edgeDict, num, cur):
    # Check the integrity of this witness file
    if node in edgeDict:
        val = edgeDict.get(node)
    else:
        if(cur == num):
            return True
        return False

    if(val == 'sink'):
        if(num == cur + 1):
            return True
        return False
    nodes = val.split(',')
    for node in nodes:
        ans = CheckIntegrity(node, edgeDict, num, cur + len(nodes))
        if(ans == False):
            return False
    return True

```

Listing 3.5: Functions for checking GraphML integrity

Listing 3.5 is intended to perform a recursive integrity check on the directed graph. The

initial node, which serves as the key for the dictionary check, points to the value discovered, which is the node pointed to. The node found is then used as the key for the secondary lookup and so on. If there have been as many iterations as the length of the dictionary by the time the loop reaches the final node, the validation is finished. If there are fewer iterations than the length of the dictionary, some nodes in the directed graph are isolated, and it is therefore possible to infer that the witness is insufficient.

```
def InspectionRing(node, edgeDict, arr):
    if node in edgeDict:
        arr.append(node)
        val = edgeDict.get(node)
        values = val.split(',')
        for val in values:
            if val in arr:
                return False
            ans = InspectionRing(val, edgeDict, arr)
            if ans == False:
                return False
    return True
```

Listing 3.6: Functions for checking the presence of a ring

The purpose of the above method is to check for the presence of loops in directed graphs (witness files in graphML format). As in Listing 3.6, the function accepts an initial node, a dictionary with the nodes connected by edges as key-value pairs and an array for storing the nodes that have been checked. The principle of recursion is utilized to find all nodes without interruption using DFS (depth-first search algorithm). Every time a node is found determine whether the node is already stored in the array, if yes then it means that there is a loop in the directed graph, immediately terminate the recursion and return a Boolean value False, the program terminates. If not, the node is stored in the array and the next recursion is performed. Note that the dictionary stores nodes in the same way as before, with multiple nodes separated by "," signs.

```
def GetType(argv):
    with open(sys.argv[1], "rt") as fin:
        type = []
        for line in fin:
            index = line.find('verifier.')
            if(index != -1):
                subStringIndex = line.find('(')
                variableType = line[index + 15:subStringIndex].lower()
                type.append(variableType)
```

```
return type
```

Listing 3.7: Functions for checking the presence of a ring

The verifier's data type is determined by the code fragment in Listing 3.7. This is done to make it easier to create valid validation harnesses from their templates, as one of the requirements is also the data type. Lines 1 and 2 in this section of the script will retrieve the verifier data type from the command parameters if the user supplies the.class file referred to in 3.2.1. We have made sure that the extracted data types' case consistency using the *lower* function. By doing this, mistakes in code execution brought on by inconsistent capitalization will be avoided. The script will go to the otherwise statement block as seen in lines 4 through 8 if the user delivers a .java file. Here, it will carefully examine the.java file to look for "verifier." Since it is quite likely that a .java file contains several verifiers of different sorts, it is necessary to maintain a global array to keep track of all the kinds. All the aforementioned operations are included in a function that produces an array of the extracted data types.

```
def GetInvariant(witnessFile, argv):
    className = argv[1][0:-4]
    arr = []
    for data in witnessFile.nodes(data=True):
        if 'invariant' and 'invariant.scope' in data[1]:
            invariant = data[1]['invariant']
            scope = data[1]['invariant.scope']
            if (invariant.startswith('anonlocal') and className + 'main' in
scope):
                Invariant = invariant.split(' = ')[1][: -1]
                arr.append(Invariant)
    return arr
def GetSeed(witnessFile):
    for data in witnessFile.nodes(data=True):
        if 'invariant' and 'invariant.scope' in data[1]:
            invariant = data[1]['invariant']
            scope = data[1]['invariant.scope']
            if (invariant.startswith('anonlocal') and 'createSeed' in scope):
                seed = invariant.split(' = ')[1][: -1]
                return seed
    return '0'
```

Listing 3.8: Functions for getting invariant in GraphML

Listing 3.8 explains the core code in the validation.py script. First, we count the number of instances that need to be extracted from the witnesses, since some benchmarks have multiple verifier values. The next operation is to extract all instances, each of which will

be injected into a statement template. These populated statement templates are then inserted one by one into the validation harness templates by design and converted into a Java program that verifies their validity based on the instances and data types. This step is called injecting instances into the validation harness. Lines 2-9 in Listing 3.8 show the extraction of the instances. In order to obtain the instances, we primarily read the witness file, look for the phrases "anonlocal" and "Invariant" describing the instances on their nodes, then divide them using the split function. Here, we utilise a list to hold numerous instances because certain benchmarks have multiple verifier values. Similarly, in order to retrieve a random seed from the witness file and restore the follow-me function, we must query the keyword "createSeed" from the instance's scope.

### 3.2.3 Validation harness creation

```
when(Verifier.Type()).thenReturn(Invariant);
```

Listing 3.9: Statement Template

```
import static org.mockito.Mockito.*;
import org.sosy_lab.sv_benchmarks.Verifier;

import static org.mockito.Mockito.*;
import org.sosy_lab.sv_benchmarks.Verifier;

public class ValidationHarness {
    public static void main ( String [] args ) {
        mockStatic(Verifier.class);

        // The statement is inserted here

        ClassName.main(new String [0]);
    }
}
```

Listing 3.10: Validation harness template built with Mockito

Listing 3.9 contains a code snippet that replaces the className, Type, and Invariant in a statement template before injecting it into the Validation Toolkit template. The Validation Toolkit template in Listing 3.10 was created using the Mockito framework. The template was created in Java because Mockito employs that programming language. The amount injected into the validation harness blocker when the statement template is fetched depends on how many variables the script discovers in the witness file. In order to confirm the reliability of the JBMC correctness witness, this will replicate the programme with

instances (effectively a static value). They are essentially text statements that have been converted to java files.

```
def StateCreation(type, Invariant, className, seed):
    with open("MockTemplate.txt", "rt") as file:
        line = file.read()
        line = line.replace('ClassName', className[0:-5])
        if(type == 'int'):
            line = line.replace('Type', 'nondetInt').replace(
                'Invariant', Invariant)
        if(type == 'short'):
            line = line.replace('Type', 'nondetShort').replace(
                'Invariant', Invariant)
        if(type == 'long'):
            line = line.replace('Type', 'nondetLong').replace(
                'Invariant', Invariant)
        if(type == 'float'):
            line = line.replace('Type', 'nondetFloat').replace(
                'Invariant', Invariant)
        if(type == 'double'):
            line = line.replace('Type', 'nondetDouble').replace(
                'Invariant', Invariant)
        if(type == 'string'):
            try:
                Invariant = int(Invariant)
                line = line.replace('Type', 'nondetString').replace(
                    'Invariant', 'null')
            except ValueError:
                line = line.replace('Type', 'nondetString').replace(
                    'Invariant', '"' + Invariant + '"')
        if(type == 'char'):
            line = line.replace('Type', 'nondetChar').replace(
                'Invariant', '\\' + chr(int(Invariant)) + '\\')
        if(type == 'boolean'):
            if(Invariant == '1'):
                line = line.replace('Type', 'nondetBoolean').replace(
                    'Invariant', 'true')
            if(Invariant == '0'):
                line = line.replace('Type', 'nondetBoolean').replace(
                    'Invariant', 'false')

        with open("MockStatement.txt", "w") as file:
            file.write(line)
```

Listing 3.11: Excerpt of script to create the validation harness from the template

The values are then inserted into the statement template above after getting all the necessary parameters. When injecting the statement from the template, the placeholders for



type, class name, and instance must all be changed. The class name alludes to the Java program's class name, and the type alludes to the verifier's data type. Most of the time, reading and editing files is the primary method used to replace placeholder values with the instances' actual values. This includes some primitive data types like strings, booleans, and strings, which necessitate a number of operations related to file reading and modification. The injected code statements are shown in Listing 3.11, and these injected statements are inserted into the validation toolkit. The exact number of insertions depends on the number of instances extracted from GraphML. The injected code statements are shown in Listing 3.11, and these injected statements are inserted into the validation toolkit. The exact number of insertions depends on the number of instances extracted from GraphML.

```
def HarnessRunning(types, Invariants, length, className, seed):
    for i in range(0, length):
        StateCreation(types[i], Invariants[i], className, seed)
        HarnessCreation(i+8)
    with open("ValidationHarness.txt", "rt") as fin:
        with open("ValidationHarness.java", "wt") as fout:
            for line in fin:
                line = line.replace('ClassName', className[0:-5])
                fout.write(line)
            subprocess.Popen(['javac', 'ValidationHarness.java']).wait()
            # Execute validation harness
            subprocess.Popen(
                ['java', '-ea', 'org.junit.runner.JUnitCore',
                'ValidationHarness']).wait()
```

Listing 3.12: Excerpt of script to running the validation harness

The compilation and execution of the validation harness are finished in Listing 3.12 after it has been constructed. It will use a Java compiler to compile the validation harness in order to produce a .class file that can be used to run the built validation harness. The results of the program's execution are output to the console as regular text output. After that, JBMC runs the compiled.class file once more to generate a fresh witness file and check for any conflicts. We can say that the correctness verification is true if there are no discrepancies between the two checks and the programme runs without giving any error messages.

### 3.2.4 Analysis about the Algorithms

In correlation with having logical judgments about the witness file, the complexity of the

algorithm is  $O(1)$ . The time complexity is always a constant value because the code in the document does not contain any loops, only the normal processing sequence. Since it is possible to see that there are several single loops in `exec.py`, which checks the witness file's integrity and the existence of loops, the complexity of the method for each function is linear and denoted by  $O(N)$ . Similarly, because it is necessary to navigate the complete GraphML file, the complexity of extracting the parameters with regard to the injection of the validation harness is also given as  $O(N)$ . Finally, there is a loop in the program `validation.py`, but it is two-tiered and takes care of the whole process of injecting the template statement arguments and creating the validation harness, thus, the time complexity of the program is  $O(N^2)$ .

### 3.3 Illustrative Examples

There are various instances of validation in this section. We investigated the many kinds of validation programmes that may be produced in each scenario and have chosen to highlight some typical instances from the SV-COMP benchmarks. Seven of Java's eight fundamental data types are shown in these examples: short, int, long, boolean, char, float, and double. We can get these benchmarks at <https://github.com/sosy-lab/sv-benchmarks> and they are open source and free.

#### 3.3.1 Int Example

```
/*
 * Origin of the benchmark:
 *   license: 4-clause BSD (see /java/jbmc-regression/LICENSE)
 *   repo: https://github.com/diffblue/cbmc.git
 *   branch: develop
 *   directory: regression/cbmc-java/assert1
 * The benchmark was taken from the repo: 24 January 2018
 */
import org.sosy_lab.sv_benchmarks.Verifier;

class assert1 {
    public static void main(String[] args) {
        int i = Verifier.nondetInt();
        if (i >= 10) assert i >= 10 : "my super assertion"; // should hold
        if (i >= 20) assert i >= 10 : "my super assertion"; // should hold
    }
}
```

Listing 3.13: Illustrative example of int Java program



A Java programme, Listing 3.13, invokes a validator class to produce data of the type int. It may be found under java/jbmc-regression/assert1.java in the sv-benchmarking folder. Line 5 of the code invokes the validator class, which gives the variable i a random int value. When either  $i \geq 10$  or  $i \geq 20$ , the validator class is invoked. When  $i \geq 10$  or  $i \geq 20$ , assert  $i \geq 10$ . In principle, this circumstance is not a mistake, hence the java file ought should execute without issue. To see if this conclusion holds true, we must examine both the JBMC findings and the script. In order to receive the preliminary JBMC findings, we first run the following command on the CLI.

```
jbmc assert1 --stop-on-fail
```

```

** Results:
[array-create-negative-size.1] Array size should be >= 0: SUCCESS
[array-create-negative-size.2] Array size should be >= 0: SUCCESS
[array-create-negative-size.3] Array size should be >= 0: SUCCESS
[array-create-negative-size.4] Array size should be >= 0: SUCCESS
[array-create-negative-size.5] Array size should be >= 0: SUCCESS
[array-create-negative-size.6] Array size should be >= 0: SUCCESS
[array-create-negative-size.7] Array size should be >= 0: SUCCESS
[array-create-negative-size.8] Array size should be >= 0: SUCCESS
[array-create-negative-size.9] Array size should be >= 0: SUCCESS
assert1.java function java:assert1.<clinit>:()V
[java:assert1.<clinit>:()V.null-pointer-exception.1] line 11 Null pointer check: SUCCESS

assert1.java function java:assert1.main:([Ljava/lang/String;)V
[java:assert1.main:([Ljava/lang/String;)V.1] line 13 no uncaught exception: SUCCESS
[java:assert1.main:([Ljava/lang/String;)V.assertion.1] line 15 assertion at file assert1.java line 15 function java:assert1.main:([Ljava/lang/String;)V bytecode-index 14: SUCCESS
[java:assert1.main:([Ljava/lang/String;)V.null-pointer-exception.1] line 15 Null pointer check: SUCCESS
[java:assert1.main:([Ljava/lang/String;)V.assertion.2] line 17 assertion at file assert1.java line 17 function java:assert1.main:([Ljava/lang/String;)V bytecode-index 27: SUCCESS
[java:assert1.main:([Ljava/lang/String;)V.null-pointer-exception.2] line 17 Null pointer check: SUCCESS

org/sosy_lab/sv_benchmarks/Verifier.java function java:org.sosy_lab.sv_benchmarks.Verifier.nondetInt:()I
[java:org.sosy_lab.sv_benchmarks.Verifier.nondetInt:()I.null-pointer-exception.1] line 31 Null pointer check: SUCCESS
[java:org.sosy_lab.sv_benchmarks.Verifier.nondetInt:()I.null-pointer-exception.2] line 31 Null pointer check: SUCCESS

** 0 of 17 failed (1 iterations)
VERIFICATION SUCCESSFUL

```

Figure 3.2: Screenshot of JBMC output of an int example

In Figure 3.2, it can be observed that the assert1 program passes the validation, as its output is "VERIFICATION SUCCESSFUL".

```

PS E:\project\2023master\correctnessVerifier\src\main\java> python3 exec.py assert1.java
stub class symbol java::java.lang.Object already exists
No violation founded
This correctness witness is complete.
This correctness witness does not have a ring.
['int']
['return_tmp0']
Completed validation, consistent with JBMC results
PS E:\project\2023master\correctnessVerifier\src\main\java>

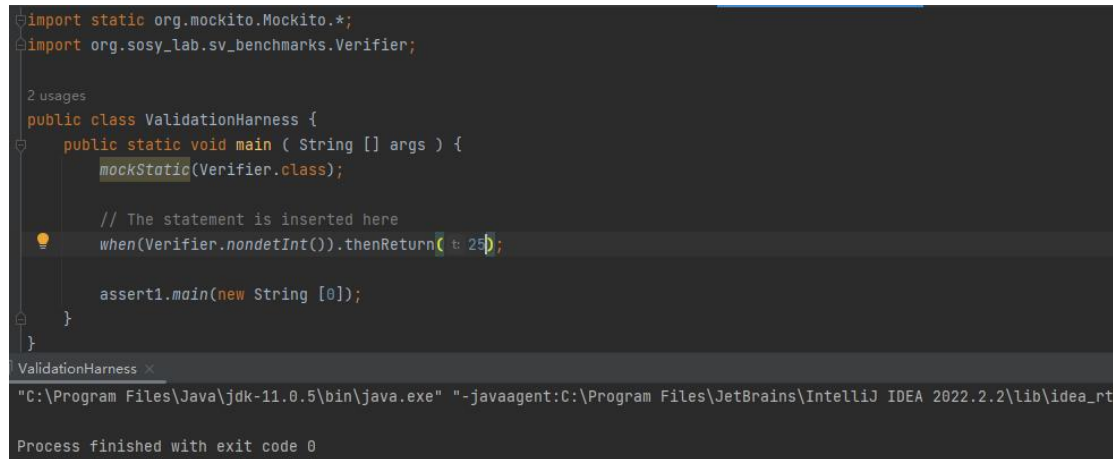
```

Figure 3.3: Screenshot of Python script output of an int example

The Python script's execution confirms that this correctness witness is true, as predicted. Run the Mockito-implemented verification harness to see if there are any issues; if there are none, the verification was successful. The CLI is used to run the Python script. The command that accomplishes this is:

```
python3 exec.py assert1.java
```

The screenshot shows the script's output, and the outcomes correspond to those from JBMC. No offensive remarks were discovered in any situation. Therefore, this indicates that the bug has been successfully validated and is a real one.



```
import static org.mockito.Mockito.*;
import org.sosy_lab.sv_benchmarks.Verifier;

2 usages
public class ValidationHarness {
    public static void main ( String [] args ) {
        mockStatic(Verifier.class);

        // The statement is inserted here
        when(Verifier.nondetInt()).thenReturn( 25);


        assert1.main(new String [0]);
    }
}

ValidationHarness x
"C:\Program Files\Java\jdk-11.0.5\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2022.2.2\lib\idea_rt
Process finished with exit code 0
```

Figure 3.4: Screenshot of the validation harness of an int example

A screenshot of the validation harness produced by the template-based script is shown in Figure 3.4. It is clear from this that the validation harness was appropriately created. The placeholder value is changed to the Java class name `assert1`, `nondetInt` is used in place of the verifier data type, and instance value `25` is injected precisely.

### 3.3.2 Long Example



```
/*
 * Origin of the benchmark:
 *   license: 4-clause BSD (see /java/jbmc-regression/LICENSE)
 *   repo: https://github.com/diffblue/cbmc.git
 *   branch: develop
 *   directory: regression/jbmc-strings/StringValueOf07
 * The benchmark was taken from the repo: 24 January 2018
 */
import org.sosy_lab.sv_benchmarks.Verifier;

public class StringValueOf07 {
    public static void main(String[] args) {
        long longValue = Verifier.nondetLong();
        System.out.printf("long = %s\n", String.valueOf(longValue));
        String tmp = String.valueOf(longValue);
        assert tmp.equals("100000000000");
    }
}
```

Listing 3.14: Illustrative example of long Java program

A benchmark test in Listing 3.14 uses a verifier class to generate data of the long type. The specific execution commands are shown below:

```
jbmc StringValueOf07 --stop-on-fail
```

StringValueOf07.java may be found in the sv-benchmarks directory at this path: java/jbmc-regression/sv-benchmarks/. The verifier class is invoked on line 5 of the code, which assigns a random long value to the variable long value. At line 7 of the code, the long value of the long type variable is converted to the string tmp using the ValueOf() function of the String class. In line 8, the converted string tmp is compared to the string "1000000000000". It goes without saying that an assertion error will occur if the produced long type random number is not equal to 1000000000000. As a result, the assertion statement will fall short as expected. As a consequence, Tong has finished the further verification and extraction of the counterexample in this example, and the JBMC execution result should be "Verification Failed". The JBMC partial log is displayed in Figure 3.5.

```
dynamic_object$18={ .@class_identifier="java::java.lang.Class" } ({ ? })
State 248 file StringValueOf07.java function StringValueOf07.main(java.lang.String[]) line 16 thread 0
-----
dynamic_object$18.@class_identifier="java::java.lang.Class" (?)
State 250 file StringValueOf07.java function StringValueOf07.main(java.lang.String[]) line 16 thread 0
-----
dynamic_object$18.@class_identifier="java::java.lang.AssertionError" (?)
State 253 file StringValueOf07.java function StringValueOf07.main(java.lang.String[]) line 16 thread 0
-----
this=&dynamic_object$18 (00000000 00011100 00000000 00000000 00000000 00000000 00000000 00000000)
Violated property:
file StringValueOf07.java function StringValueOf07.main(java.lang.String[]) line 16 thread 0
assertion at file StringValueOf07.java line 16 function java::StringValueOf07.main:([Ljava/lang/String;JV bytecode-index 25
false
```

Figure 3.5: Screenshot of JBMC output of a long example

We can see from the witness file that the counterexample's value, 0L, will be obtained by Song Tao or Tong's Python programme. We now print out "Violation founded" and go because the focus of this project is on correctness witness.

```
PS E:\project\2023master\correctnessVerifier\src\main\java> python3 exec.py StringValueOf07.java
stub class symbol java::java.lang.Object already exists
Violation founded
PS E:\project\2023master\correctnessVerifier\src\main\java> █
```

Figure 3.6: Screenshot of Python script output of a long example

In Figure 3.6, a Python script is executed on the CLI. The command executed is as follows.

As you can see from the screenshot, the output of the `StringValueOf07.java` script directly matches the results obtained by JBMC. In other words, both JBMC and the validator agree that there is a conflict in the java file. Therefore, this means that this is indeed a valid error and has been successfully validated.

```
<node id="97.179"/>
<edge source="97.179" target="100.207">
  <data key="originfile">StringValueOf07.java</data>
  <data key="startline">15</data>
  <data key="threadId">0</data>
  <data key="assumption">stub_ignored_arg0 = 0L;</data>
  <data key="assumption.scope">java</data>
</edge>
```

Figure 3.7: Screenshot of witness of a long example

In the previous implementation, the keyword "assume" was primarily looked up from the witnesses to locate the counterexamples and inject them into the verification harness. As shown in Figure 3.7, the counterexample that can be extracted in this example is `0L`, and the counterexample will be extracted and injected again. However, without going into too much detail here, we will only focus on correctness witnesses for now.

### 3.3.3 Short Example

```
import org.sosy_lab.sv_benchmarks.Verifier;
public class Short {
  static int field;
  static int field2;
  public static void main(String[] args) {
    int arg = Verifier.nondetShort();
    if (arg < 0) return;
    int x = arg;
    Short inst = new Short();
    field = arg;
    inst.test(x, arg, field2);
    // test(x,x);
  }
}
```



```

/* we want to let the user specify that this method should be symbolic */
/*
 * test IF_ICMPGT, IADD & ISUB bytecodes
 */
public void test(int x, int z, int r) {
    System.out.println("Testing ExSymExe14");
    int y = 3;
    r = x + z;
    x = z - y;
    z = r;
    if (z <= x) {
        System.out.println("branch F001");
        assert true;
    } else System.out.println("branch F002");
    if (x <= r) System.out.println("branch B001");
    else System.out.println("branch B002");
    // assert true;
}
}

```

Listing 3.15: Illustrative example of short Java program

The benchmark test in Listing 3.15 uses the verifier class to create data of the short type. It may be found in `java/jpf-regression/ExSymExe15_true.java` in the `sv` benchmarks folder. We disregard the lengthy remark area in the source code and only look at the programme code portion in this instance. Field and Field 2 are two member variables in the main class of the code. To assign a random short value to the variable field, you must first instantiate an object in `inst` and then call the function of the verifier class. An `if` statement in line 13 makes sure that the field's value is higher than or equal to 0. The parameter `r` has been given the value `x+z` during the execution of the test function, therefore in this case, `r` is equal to the sum of two values that must not amount to zero, and the result must be greater than zero. In addition, `z-y` must be bigger than `z+x` because `y=3`. Consequently, the claim ought to be accurate.

```

Short.java function java::Short.test:(III)V
[java::Short.test:(III)V.null-pointer-exception.1] line 49 Null pointer check: SUCCESS
[java::Short.test:(III)V.null-pointer-exception.2] line 55 Null pointer check: SUCCESS
[java::Short.test:(III)V.assertion.1] line 56 assertion at file Short.java line 56 function java::Short.test:(III)V bytecode-index 26: SUCCESS
[java::Short.test:(III)V.null-pointer-exception.3] line 56 Null pointer check: SUCCESS
[java::Short.test:(III)V.null-pointer-exception.4] line 57 Null pointer check: SUCCESS
[java::Short.test:(III)V.null-pointer-exception.5] line 58 Null pointer check: SUCCESS
[java::Short.test:(III)V.null-pointer-exception.6] line 59 Null pointer check: SUCCESS

org.sosy_lab.sv_benchmarks.Verifier.java function java:org.sosy_lab.sv_benchmarks.Verifier.nondetShort:()S
[java:org.sosy_lab.sv_benchmarks.Verifier.nondetShort:()S.null-pointer-exception.1] line 25 Null pointer check: SUCCESS
[java:org.sosy_lab.sv_benchmarks.Verifier.nondetShort:()S.null-pointer-exception.2] line 25 Null pointer check: SUCCESS

** 0 of 23 failed (1 iterations)
VERIFICATION SUCCESSFUL
PS E:\project\2023master\correctnessVerifier\src\main\java>

```

Figure 3.8: Screenshot of JBMC output of a short example

In this instance, we validated a flawless example. As a consequence, further validation will be carried out, including the injection of the validation harness and the verification of the JBMC results using Python scripts. The JBMC partial log is displayed in Figure 3.8.

```
PS E:\project\2023master\correctnessVerifier\src\main\java> python3 exec.py Short.java
stub class symbol java::java.lang.Object already exists
No violation founded
This correctness witness is complete.
This correctness witness does not have a ring.
['short']
['(int)return_tmp0', 'anonlocal::1i', '(void *)&dynamic_object$11']
Completed validation, consistent with JBMC result, the result is true
PS E:\project\2023master\correctnessVerifier\src\main\java> |
```

Figure 3.9: Screenshot of Python script output of a short example

The validation toolkit implemented by Mockito is run to determine if the results produced by the verifier are consistent with the JBMC and if there are no errors then the validation is successful. the Python script is executed on the CLI. This is done by with the following command:

```
python3 exec.py Short.java
```

The screenshot shows the script's output, and the outcomes correspond to those from JBMC. No offensive remarks were discovered in any situation. As a result, this indicates that the witness has been effectively validated and is a true witness to correctness.

```
2 usages
public class ValidationHarness {
    public static void main ( String [] args ) {
        mockStatic(Verifier.class);

        // The statement is inserted here
        when(Verifier.nondetShort()).thenReturn((short) 25);

        Short.main(new String [0]);
    }
}

ValidationHarness
"C:\Program Files\Java\jdk-11.0.5\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2022.2.2\lib\idea_
Testing ExSymExe14
branch F002
branch B001

Process finished with exit code 0
```

Figure 3.10: Screenshot of the validation harness of a short example

Figure 3.10 shows a screenshot of the validation harness generated from the template-based script. the Java class name is replaced with a placeholder value of "Short",

the verifier datatype is replaced with `nondetShort`, and the instance value `-13` is injected exactly as it is. note that the value here needs to be forcibly converted to the short type. In the final console, we can see that the values "branch FOO2" and "branch BOO1", which matches the result of the correctness verification.

### 3.3.4 Float Example

```
import org.sosy_lab.sv_benchmarks.Verifier;

public class ExSymExeFNEG_true {
    public static void main(String[] args) {
        float x = Verifier.nondetFloat();
        ExSymExeFNEG_true inst = new ExSymExeFNEG_true();
        if (x >= 0) inst.test(x);
    }
    public void test(float x) {
        System.out.println("Testing FNEG");
        float y = -x;
        if (y > 0) {
            assert false;
            System.out.println("branch -x > 0");
        } else System.out.println("branch -x <= 0");
    }
}
```

Listing 3.16: Illustrative example of float Java program

A benchmark test in Listing 3.16 uses a verifier class to create data of the type floating point. It may be found in `java/jpf-regression/ExSymExeFNEG_true.java` in the `sv_benchmarks` folder. On line 6 of the code, the verifier class is invoked, which gives the variable `x` a random float value. Line 8 invokes the `test()` function with the input `x` after line 7 instantiates the object `inst`. Note that here `x` satisfies the condition of being greater than or equal to zero. All is known that  $y < 0$  since the `test()` function assigns the floating-point variable `y` to the value `-x`. When `y` is less than or equal to 0, the if statement on line 14 states that it is untrue. However, if this doesn't happen, the statement is accurate.

```

ExSymExeFNEG_true.java function java::ExSymExeFNEG_true.main:([Ljava/lang/String;)V
[java::ExSymExeFNEG_true.main:([Ljava/lang/String;)V.1] line 32 no uncaught exception: SUCCESS
[java::ExSymExeFNEG_true.main:([Ljava/lang/String;)V.null-pointer-exception.1] line 33 Null pointer check: SUCCESS
[java::ExSymExeFNEG_true.main:([Ljava/lang/String;)V.null-pointer-exception.2] line 34 Null pointer check: SUCCESS

ExSymExeFNEG_true.java function java::ExSymExeFNEG_true.test:(F)V
[java::ExSymExeFNEG_true.test:(F)V.null-pointer-exception.1] line 38 Null pointer check: SUCCESS
[java::ExSymExeFNEG_true.test:(F)V.assertion.1] line 41 assertion at file ExSymExeFNEG_true.java line 41 function java::ExSymExeFNEG_true.test:(F)V bytecode-index 15: SUCCESS
[java::ExSymExeFNEG_true.test:(F)V.null-pointer-exception.2] line 41 Null pointer check: SUCCESS
[java::ExSymExeFNEG_true.test:(F)V.null-pointer-exception.3] line 42 Null pointer check: SUCCESS
[java::ExSymExeFNEG_true.test:(F)V.null-pointer-exception.4] line 43 Null pointer check: SUCCESS

org/sosy_lab/sv_benchmarks/Verifier.java function java::org.sosy_lab.sv_benchmarks.Verifier.nondetFloat:()F
[java::org.sosy_lab.sv_benchmarks.Verifier.nondetFloat:()F.null-pointer-exception.1] line 37 Null pointer check: SUCCESS
[java::org.sosy_lab.sv_benchmarks.Verifier.nondetFloat:()F.null-pointer-exception.2] line 37 Null pointer check: SUCCESS

** 0 of 21 failed (1 iterations)
VERIFICATION SUCCESSFUL

```

Figure 3.11: Screenshot of JBMC output of a float example

Figure 3.11 shows a portion of the code in the witness file generated by JBMC, which was executed via the command line:

```

jbmcc ExSymExeFNEG_true --stop-on-fail

```

The result of JBMC's execution is true, so there should be no counterexamples in the entire witness file, where its instance type is floating.

```

PS E:\project\2023master\correctnessVerifier\src\main\java> python3 exec.py ExSymExeFNEG_true.java
• stub class symbol java::java.lang.Object already exists
No violation founded
This correctness witness is complete.
This correctness witness does not have a ring.
['float']
['return_tmp0', '(void *)&dynamic_object$11']
Completed validation, consistent with JBMC result, the result is true

```

Figure 3.12: Screenshot of Python script output of a float example

In Figure 3.12, the Python script is executed on the CLI. This was done with the following commands.

```

python3 exec.py ExSymExeFNEG_true.java

```

The output of the StringValueOf04.java script can be seen in the screenshot, and the results directly match those obtained from JBMC. In both cases, the assertion statement was not violated. Therefore, this means that this is indeed a valid witness to success and has been successfully verified.



```

2 usages
public class ValidationHarness {
    public static void main ( String [] args ) {
        mockStatic(Verifier.class);

        // The statement is inserted here
        when(Verifier.nondetFloat()).thenReturn( + 33F);}

        ExSymExeFNEG_true.main(new String [0]);
    }
}

ValidationHarness x
"C:\Program Files\Java\jdk-11.0.5\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2022.2.2\lib\idea_
Testing FNEG
branch -x <= 0

Process finished with exit code 0

```

Figure 3.13: Screenshot of the validation harness of a float example

A screenshot of the validation harness produced by the template-based script is shown in Figure 3.13. You can see that the validation harness was appropriately created in this instance. The counterexample value "33F" was precisely injected, the verifier data type was changed to `nondetFloat`, and the placeholder value was changed to the Java class name `ExSymExeFNEG_true`.

### 3.3.5 Double Example

```

import org.sosy_lab.sv_benchmarks.Verifier;

public class Main {
    public static void ExSymExeD2L_true(String[] args) {
        double x = Verifier.nondetDouble();
        if (x >= 0.0 && x <= Integer.MAX_VALUE / 2) {
            ExSymExeD2L_true inst = new ExSymExeD2L_true();
            inst.test(x);
        }
    }
    public void test(double x) {
        int res = (int) ++x;
        if (res > 0) System.out.println("x >0");
        else {
            assert false;
            System.out.println("x <=0");
        }
    }
}

```

Listing 3.17: Illustrative example of double Java program

Listing 3.17 is a benchmark test that calls the verifier class to generate data of type double. In the code, the verifier class is called on line 5, which provides a random double value  $x$  for the boundary value of the variable, which is known to be positive from  $x \geq 0$ . In the called test method, the value of  $x$  is incremented and an error is asserted if the result is less than or equal to zero. Obviously, this does not happen, so the correct case is that at all times the value of  $x$  will be greater than 0. Therefore, the assertion statement will succeed as expected.

```
[java:ExSymExeD2L_true.main:([Ljava/lang/String;)V.1] line 32 no uncaught exception: SUCCESS
[java:ExSymExeD2L_true.main:([Ljava/lang/String;)V.null-pointer-exception.1] line 34 Null pointer check: SUCCESS
[java:ExSymExeD2L_true.main:([Ljava/lang/String;)V.null-pointer-exception.2] line 35 Null pointer check: SUCCESS

ExSymExeD2L_true.java function java:ExSymExeD2L_true.test:(D)V
[java:ExSymExeD2L_true.test:(D)V.null-pointer-exception.1] line 42 Null pointer check: SUCCESS
[java:ExSymExeD2L_true.test:(D)V.assertion.1] line 44 assertion at file ExSymExeD2L_true.java line 44 function java:ExSymExeD2L_true.test:(D)V bytecode-index 18: SUCCESS
[java:ExSymExeD2L_true.test:(D)V.null-pointer-exception.2] line 44 Null pointer check: SUCCESS
[java:ExSymExeD2L_true.test:(D)V.null-pointer-exception.3] line 45 Null pointer check: SUCCESS

org/sosy_lab/sv_benchmarks/Verifier.java function java:org.sosy_lab.sv_benchmarks.Verifier.nondetDouble:()D
[java:org.sosy_lab.sv_benchmarks.Verifier.nondetDouble:()D.null-pointer-exception.1] line 41 Null pointer check: SUCCESS
[java:org.sosy_lab.sv_benchmarks.Verifier.nondetDouble:()D.null-pointer-exception.2] line 41 Null pointer check: SUCCESS

** 0 of 20 failed (1 iterations)
VERIFICATION SUCCESSFUL
```

Figure 3.14: Screenshot of the validation harness of a double example

From the JBMC output we can see that the benchmark test `jpj-regression/ExSymExeD2L_true.java` of type double has been verified by JBMC and the result is true. As the result of JBMC validation tool is true, some columns of validation operations will be performed. The above Figure 3.14 shows a partial screen shot of the JBMC log.

```
PS E:\project\2023\master\correctnessVerifier\src\main\java> python3 exec.py ExSymExeD2L_true.java
stub class symbol java::java.lang.Object already exists
No violation founded
This correctness witness is complete.
This correctness witness does not have a ring.
['double']
['return_tmp0', '(void *)&dynamic_object$11']
Completed validation, consistent with JBMC result, the result is true
```

Figure 3.15: Screenshot of Python script output of a double example

The Python script is run using the CLI in Figure 3.15. This was accomplished with the subsequent command. The screenshot shows the `ExSymExeD2L_true.java` script's output, and the outcomes exactly match those from JBMC. This indicates that none of the claims are false in any scenario. As a result, this indicates that the witness has been effectively validated and is a true witness to accuracy.

```
import static org.mockito.Mockito.*;
import org.sosy_lab.sv_benchmarks.Verifier;

2 usages
public class ValidationHarness {
    public static void main ( String [] args ) {
        mockStatic(Verifier.class);

        // The statement is inserted here
        when(Verifier.nondetDouble()).thenReturn( t.43.0);

        ExSymExeD2L_true.main(new String [0]);
    }
}

ValidationHarness x
"C:\Program Files\Java\jdk-11.0.5\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2022.2.2\lib\idea_r
Testing FNEG
branch -x <= 0

Process finished with exit code 0
```

Figure 3.16: Screenshot of the validation harness of a double example

A screenshot of the validation harness produced by the template-based script is shown in Figure 3.16. It is evident that the validation harness was appropriately created. The counterexample value "43.0" was precisely injected, the placeholder value was changed to the Java class name ExSymExeFNEG\_true, and the verifier data type was changed to nondetFloat.

### 3.3.6 Boolean Example

```
/*
 * Origin of the benchmark:
 *   license: 4-clause BSD (see /java/jbmc-regression/LICENSE)
 *   repo: https://github.com/diffblue/cbmc.git
 *   branch: develop
 *   directory: regression/jbmc-strings/StringValueOf04
 * The benchmark was taken from the repo: 24 January 2018
 */
import org.sosy_lab.sv_benchmarks.Verifier;

public class StringValueOf04 {
    public static void main(String[] args) {
        boolean booleanValue = Verifier.nondetBoolean();
        String tmp = String.valueOf(booleanValue);
        assert tmp.equals("true");
    }
}
```

Listing 3.18: Illustrative example of boolean Java program

A verifier class is called by the Boolean benchmark test in Listing 3.18 to provide data of the Boolean type. `StringValueOf04.java` is located at the following path: `java/jbmc-regression/sv-benchmarks/`. Line 5 of the code calls the verifier class, which produces a random Boolean value for the variable's boundary value. The Boolean value of the variable of type Boolean is converted to the string `tmp` at line 6 of the code by calling the `ValueOf()` method of the String class. The transformed string `tmp` is compared to the Boolean value "T" in line 7. Naturally, if the generated random value of the Boolean type is false, an assertion error will take place. The assertion statement will therefore fail as anticipated.

```
State 161 file StringValueOf04.java function StringValueOf04.main(java.lang.String[]) line 15 thread 0
-----
dynamic_object$13.@class_identifier="java::java.lang.Class" (?)

State 163 file StringValueOf04.java function StringValueOf04.main(java.lang.String[]) line 15 thread 0
-----
dynamic_object$13.@class_identifier="java::java.lang.AssertionError" (?)

State 166 file StringValueOf04.java function StringValueOf04.main(java.lang.String[]) line 15 thread 0
-----
this=&dynamic_object$13 (00000000 00010100 00000000 00000000 00000000 00000000 00000000 00000000)

Violated property:
file StringValueOf04.java function StringValueOf04.main(java.lang.String[]) line 15 thread 0
assertion at file StringValueOf04.java line 15 function java::StringValueOf04.main:([Ljava/lang/String;)V bytecode-index 14
false
```

Figure 3.17: Screenshot of the validation harness of a boolean example

Due to the limitations of the test set, we did not find any Boolean-type benchmark tests without bugs in the jpf regression and jbmc regression test sets. In this case, we validated an example with one or more validation conflicts. Therefore, the result of the JBMC execution should be a "VERIFICATION FAILED" and subsequent validation operations to extract the counterexamples and witnessing tools should be performed. Some of the logs generated by JBMC are shown in Figure 3.17.

```
PS E:\project\2023master\correctnessVerifier\src\main\java> python3 exec.py StringValueOf04.java
stub class symbol java::java.lang.Object already exists
Violation founded
```

Figure 3.18: Screenshot of Python script output of a boolean example

As the screenshot shows, the validation will throw an exception and exit, and the console will output "Violation founded", which means that a conflict has been found.

```
<edge source="57.92" target="60.120">
  <data key="originfile">StringValueOf04.java</data>
  <data key="startline">14</data>
  <data key="threadId">0</data>
  <data key="assumption">stub_ignored_arg0 = false;</data>
  <data key="assumption.scope">java</data>
</edge>
```

Figure 3.19: Screenshot of witness of a boolean example

In the previous implementation, the keyword "assumption" was primarily looked up from witnesses to determine the location of the counterexample, which would be extracted and injected again into the validation harness. As shown in Figure 3.19, the counterexample for this java file is that an assertion error will occur when the `nondetBoolean` function returns a value of false. However, without going into too much detail here, we will only focus on correctness witness for now.

### 3.3.7 Char Example

```
import org.sosy_lab.sv_benchmarks.Verifier;
public class StaticCharMethods04 {
public static void main(String[] args) {
    char c = Verifier.nondetChar();
    assert Character.isLetter(c);
}
}
```

Listing 3.19: Illustrative example of char Java program

A benchmark test in Listing 3.19 generates data for a char type using a verifier class. In the `sv-benchmarks` subdirectory, it may be found at `java/jbmc-regression/StaticCharMethods04.java`. Line 5 of the code calls the verifier class, which gives variable `c` a random character value. The character class's `isLetter()` function is used at line 6 of the code to determine if the character `c` is a letter. It goes without saying that a false statement is made if the random character value is not a letter.

```
State 112 file StaticCharMethods04.java function StaticCharMethods04.main(java.lang.String[]) line 5 thread 0
-----
dynamic_object$11.@class_identifier="java:java.lang.AssertionError" (?)
State 115 file StaticCharMethods04.java function StaticCharMethods04.main(java.lang.String[]) line 5 thread 0
-----
this->dynamic_object$11 (00000000 00010000 00000000 00000000 00000000 00000000 00000000 00000000)
Violated property:
file StaticCharMethods04.java function StaticCharMethods04.main(java.lang.String[]) line 5 thread 0
assertion at file StaticCharMethods04.java line 5 function java::StaticCharMethods04.main:([[Ljava/lang/String;)V bytecode-index 10
false
VERIFICATION FAILED
```

Figure 3.20: Screenshot of witness of a char example



In the jpf regression and jbmc regression test sets, we were unable to locate any char type benchmarks without errors due to the test set's constraints. In this instance, cases with one or more validation conflicts were validated. As a result, "VERIFICATION FAILED" should be the outcome of the JBMC execution, and further validation procedures should be carried out to extract the witness tools and counterexamples. The JBMC partial log is displayed in Figure 3.20.

```
PS E:\project\2023\master\correctnessVerifier\src\main\java> python3 exec.py StaticCharMethods04.java
stub class symbol java::java.lang.Object already exists
Violation founded
PS E:\project\2023\master\correctnessVerifier\src\main\java> |
```

Figure 3.21: Screenshot of Python script output of a char example

As the screenshot shows, the validation will throw an exception and exit, and the console will output "Violation founded", which means that a conflict has been found.

```
<edge source="29.78" target="59.89">
  <data key="originfile">StaticCharMethods04.java</data>
  <data key="startline">4</data>
  <data key="threadId">0</data>
  <data key="assumption">anonlocal::1i = 60;</data>
  <data key="assumption.scope">java::StaticCharMethods04.main:([Ljava/lang/String;)V</data>
</edge>
```

Figure 3.22: Screenshot of witness of a char example

In previous implementations, the keyword "assumption" was primarily looked up from the witness to determine the location of the counterexample, which was then extracted and injected again into the validation toolkit. As shown in Figure 3.22, the counterexample for this java file is 60, and again in ASCII, the number 60 stands for "<". When the *nondetChar* function returns a value of "<", an assertion error will occur. However, without going into too much detail here, we just focus on correctness validation for now.

# Chapter 4 Experimental Evaluation

We will concentrate on the experimental assessment of the algorithms used in the "Research Methodology" chapter in this chapter. There are three sections in this chapter. The circumstances necessary to set up the experimental assessment environment are described in Section 4.1. The aims of our review are described in Section 4.2. We demonstrate the efficiency of our algorithm by displaying all validation results in Section 4.3, and we analyse these findings to identify the programme's advantages and disadvantages. Finally, we discuss the causes of our algorithm's shortcomings.

## 4.1 Setup

The environment in which we create and implement our algorithmic programmes is covered in depth in this section, together with the versions of the libraries, third-party tools, and languages that are needed. We also describe some of the libraries and dependencies, along with their versions, that are needed by the programmes in this part. With the exception of Java, there is no fixed sequence in which the different packages must be installed.

### 4.1.1 Environment installation for Java

The following official website is where you may get Java:

<https://www.java.com/en/download/manual.jsp>

Visit the following website to obtain the Java Development Kit (JDK):

<https://www.oracle.com/java/technologies/javase-downloads.html>

Here, at [https://www3.ntu.edu.sg/home/ehchua/programming/howto/JDK\\_Howto.html](https://www3.ntu.edu.sg/home/ehchua/programming/howto/JDK_Howto.html), you may find a thorough installation instructions for setting up the JDK on a variety of systems. The programme uses JDK version 11.0.5.

## 4.1.2 Environment installation for Python

Python is available for download at the following official site:

<https://www.python.org/downloads/>

Installing Python 3 is necessary for this project. Here you can find a thorough installation instructions for setting up Python on several platforms:

<https://realpython.com/installing-python/>

The additional third-party packages required by this Python script are as follows

(mandatory):

- subprocess
- sys
- networkx
- random

For a comprehensive guide on how to install Python packages, the links below may be helpful:

<https://packaging.python.org/tutorials/installing-packages/>

## 4.1.3 JBMC installation

JBMC supports multiple platforms including Ubuntu, macOS, Windows and Docker.

Different versions for different platforms can be found on the official GitHub. Below is an example for Windows only. The GitHub link for JBMC is shown below:

<https://github.com/diffblue/cbmc/releases>.

You can find details of the installation process in the README file in the Github repository, and click on "latest release" to get the latest version or the version that this project is running on. Then, finally add the JBMC PATH variable to point to the directory containing the JBMC executable as below:

```
PATH="C:\Program Files\cbmc\bin";%PATH%
```



## 4.1.4 Mockito installation

Mockito is available to download from the following site:

<https://mvnrepository.com/artifact/org.mockito/mockito-core>

If you want to use Mockito successfully, you need to download the other three dependencies separately, or just import Mockito-core if you are using a tool such as IDEA to install it via Maven. These dependencies are shown below:

- byte-buddy
- byte-buddy-agent
- objenesis

These dependencies can be found on the same site and can be seen under the "Compile Dependencies" section of the page. It lists the versions that are compatible with the selected version of the mockito core and the latest version of each dependency. Note that it is highly recommended that the Mockito version is the same as this project. Simulating static classes and methods is not supported in earlier versions of Mockito, and Mockito will be utilized several times in this project to simulate static classes.

## 4.1.5 Proposed extension

The code for this project has been uploaded to GitHub and is available for download at the following address :

<https://github.com/zaizai000816/JavaExtension>.

The zip file can be downloaded to a local folder or the project can be cloned to a local repository by cloning. The project contains 2 python scripts, 2 validation harness templates and several other files, including modified verifier classes.

## 4.1.6 Benchmarks

The complete sv-benchmarks folder is available from the following website:

<https://github.com/sosy-lab/sv-benchmarks>.

There are benchmarks for many languages in the sv-benchmarks folder. This project focuses on benchmark programs in the Java domain. In addition, my project files contain the entire sv-benchmarks folder and can be used directly. Note that sv-benchmarks may be updated during subsequent development, so you should download it in time.

### **4.1.7 Environment versions**

In this project, the software and its respective versions used are as below:

- JDK/JRE (Version 11.05)
- python3 (Version 3.8.3)
- CBMC/JBMC (Retrieved from GitHub repository on 29th July 2023)
- mockito-core (Version 5.2.0)
- byte-buddy (Version 1.14.1)
- byte-buddy-agent (Version 1.14.1)
- objenesis (Version 3.3)

### **4.1.8 Running the tests**

Before running the tests, information about the hardware used in this project is shown below:

- Model: MSI Laptop (2019)
- Processor: 2.6 GHz Intel Core i7-9750
- RAM: 16GB
- Operating System: Windows 10

The Python scripts and validation tool bundle templates must be located in the same directory as the benchmark files that will be put to the test. The specifics of the relative and absolute routes must be given if they are put on a different directory level. The project repository now includes a few submitted example benchmark files.

The experiment's execution was broken down into two key steps. The benchmark file must

first be tested using JBMC in the first step. The benchmark test file must first be built because it is a .java file. The following command will enable you to achieve this:

```
javac assert1.java
```

The next step will be to utilize the JBMC tool for initial validation of the file. This is done by running the compiled file with the following command:

```
jbmc assert1 -stop-fail.
```

If authentication fails, a log of the failure will be printed on the terminal or console. If the validation is successful, "Validation Successful" will be output on the console. Next, run the script with the benchmark using the following command:

```
python3 exec.py assert1.java
```

Specific output results will be printed on the console. All results for this experiment will be recorded and displayed in Table 4.1 in Section 4.3.

## 4.2 Objectives of the evaluation

The main objective of the experimental assessments in this chapter is to demonstrate the efficiency of the verification tools we devised and built for Java programme verification by displaying the outcomes of their use. We can more clearly describe the features and worth of the product by actually doing the tests.

The specific objectives regarding this experiment are as follows:

- Test our project by benchmarking it with SV-COMP and output the results of the implementation.
- Analyze the results of the experiment and give conclusions.
- Compare the results of the JBMC and my tool to determine the accuracy of the JBMC.
- Identifies the flaws that still exist in my extension tool and explains their causes.

## 4.3 Results

NO	Test suite	Title name	Type	Correct output	JBMC output	Script output	Comment
1	jbmc-regression/	ArithmeticException1	int	F	F	F	

2	jbmc-regression/	ArithmeticException5		T	T	T	No type founded
3	jbmc-regression/	ArithmeticException6	int	F	F	F	
4	jbmc-regression/	ArrayIndexOutOfBoundsException1	int	F	F	F	
5	jbmc-regression/	ArrayIndexOutOfBoundsException2	int	F	F	F	
6	jbmc-regression/	ArrayIndexOutOfBoundsException3	int	F	F	N/A	
7	jbmc-regression/	BufferedReaderReadLine	string	F	N/A	N/A	Execution time out
8	jbmc-regression	CharSequenceBug	string	F	N/A	N/A	
9	jbmc-regression/	CharSequenceToString	string	T	F	T	
10	jbmc-regression/	ClassCastException1		F	F	F	No type founded
11	jbmc-regression/	ClassCastException2		T	T	T	No type founded
12	jbmc-regression/	ClassCastException3		F	F	F	No type founded
13	jbmc-regression/	Class_method1		T	T	T	No type founded
14	jbmc-regression/	Inheritance1		T	T	T	No type founded
15	jbmc-regression/	NegativeArraySizeException1		F	F	F	No type founded
16	jbmc-regression/	NegativeArraySizeException2		F	F	F	No type founded
17	jbmc-regression/	NullPointerException1		T	F	T	No type founded
18	jbmc-regression/	NullPointerException2		F	F	F	No type founded
19	jbmc-regression/	NullPointerException3		F	F	F	No type founded
20	jbmc-regression/	NullPointerException4		F	F	F	No type founded
21	jbmc-regression/	RegexMatches01		T	N/A	N/A	Execution time out
22	jbmc-regression/	RegexMatches02	string	F	N/A	N/A	Execution time out
23	jbmc-regression/	RegexSubstitution01		T	F	T	No type founded
24	jbmc-regression/	RegexSubstitution02	string	F	F	F	Multiple verifiers
25	jbmc-regression/	RegexSubstitution03		T	F	T	No type founded
26	jbmc-regression/	StaticCharMethods01		T	T	T	No type founded
27	jbmc-regression/	StaticCharMethods02	string	F	F	F	
28	jbmc-regression/	StaticCharMethods03	string	F	F	F	
29	jbmc-regression/	StaticCharMethods04	char	F	F	F	
30	jbmc-regression/	StaticCharMethods05	string	F	F	F	Null pointer exception
31	jbmc-regression/	StaticCharMethods06	string	T	F	T	Null pointer exception
32	jbmc-regression/	StringBuilderAppend01		T	N/A	T	No type founded
33	jbmc-regression/	StringBuilderAppend02	string	F	F	F	Multiple verifiers
34	jbmc-regression/	StringBuilderCapLen01		T	F	T	No type founded
35	jbmc-regression/	StringBuilderCapLen02	string	F	F	N/A	
36	jbmc-regression/	StringBuilderCapLen03	string	F	F	N/A	
37	jbmc-regression/	StringBuilderCapLen04	string	F	F	N/A	
38	jbmc-regression/	StringBuilderChars01		T	F	T	No type founded
39	jbmc-regression/	StringBuilderChars02	string	F	F	N/A	
40	jbmc-regression/	StringBuilderChars03	string	F	F	N/A	

41	jbmc-regression/	StringBuilderChars04	string	F	N/A	N/A	Execution time out
42	jbmc-regression/	StringBuilderChars05	string	F	F	N/A	
43	jbmc-regression/	StringBuilderChars06	string	F	F	N/A	
44	jbmc-regression/	StringBuilderConstructors01	string	T	T	F	
45	jbmc-regression/	StringBuilderConstructors02	string	F	F	N/A	
46	jbmc-regression/	StringBuilderInsertDelete01		T	F	T	No type founded
47	jbmc-regression/	StringBuilderInsertDelete02	string	F	F	F	Multiple verifiers
48	jbmc-regression/	StringBuilderInsertDelete03	string	F	F	F	Multiple verifiers
49	jbmc-regression/	StringCompare01		T	F	T	No type founded
50	jbmc-regression/	StringCompare02	string	F	F	F	Multiple verifiers
51	jbmc-regression/	StringCompare03	string	F	F	F	Multiple verifiers
52	jbmc-regression/	StringCompare04	string	F	N/A	N/A	Execution time out
53	jbmc-regression/	StringCompare05	string	F	F	N/A	
54	jbmc-regression/	StringConcatenation01	string	T	F	F	Multiple verifiers
55	jbmc-regression/	StringConcatenation02	string	F	F	F	Multiple verifiers
56	jbmc-regression/	StringConcatenation03	string	F	F	F	Multiple verifiers
57	jbmc-regression/	StringConcatenation04	string	F	F	F	
58	jbmc-regression/	StringConstructors01		T	F	T	No type founded
59	jbmc-regression/	StringConstructors02	string	F	F	N/A	
60	jbmc-regression/	StringConstructors03	string	F	F	F	Multiple verifiers
61	jbmc-regression/	StringConstructors04	string	F	F	N/A	
62	jbmc-regression/	StringConstructors05	string	F	F	N/A	
63	jbmc-regression/	StringContains01	string	F	N/A	N/A	Execution time out
64	jbmc-regression/	StringContains02	string	F	F	N/A	
65	jbmc-regression/	StringIndexMethods01		T	T	N/A	No type founded
66	jbmc-regression/	StringIndexMethods02	string	F	N/A	N/A	Execution time out
67	jbmc-regression/	StringIndexMethods03	string	F	N/A	N/A	Execution time out
68	jbmc-regression/	StringIndexMethods04	string	F	F	N/A	
69	jbmc-regression/	StringIndexMethods05	string	F	N/A	N/A	
70	jbmc-regression/	StringMiscellaneous01		T	F	T	No type founded
71	jbmc-regression/	StringMiscellaneous02	string	F	F	N/A	
72	jbmc-regression/	StringMiscellaneous03	string	F	N/A	N/A	Execution time out
73	jbmc-regression/	StringMiscellaneous04		T	F	T	No type founded
74	jbmc-regression/	StringStartEnd01		T	T	T	No type founded
75	jbmc-regression/	StringStartEnd02	string	F	F	F	Multiple verifiers
76	jbmc-regression/	StringStartEnd03	string	F	F	F	Multiple verifiers
77	jbmc-regression/	StringValueOf01		T	F	T	No type founded
78	jbmc-regression/	StringValueOf02	string	F	F	N/A	
79	jbmc-regression/	StringValueOf03	string	F	F	N/A	
80	jbmc-regression/	StringValueOf04	boolean	F	F	F	
81	jbmc-regression/	StringValueOf05	string	F	F	F	

82	jbmc-regression/	StringValueOf06	int	F	F	F	
83	jbmc-regression/	StringValueOf07	long	F	F	F	
84	jbmc-regression/	StringValueOf08	string	F	F	F	
85	jbmc-regression/	StringValueOf09	string	F	F	F	
86	jbmc-regression/	StringValueOf10	string	F	F	N/A	
87	jbmc-regression/	SubString01		T	F	T	No type founded
88	jbmc-regression/	SubString02	string	F	F	N/A	
89	jbmc-regression/	SubString03	string	F	F	N/A	
90	jbmc-regression/	TokenTest01		T	N/A	N/A	Execution time out
91	jbmc-regression/	TokenTest02	string	F	N/A	N/A	Execution time out
92	jbmc-regression/	Validate01		T	F	T	No type founded
93	jbmc-regression/	Validate02	string	F	F	N/A	Multiple verifiers
94	jbmc-regression/	aastore_aaload1	int	T	N/A	N/A	Execution time out
95	jbmc-regression/	array1	int	T	N/A	N/A	Execution time out
96	jbmc-regression/	array2	int	T	T	T	
97	jbmc-regression/	arraylength1	int	T	T	T	
98	jbmc-regression/	arrayread1	int	T	T	T	
99	jbmc-regression/	assert1	int	T	T	T	
100	jbmc-regression/	assert2	int	F	F	F	
101	jbmc-regression/	assert3	int	F	F	F	
102	jbmc-regression/	assert4	int	F	F	F	
103	jbmc-regression/	assert5	int	T	T	T	
104	jbmc-regression/	assert6	int	T	T	T	
105	jbmc-regression/	astore_aload1		T	T	T	No type founded
106	jbmc-regression/	athrow1		F	F	F	No type founded
107	jbmc-regression/	basic1		T	T	T	No type founded
108	jbmc-regression/	bitwise1	int	T	T	T	
109	jbmc-regression/	boolean1	boolean	T	T	T	
110	jbmc-regression/	boolean2	boolean	T	T	T	
111	jbmc-regression/	bug-test-gen-095	string	F	F	N/A	
112	jbmc-regression/	bug-test-gen-119-2		T	F	T	No type founded
113	jbmc-regression/	bug-test-gen-119	boolean	T	F	T	Null pointer exception
114	jbmc-regression/	calc	int	T	F	T	Multiple verifiers
115	jbmc-regression/	cast1	int	T	T	T	
116	jbmc-regression/	catch1		T	T	T	No type founded
117	jbmc-regression/	char1	string	T	T	F	
118	jbmc-regression/	charArray	string	T	F	T	Null pointer exception
119	jbmc-regression/	classtest1		T	T	T	No type founded
120	jbmc-regression/	const1		T	T	T	No type founded
121	jbmc-regression/	constructor1		T	T	T	No type founded
122	jbmc-regression/	enum1		T	T	T	No type founded

123	jbmc-regression/	exceptions1		F	F	F	No type founded
124	jbmc-regression/	exceptions10		F	F	F	No type founded
125	jbmc-regression/	exceptions11		F	F	F	No type founded
126	jbmc-regression/	exceptions12		F	F	F	No type founded
127	jbmc-regression/	exceptions13		F	F	F	No type founded
128	jbmc-regression/	exceptions14		T	T	T	No type founded
129	jbmc-regression/	exceptions15		T	T	T	No type founded
130	jbmc-regression/	exceptions16	int	F	F	F	
131	jbmc-regression/	exceptions18		T	T	T	No type founded
132	jbmc-regression/	exceptions2		F	F	F	No type founded
133	jbmc-regression/	exceptions3		F	F	F	No type founded
134	jbmc-regression/	exceptions4		T	T	T	No type founded
135	jbmc-regression/	exceptions5		T	T	T	No type founded
136	jbmc-regression/	exceptions6		F	F	F	No type founded
137	jbmc-regression/	exceptions7		F	F	F	No type founded
138	jbmc-regression/	exceptions8		F	F	F	No type founded
139	jbmc-regression/	exceptions9		T	T	T	No type founded
140	jbmc-regression/	fcmpx_dcmpx1		T	T	T	No type founded
141	jbmc-regression/	iarith1		T	T	T	No type founded
142	jbmc-regression/	iarith2		T	T	T	No type founded
143	jbmc-regression/	if_acmp1		T	T	T	No type founded
144	jbmc-regression/	if_expr1	int	T	T	T	
145	jbmc-regression/	if_icmp1	int	T	T	T	
146	jbmc-regression/	ifxx1		T	T	T	No type founded
147	jbmc-regression/	instanceof1		T	T	T	No type founded
148	jbmc-regression/	instanceof2		T	F	T	No type founded
149	jbmc-regression/	instanceof3		T	T	T	No type founded
150	jbmc-regression/	instanceof4		T	T	T	No type founded
151	jbmc-regression/	instanceof5		T	T	T	No type founded
152	jbmc-regression/	instanceof6		T	T	T	No type founded
153	jbmc-regression/	instanceof7		T	T	T	No type founded
154	jbmc-regression/	instanceof8		T	T	T	No type founded
155	jbmc-regression/	interface1		F	F	F	No type founded
156	jbmc-regression/	java_append_char	boolean	F	F	T	
157	jbmc-regression/	lazyloading4		T	T	T	No type founded
158	jbmc-regression/	list1	int	T	T	N/A	
159	jbmc-regression/	long1		T	T	T	No type founded
160	jbmc-regression/	lookupswitch1	int	T	T	T	
161	jbmc-regression/	multinewarray		T	T	T	No type founded
162	jbmc-regression/	overloading1		T	T	T	No type founded
163	jbmc-regression/	package1		T	T	T	No type founded
164	jbmc-regression/	putfield_getfield1		T	T	T	No type founded
165	jbmc-regression/	putstatic_getstatic1		T	T	T	No type founded
166	jbmc-regression/	recursion2		T	T	T	No type founded

167	jbmc-regression/	return1		F	F	F	No type founded
168	jbmc-regression/	return2	int	F	F	F	Multiple verifiers
169	jbmc-regression/	store_load1		T	T	T	No type founded
170	jbmc-regression/	swap1		T	T	T	No type founded
171	jbmc-regression/	synchronized		T	F	T	No type founded
172	jbmc-regression/	tableswitch1	int	T	T	T	
173	jbmc-regression/	uninitialised1		T	T	T	No type founded
174	jbmc-regression/	virtual1		T	T	T	No type founded
175	jbmc-regression/	virtual2		F	F	F	No type founded
176	jbmc-regression/	virtual4		T	T	T	No type founded
177	jbmc-regression/	virtual_function_unwinding		T	T	T	No type founded

Table 4.1: Witness validation results

Table 4.1 shows the results of the experimental evaluation using the SV-COMP benchmark test. All together, there are six columns: "No", "Filename", "Type", "JBMC Output", "Script Output", and "Comments". The "Filename" column shows the filename of the benchmark test. The term "JBMC Output" refers to JBMC output, whereas the term "Script Output" refers to Python script output. Files that don't call the verifier class or call it frequently are inappropriate for evaluation since the verifier class is not supported. This is explained in the Comments section. "T" indicates that the validation was successful, "F" indicates that it was unsuccessful, and "N/A" indicates that the result was not available for the "JBMC Output" and "Script Output" columns. If the answer is "N/A," the "Comment" section will explain why the result was not possible to get.

Additional information on the outcomes is provided in the "Comment" section. If this column is empty, benchmarking with JBMC and scripts both produced identical results. Three comments in total should be added to the execution results. The first one reads "No type founded," indicating that verification techniques are not used in this specific benchmark test. The benchmark cannot be assessed if it does not make use of validation methods, as the suggested enhancement entails inserting static instance values into the simulated validation methods. The second option is "Multiple verifiers," which denotes that a certain benchmark test involves calling the verifier function while employing several verifiers. This enables the programme to function properly and produce a range of values for the algorithm's validation. The "type" column in this instance displays the type of the variable that was created when the method was initially invoked. The phrase "Execution timed out" indicates that the program's execution has expired. This is because it required too much time or memory to determine if the validation was effective. A "null pointer



exception" denotes a null pointer exception that JBMC found. In this situation, it indicates that no pertinent instance was added to the validation harness. Null pointer exceptions often happen most often with string types.

## **4.4 Threats to validity**

The JBMC validation findings and the output of the Witness Extension tool were as predicted for the bulk of the benchmarked programmes. Only a tiny portion of the findings were troublesome, primarily because of string types, and the tool's shortcomings will eventually be partially fixed. Based on the findings, it is evident that JBMC may generate inaccurate correctness witnesses when discovering programme flaws if particular benchmarking examples are not available. The absence of verifiers also makes it impossible for the script to appropriately assess the credibility of the witnesses for certain of the criteria. Similarly, The absence of byte type verification is one of the shortcomings of the experimental assessment phase. Since the benchmarks we used do not contain programmes that create byte types in the validation classes, the test set we used is to blame for this issue. Another restriction is the inability to validate several benchmarks simultaneously and provide a table of results, both of which will be optimised in a future step.

We have fixed some of these concerns compared to earlier enhancements for the Vi tool. Without making any modifications, we may immediately mimic static methods in classes and evaluate the verified benchmarks offered by SV-COMP. We validated the programme using one verifier and many verifiers for the categories of software that were verified, and we checked virtually all data types, including floating-point types and double-types.

## Chapter 5 Related Work

Since Java Bounded Model Checking (JBMC) has just recently begun to evolve as a field, there is still much room for improvement and expansion in this relatively young area. The JBMC algorithm's foundation, CBMC, has garnered interest since 2014 [27]. As a result, a lot of individuals have done research on and successfully used witness verifiers for C. A team from the University of Munich in Germany created MetaVal, a witness verifier for the programming language C. It was submitted to SV-COMP 2020 [28] and received favourable feedback. 3653 false testimonies and 16376 true testimonials were found throughout the verification procedure. Such outstanding outcomes attest to MetaVal's efficacy. Another witness verification technology submitted to SV-COMP 2020 is NITWIT, which was created by a team from RWTH Aachen University in Germany. Its verification time is quicker than that of rivals because of its minimal memory footprint [29].

A basic enhancement for already-existing verifiers for conflicting witnesses was previously optimised and implemented [30]. The major objective was to develop a MetaVal-like implementation that verifies witnesses produced by a software verification tool that serve as counterexamples. The implemented extension tool validates the accuracy of the flaws found by the Java program's software verifier and serves as evidence of the reliability of the program's witnesses [30]. They are presented to the user in a visual form and have been enhanced based on the early Vi, allowing them to execute the extraction of several counterexamples. Based on this, our current method of extracting the counterexamples and injecting them into the verification harness still involves identifying the instances in the witness file that may be used to replicate the java file that has to be checked. Nowadays, the methods and algorithms we have implemented are still relatively rudimentary, and there are many examples in benchmarks that we cannot test well. However, after research that further

validates the feasibility of witness verification methods for Java programs, research on correctness witnessing still needs to be done.

## Chapter 6 Conclusion

For software verifiers of Java programmes, this work has successfully presented an implementation of a witness verification extension that targets correctness witnessing. In particular, it has shown that it is possible to carry out witness verification for correctness witnesses using GraphML. Python and Mockito are mostly used in the implementation. The execution of the project has a number of constraints, which have been discovered and discussed in Chapter 4 of the experimental evaluation, and the next step is to be able to further optimise it. The related extended algorithm has been mostly put into practise. It has been able to demonstrate the potential of witness verification in Java software verification while being rather simple and unsophisticated. More than ever, there is a never-ending rush to identify and evaluate security flaws in order to prevent disastrous security breaches and assaults. Nowadays, software validation is crucial since software security is linked to personal property and data protection. Over time, software validation has gained popularity. JBMC is based on CBMC, which was created to verify Java programme vulnerabilities. CBMC, a more developed field, has seen numerous advances and extensions, including witness verification extensions that can perform well in SV-COMP. Witness verification-based enhancements for JBMC are still in the conceptual and initial implementation stages. Therefore, this extension to the implementation of the Java verifier designed to perform witnessed validation will hopefully support the research area surrounding this topic.

However, there are still some unresolved issues involved. The biggest technical fault is that our witness verification tool still does not handle string types, which is quite critical. So, exploring JBMC to address this issue is a noteworthy accomplishment. Additionally, our technology can be connected with other automated testing tools like BenceExec, which will enable us to carry out our verification duty more effectively.

Last but not least, I frequently discover that I am lacking in a variety of areas during the programme design and thesis writing phases. I was able to learn a lot about a lot of topics that I had never explored before thanks to the internet, and I learned a lot of new information from them. Future advancement in the subject of software verification may be more rapid, and I'll continue to monitor it, as well as learn and comprehend more about it.

# Bibliography

- [1] Prepbytes. (2023, March 20). Java Compilation Process. PrepBytes Blog.  
<https://www.prepbytes.com/blog/java/java-compilation-process/>.
- [2] GeeksforGeeks. (2023). Compilation and execution of a Java program.  
GeeksforGeeks. Available at:  
<https://www.geeksforgeeks.org/compilation-execution-java-program/>.
- [3] Pyp1.github.io. PYPL PopularitY of Programming Language index. [online]  
Available at: <https://pyp1.github.io/PYPL.html> .
- [4] Livshits, V. and Lam, M., 2005. Finding security vulnerabilities in java applications with static analysis. SSYM'05: Proceedings of the 14<sup>th</sup> conference on USENIX Security Symposium, 14, p.18.
- [5] Bekker, E., 2021. 2020 Data Breaches - The Most Significant Breaches of the Year | IdentityForce®. [online] We Aren't Just Protecting You From Identity Theft. We Protect Who You Are. Available at:  
<https://www.identityforce.com/blog/2020-data-breaches>.
- [7] PYPL. "PYPL Popularity of Programming Language."  
<https://pyp1.github.io/PYPL.html> (accessed 30 April, 2021).
- [8] Krill, P., 2021. 4 reasons to stick with Java -- and 4 reasons to dump it. [online] InfoWorld. Available at:  
<https://www.infoworld.com/article/2687995/4-reasons-to-stick-with-java.html>.
- [9] B. Beizer, Software testing techniques, 2nd ed. ed. Van Nostrand Reinhold (in eng), 1990.
- [10] Lu Luo. Software testing techniques: Technology Maturation and Research Strategy. Institute for Software Research International, Carnegie Mellon University, 2001.
- [11] Sv-comp.sosy-lab.org. 2021. SV-COMP 2021 - 10th International Competition on Software Verification. [online] Available at:  
<https://sv-comp.sosy-lab.org/2021> .

- [12] Cordeiro, L., Kroening, D. and Schrammel, P., 2019. JBMC: Bounded Model Checking for Java Bytecode - (competition contribution). Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, volume 11429 of Lecture Notes in Computer Science, pp.219-223.
- [13] Beyer D. (2021) Software Verification: 10th Comparative Evaluation (SV-COMP 2021). In: Grootte J.F., Larsen K.G. (eds) Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2021. Lecture Notes in Computer Science, vol 12652. Springer, Cham. Pp.401-422
- [14] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, "Bounded model checking," 2003.
- [15] Clarke, E. and Emerson, E., 1981. Design and synthesis of synchronization skeletons using branching time temporal logic. Logics of Programs, volume 131 of Lecture Notes in Computer Science, pp.52-71.
- [16] D'silva, Vijay, Daniel Kroening, and Georg Weissenbacher. "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.
- [17] L. Cordeiro, P. Kesseli, D. Kroening, P. Schrammel, and M. Trtik, "JBMC: A Bounded Model Checking Tool for Verifying Java Bytecode," Cham, 2018: Springer International Publishing, in Computer Aided Verification, pp. 183-190.
- [18] TACAS. "Results of the Competition."  
<https://sv-comp.sosy-lab.org/2021/results/results> verified/ (accessed August 25, 2021).
- [19] Cprover.org. n.d. JBMC – A Bounded Model Checking Tool for Verifying Java Bytecode. [online] Available at: <http://www.cprover.org/jbmc/>.
- [20] L. Cordeiro, D. Kroening, and P. Schrammel, "JBMC: Bounded Model Checking for Java Bytecode," Cham, 2019: Springer International Publishing, in Tools and Algorithms for the Construction and Analysis of Systems, pp. 219-223.
- [18] Svejda J., Berger P., Katoen JP. (2020) Interpretation-Based Violation Witness Validation for C: NITWIT. In: Biere A., Parker D. (eds) Tools and Algorithms for

- the Construction and Analysis of Systems. TACAS 2020. Lecture Notes in Computer Science, vol 12078. Springer, Cham. Pp40-57
- [21] Graphml.graphdrawing.org. n.d. The GraphML File Format. [online] Available at: <http://graphml.graphdrawing.org/>.
- [22] D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, and A. Stahlbauer, "Witness validation and stepwise testification across software verifiers," presented at the Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, Bergamo, Italy, 2015. [Online]. Available: <https://doi.org/10.1145/2786805.2786867>.
- [23] V. L. Tan, "SECURITY ANALYSER TOOL FOR FINDING VULNERABILITIES IN JAVA PROGRAMS," THE UNIVERSITY OF MANCHESTER, 2020. [Online]. Available: [https://ssvlab.github.io/lucascordeiro/supervisions/msc\\_thesis\\_vi.pdf](https://ssvlab.github.io/lucascordeiro/supervisions/msc_thesis_vi.pdf).
- [24] Tan, V., 2020. Security analyser tool for finding vulnerabilities in Java programs. University of Manchester.
- [25] Mockito. "Tasty mocking framework for unit tests in Java." <https://site.mockito.org/> (accessed 2 May, 2021).
- [26] GitHub. n.d. GitHub - vaibhavbsharma/java-ranger: Java Ranger is a path-merging extension of Symbolic PathFinder. [online] Available at: <https://github.com/vaibhavbsharma/java-ranger>.
- [27] Sharma, V., Hussein, S., Whalen, M., McCamant, S. and Visser, W., 2020. Java Ranger: statically summarizing regions for efficient symbolic execution of Java. Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering,.
- [28] Beyer D. (2021) Software Verification: 10th Comparative Evaluation (SV-COMP 2021). In: Groote J.F., Larsen K.G. (eds) Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2021. Lecture Notes in Computer Science, vol 12652. Springer, Cham. Pp.401-422
- [29] Luckow K. et al. (2016) JDart: A Dynamic Symbolic Analysis Framework. In:



Chechik M., Raskin JF. (eds) Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2016. Lecture Notes in Computer Science, vol 9636. Springer, Berlin, Heidelberg.

- [30] D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, and A. Stahlbauer, "Witness validation and stepwise testification across software verifiers," presented at the Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, Bergamo, Italy, 2015. [Online]. Available: <https://doi.org/10.1145/2786805.2786867>.