MANCHESTER
1824
The University of Manchester

Third-Year Project

Understanding Programming Bugs in Java Programs Using Counterexamples

Using Bounded Model Checker for Java Bytecode (JBMC) results to generate intuitive, able-to-be-run, and humanly-readable counterexamples for detected bugs

---

Submission Year: 2024

Author: Stefan Tatu

Supervisor: Dr. Lucas Cordeiro

Faculty of Science and Engineering
The School of Engineering
Department of Computer Science

Contents

Abstract

This project presents an innovative approach to enhance the debugging process of Java programs by leveraging the Java Bounded Model Checker (JBMC) to generate apparent and executable counterexamples for identified bugs. The core objective is to bridge the gap between complex bug detection outputs provided by JBMC and the practical needs of developers for understandable and actionable insights into errors within their code. By developing a user-friendly wrapper around JBMC, this project introduces a tool that simplifies the error analysis process, enabling Java developers to pinpoint, understand efficiently, and correct application bugs.

The project consists of a wrapper that automatically processes the output of JBMC, transforming detailed verification data into executable counterexamples that precisely highlight the conditions under which bugs occur. Through a comprehensive set of tests, including methodically constructed cases and applications on real-world Java classes, the effectiveness of this approach in identifying a wide range of programming errors was thoroughly evaluated. The results demonstrate the tool's capacity to uncover various types of bugs, from straightforward arithmetic mishaps to more intricate algorithmic faults, thereby significantly aiding in the debugging process.

Moreover, the project explores the tool's practical application in real debugging scenarios, engaging Java engineers in case studies that showcase the utility of the generated counterexamples in speeding up the bug-fixing workflow.

In conclusion, this project makes a significant advancement in the realm of Java debugging by automating the generation of intuitive counterexamples from JBMC output, effectively helping developers understand and quickly resolve programming errors. This contribution significantly improves debugging efficiency, offering a practical solution to the often complex and time-consuming task of bug fixing in Java applications.

Acknowledgements

I would want to thank my supervisor, Dr. Lucas Cordeiro, for his knowledge, support, and guidance throughout this project. With his feedback and advice, the project was able to achieve all the requirements of the wrapper developed, which are presented in the subsequent chapters.

Moreover, I acknowledge the contribution of the five Software Engineers from my work team and twelve fellow Third-Year Students who participated in the two case studies presented in this report. Without their contributions, this paper's testing and conclusions would not have been achieved to the level presented.

I would like to express my gratitude for the technological aids that contributed to the phrasing and clarity of this report. Specifically, AI tools and Grammarly played instrumental roles in refining the language used, and ensuring the articulation of my ideas has been clear and precise, especially as a non-native English speaker. It is important to note that while these tools assisted in language enhancement, the conceptualisation, research, and substantive content of this report were solely my own creation, without any ideas being generated by AI.

1 Introduction

1.1 Projected Proposal

According to a survey [1] conducted in June 2023, Java is used by 33.55% of the 87,585 respondents. This shows that this programming language is still widely utilised, mainly due to its robustness, learnability and reliability [2]. This popularity results in more complex programs being built, therefore conquering more possible bugs that can not be easily realised and discovered using traditional debugging methods and testing. Therefore, tools such as the Bounded Model Checkers (BMC) were developed to ease this process of detecting and understanding introduced bugs.

Even if results produced by such BMC can help the user identify if the program has any potential bugs or errors [3], it is often hard to actually understand the bug, especially in large-scale codebases, and in the specific scope of the project, Java classes or methods [4]. So, this project will look at the benefit of producing counterexamples for the bugs detected by the BMC model in order to develop a humanly readable tool for bug understanding.

This project will investigate how to manipulate the results produced by the Java Bounded Model Checker (JBMC), based on running the checker with various settings, in order to be able to produce a Java runnable counterexample for every bug detected so that the user will be helped in the debugging process.

1.2 Motivation

During my placement internship and following the part-time work completed after the year in the industry, I mainly worked in Java systems, having to write complex code for a plethora of tasks. Therefore, at times, I spent ample time debugging parts of my task on bugs that, even if evident, were challenging to locate and resolve due to the complexity and size of the code. Therefore, this is the reason why this project flared my interest: I acknowledge that for Java programmers, a tool such as JBMC, associated with a wrapper to construct more intuitive counterexamples, would facilitate the process of debugging for various cases, and would help the developer not only to resolve the bug but, to firstly, efficiently understand it. This makes the project exceptionally appealing and engaging, as it can serve as a helper tool for any Java developers in the future, and I can also use it in a professional setting within my job.

1.3 Aims and Objectives

In the context of this project, the definition of a more intuitive counterexample is a program that is able to call the original program and replicate a detected bug. The aim is to deliver a wrapper to get an input Java file, run it through the JBMC, take and process the results, and generate an executable (Java file) counterexample for every bug witnessed. If a counterexample is run, it should call the initial input file and replicate the bug to facilitate the user's debugging process.

Therefore, in order to achieve this aim, the project has the following objectives:

- Understand the Type of Bugs Detected by the JBMC and its limitations.
- Interpret the output produced by the JBMC based on the settings specified.
- Process the XML formatted output produced by the checker and use it to assemble a counterexample executable Java file for every detected bug.
- Include a verbose description of the bug in every produced counterexample for more straightforward human interpretation.
- All the project's functionality is to be contained in a user-friendly wrapper that is able to take the file as input, run it through the JBMC (with different specified settings), interpret the output and produce the counterexamples in the required format.

1.4 Summary of Evaluation Strategy

The evaluation strategy would be based on a range of Java input file tests, which will aim to analyse the counterexample generations of the method based on the capabilities of the JBMC [5]–[7]. The tests would range from simple cases to more complex ones used in automated software verification tools competitions where JBMC participates, as well as analysing open-source code. Moreover, two case studies will be conducted to assess if counterexamples aid the debugging process for Java Engineers and Students.

1.5 Report Structure and Planning

This report is split into six distinct chapters, including the Introduction (Chapter 1) and the Conclusion and Reflection Chapters (Chapter 6). The second Chapter focuses on setting the background knowledge needed to understand the project, including, but not limited to, an overview of bounded model checkers, an explanation of the JBMC tool, and reasoning why counterexamples might be beneficial when debugging. The third and fourth Chapters concentrate on the implementation of the design and developmental aspects of the software, including how the trace produced by the JBMC is processed in order to create viable and relevant counterexamples, as well as the testing methodologies used in assessing the performance of the developed software tool (including simple cases, SV-COMP Benchmarks - SV-COMP [8], [9] is an annual contest between automated software verification tools -, and a large-scale open-source code). The fifth Chapter will revolve around two case studies performed for this report, which should exemplify whether producing counterexamples supports debugging scenarios for professionals working in Java and for students with medium Java knowledge and expertise. The last Chapter will tackle not only conclusions but also reflections on the next steps for the project and other approaches that could have facilitated and enhanced the software implementation.

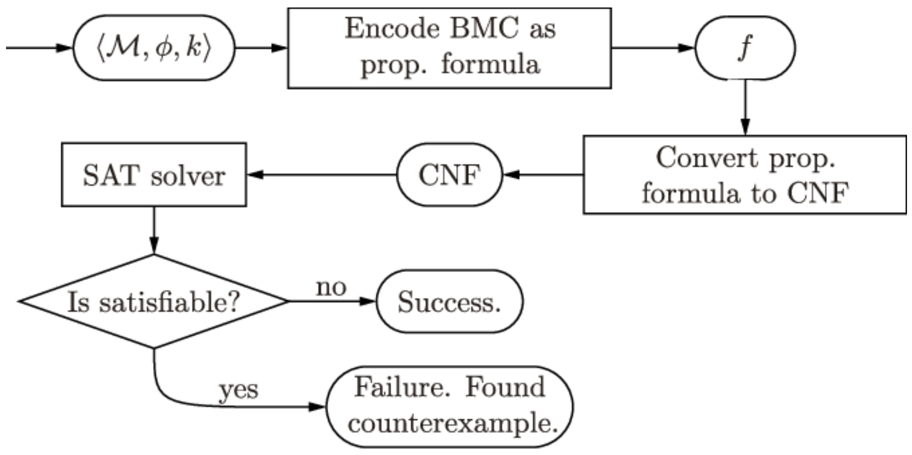
2 Background

2.1 Bounded Model Checkers (BMC)

Bounded Model Checkers (BMC) involve a formal verification technique developed to check the correctness of system models up to a defined number of steps or bounds [10]. Due to its bounded nature, it is especially effective in identifying errors or bugs in software and hardware systems, providing a more feasible method than exhaustive verification methods (e.g., traversing every potential path through a state machine to verify properties or determine correctness). The first step of a BMC model is to transform the given software or piece of code into a formal representation that efficiently encapsulates the possible states of a system and the transitions among them. One of the main such representations is Kripke structures [11], which incorporate states, transitions between those states, and labelling functions to represent properties contained in states.

A specific characteristic of BMC is its approach to problem-solving, where it transforms the verification task into a propositional satisfiability (SAT) or Satisfiability Modulo Theories (SMT) problem [12], [13]. This includes encoding both the system model within the defined bound and the negation of the property to be proved. Hereafter, an SAT or SMT solver processes this problem, with a successful solution indicating a property violation. When the solver detects a violation, the output can be analysed and processed in order to facilitate the understanding of the bug or issue found by the checker.

One important step in the BMC process of detecting correctness is property specification. The properties to be verified are defined as the expected behaviour of the system over time, using a formal specification language, such as linear temporal logic (LTL), to describe various time-dependent conditions [14].



$\langle \mathcal{M}, \phi, k \rangle$
Encode BMC as prop. formula
f
Convert prop. formula to CNF
CNF
SAT solver
Is satisfiable?
no
Success.
yes
Failure. Found counterexample.

Fig. 1. Example of BMC Flow - Source [Online] – Available at: `https://www.researchgate.net/figure/Diagram-for-classical-Bounded-Model-Checking-If-we-directly-define-the-semantics-of-G_fig2_258649981`

BMC confers a wide range of advantages [15], [16], such as being, in most cases, a highly automated process that needs from the user just the application and the definition of various properties and settings depending on the model (such as ways of running or specifying the limit bound). Moreover, most of the checkers produce an output (often called counterexamples) which can be further processed or analysed, which can foster insight into the issues discovered by the checker.

As common disadvantages [15], [16], it is worth mentioning that BMC is inherently incomplete, and would check only within the bound specified, and cannot guarantee correctness beyond that. Furthermore, even if more efficient than exhaustive checkers, it can still be resource-intensive to solve, especially for large or complex problems or big bounds.

2.1.1 SAT and SMT Solvers

Boolean satisfiability (SAT) – the problem of determining whether a given Boolean formula can be satisfied by some truth assignment to its variables [17] – is among the most fundamental problems in computer science. In BMC, SAT solvers play a pivotal role: their goal is to verify whether there is any sequence of a certain model's system transitions (at most a given bound or length) that leads to a violation of the given property. In BMC, the system model and the negation of the property are transformed into a Boolean formula. If the SAT solver finds an assignment that makes this formula true, this means that the model allows a counterexample to the property within the given bounds, meaning that a bug was found within the model [12], [13].

Satisfiability Modulo Theories (SMT) extends the problem of SAT [17] and adds theories such as arithmetic, bit-vectors and arrays to allow the representation of more expressive descriptions of system behaviours and properties. SMT solvers are critical to BMC if the description of the property expresses rich data types or needs to reason about the system's state in any way other than simple Boolean properties, aiding in the identification of subtle bugs that might elude simpler, Boolean-only analyses [13].

2.1.2 Linear Temporal Logic

A crucial element for verifying specifications about properties of an intended behaviour of a system model is Linear Temporal Logic (LTL) [18]. This is logically based on events over a path, which is a sequence of events over time. The primary temporal operators enable expressions about states in the future of a state, specifying whether a condition is true without fixing the time-point. For instance, the global operator (G) asserts that a condition holds in all future states, while the eventual operator (F) stipulates that a condition will hold at some point in the future. These operators, along with others like X (next) and U (until) [19], supply a rich language for describing the intended behaviours and restrictions of the system under observation.

These temporal properties specify what the system should ensure correctness over a bounded region. When encoded together with the model containing the system description, BMC assigns the task of proving compliance or violation to an SAT or SMT solver [14]. The output result, when generated from a failure, gives more guidance on how to fix the bug: not only is the property violated described, but there is a concrete sequence of events that leads up to the violating execution – essential information in helping the developer to track down and correct the problem. [20]

2.1.3 Known Examples

Over the years, BMC has substantially improved the hardware verification analysis [15], which is used to ensure that the given circuit diagrams operate exactly as they should according to the instructions of the microprocessor. For instance, BMC can carry out automatic and step-wise instructions for Verification Microprocessors up to a certain depth, allowing it to identify incorrect transitions that do not lead to any readable outcomes.

The usage of BMC is also used in software systems verification [10] for various programs, specifically in systems where reliability and quality are critical. It lets model software behaviour and evaluates it, resulting in the detection of bugs and problems such as deadlocks and unintended transitions in concurrent or safety apps. For instance, CBMC (C Bounded Model Checker) is a showcase of using this approach as it helps developers verify the correctness of C programs [21] within bounded limits, making the C more trustworthy and resilient.

When it comes to communication and security protocol, BMC plays the role of system-level gatekeeper that not only validates protocol adherence but also checks for security properties during a process [16]. With such thorough analysis of the parameters of the protocol within a certain amount of time, BMC is able to detect faults using information which may lead to security issues and even miscarriage of protocol.

2.2 Java Bounded Model Checker (JBMC)

One of the examples of BMC, and the subject around this project, is the Java Bounded Model Checker (JBMC), which verifies Java programs within a given bound [6].

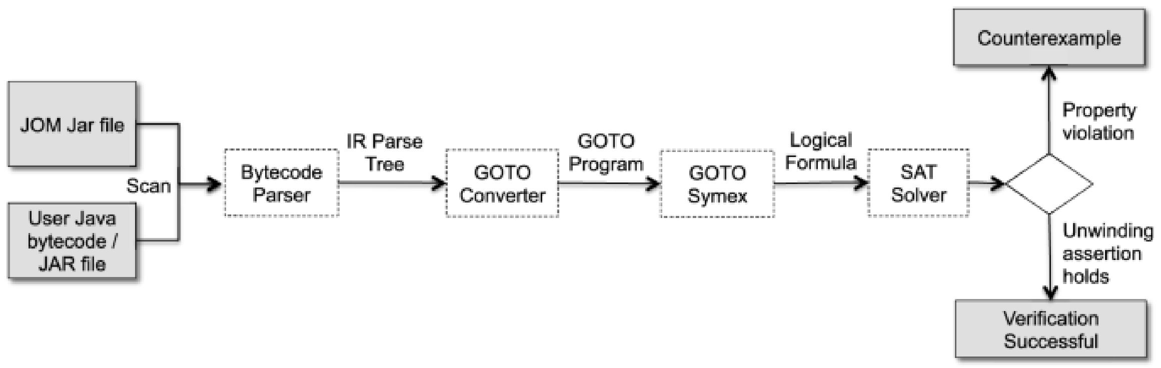


Fig. 2. JBMC Flow - Source [7]

2.2.1 Overview

The JBMC is a remarkable breakthrough in the verification of Java byte-code with a verification model based on SAT and SMT solving [4]–[7], which are well-established frameworks. Instead of going through the abstraction layers, JBMC operates on Java byte-code and, by doing so, verifies the task with high precision using its efficient approach. CPROVER stands for the basis of its framework, which in turn allows it to perform the necessary analysis that handles the complexities of Java's operational semantics [4], [5].

It is based on the CBMC engine used by the analogue C Bounded Model Checker [21] and relies on MiniSAT 2.2.1 to resolve verification conditions. JBMC has a novel methodology containing a verification-friendly Java Operational Model (JOM) [4], [22]. Because of this, JBMC can handle and verify more complex features of Java, for example, class inheritance, polymorphism, array and bit-level operations, as well as floating-point arithmetic, thus providing a solid basis for asserting the trustworthiness of Java programs within the specified limits [5].

2.2.2 JBMC Versatility

The JOM model is just a general abstraction that is based on the Java standard libraries, amending and simplifying standard Java classes to avoid the verification details and, hence, reduce the verification process. Furthermore, [4], [22] JBMC has a string solver as a specialised component, which is a fundamental one that checks the satisfiability of string conditions which involve string operations. The solver is programmed to handle a wide variety of string operations, from concatenation to different types of conversion actions, thus increasing JBMC's analytical powers. JBMC provides a robust API for creating verification harnesses as well as stub functions; therefore, it can be exploited in different verification domains. This emphasises the versatility of JBMC, and it will be constantly evolving to add regular expressions, multi-threading and a broader verification scope [4], [6], [7], [22].

2.2.3 Output

JBMC's performance and testability levels have been one of the key reasons for it being viewed as a powerful verification tool [4], [7]. It provides correct results on many Java verification cases, which proves its validity across different benchmarks. Through this, though, its performance is inherently limited by the boundaries of the analysis set and the concreteness of the Java Operational Model [4], [7]. A restriction of the tool has been recognised as it does not support the advanced Java features and concurrency, which indicates the need for further improvement [7]. Although limited in terms of the boundary context investigation and the possibility of accepting only those Java features which are not too advanced, such as Java 8 lambdas and JNI, the JBMC continues to confirm its validity in the sphere of Java program verification.

JBMC output is highly detailed and is meant to provide accurate information that the developer should use for the decision-making process. JBMC also gives a complete understanding of the problem by constructing counterexamples that reveal the issue step-by-step in detail, that is, between the states of execution that led to the failure. The details of this information not only help to identify the exact point of failure but also give developers a flow of the execution path that can allow debugging to proceed more naturally.

2.2.4 Limitations

JBMC, though a strong helper for Java byte-code verification, also has a set of limitations that were brought out during its participation in both SV-COMP'21, SV-COMP'22 and SV-COMP'23 competitions [8], [9], [22], [23]. The main restriction is that the JBMC is the bounding model checker that is most powerful at bounded programs. For programs which need to make many unwinds, the BMC engine produces large formulas, which can be very difficult for the SAT/SMT solver to process efficiently.

The JBMC architecture has a complex Java operational model that is still in the process of being completely built. It is also true that many standard Java classes, especially those with complex prerequisites, post-requisites, and behaviour patterns, still need to be fully supported [4], [7]. Since this kind of coverage is incomplete, it will result in an over-approximation of the program's original semantics, and the correct verification details could be missed. For multi-threaded Java programs, the efficiency drops drastically for JBMC [4]–[7]. The tool uses partial orders to represent concurrent processes when modelling the execution of programs. However, it does so at the expense of losing part of the automatic analysis capability, resulting in less efficient tools and harder to use.

Furthermore, JBMC's string solver does not support regular expressions, which is a crucial limitation because of the abundance of regex operations in Java programming. On the other hand, the reflection method and the missing native method support further limit the applicability of JBMC, especially when it comes to more complex Java applications that leverage these features [5], [7].

2.3 Counterexamples for Debugging

In software verification, counterexamples are pivotal, particularly within the debugging processes for Java applications [24], [25]. They provide a concrete instance where the application deviates from its expected behaviour, thereby paving the way to identifying and rectifying errors. These counterexamples are essential in the context of formal verification methodologies, enabling developers to visualise the exact sequence of events leading to failure [26].

In Java debugging, counterexamples serve as a bridge between the abstract verification results and concrete actionable insights. They detail the conditions under which specific bugs occur [25], guiding developers through the complex landscape of application logic to pinpoint inaccuracies. This insight is invaluable in environments where Java's reliability is crucial, such as in financial services, healthcare, and other critical systems [26].

Utilising counterexamples effectively accelerates the debugging process. They allow developers to reproduce errors systematically, understand the underlying issues, and verify the impact of their fixes. For Java programmers, this means a significant reduction in time-to-resolution for bugs and an increase in overall code quality [24], [25].

Moreover, integrating the concept of counterexamples into regular development practices encourages a proactive approach to bug resolution and software maintenance. It fosters a deeper comprehension of how application state and behaviour interlink, leading to more robust and fault-tolerant Java applications [24]–[26].

2.4 CPROVER Libraries in Java Verification

CPROVER libraries provide a comprehensive suite of tools for software verification, extending their utility to Java applications through an interface that allows for rigorous analysis and verification processes. These libraries underpin tools like JBMC [4], [5], facilitating detailed examination and verification of Java byte-code against specified correctness criteria. The automation capabilities offered by these libraries streamline the verification process, embedding robustness into the software development life-cycle and fostering a culture of proactive error detection and resolution. As essential tools in the software engineer's toolkit, CPROVER libraries play a pivotal role in modern software verification practices, supporting the development of reliable and error-free Java applications [4]–[7].

2.5 XML Overview and Manipulation

Extensible Markup Language (XML) is a robust, text-based format that serves many purposes in data storage and transportation. Its structure is both human-readable and machine-readable, which makes XML an integral tool in a wide array of applications ranging from simple data storage to complex web services [27]. The hierarchical nature of XML documents allows for detailed data modelling, capturing intricate relationships with nested elements, attributes, and embedded content. The manipulation of XML is a critical skill, encompassing reading (parsing), modifying, and writing XML data. Parsing is the initial step, transforming XML into a directly manipulable structure. Two main methods exist for parsing: the Document Object Model (DOM), which creates an in-memory representation of the XML document, and the Simple API for XML (SAX), an event-driven parser more suitable for large documents due to its lower memory footprint.

For XML modification and transformation, XSLT (Extensible Stylesheet Language Transformations) offers a potent mechanism to reshape XML documents into other XML formats or even different data types like HTML [28]. This is complemented by XPath, a query language for XML, allowing precise data extraction and manipulation within an XML document.

XML also plays a vital role in the interchange of information between disparate systems, ensuring data consistency and interpretability across platforms and programming environments. It facilitates the encapsulation of complex data structures [29], making XML an indispensable component in modern computing environments.

2.6 Summary

The JBMC is an adaptation of BMC specifically tailored for verifying Java programs, which is crucial for ensuring the reliability of Java applications across various industries. Leveraging the foundations of SAT and SMT solving, JBMC verifies Java byte-code with notable precision [22], benefiting from a robust framework (e.g. CPROVER [4]) and an operational model optimised for Java's intricacies, such as class inheritance and polymorphism.

JBMC stands out by directly interacting with Java byte-code, thus eliminating the abstraction layers often present in other model checkers, which allows for a more accurate verification process. It is equipped to analyse complex Java features and employs specialised solvers [4]–[7], like the string solver, to manage a broad spectrum of string operations. Despite its advanced capabilities, JBMC is subject to limitations, particularly when addressing programs requiring extensive unwinds or advanced Java features like concurrency, reflecting an area ripe for further enhancement.

In the broader context of the background section, BMCs, including JBMC, form an integral part of the verification landscape, offering automated and insightful analyses to identify system discrepancies within set boundaries. Counterexamples generated during the BMC process are invaluable for debugging, providing a detailed sequence of events leading to failures, which is particularly beneficial in Java's debugging processes where JBMC is applied.

3 Implementation and Design

3.1 Requirements Overview

The primary objective of this project is to develop a comprehensive wrapper designed to interface seamlessly with Java applications. This wrapper is engineered to accept any Java code — whether compiled or uncompiled — process it using the JBMC, and generate intelligible, executable counterexamples. These counterexamples aim to facilitate the reproduction and subsequent examination of any bugs identified during analysis.

Key functionalities of the wrapper include:

- **Java File Processing:** The wrapper should accept any Java file, regardless of its compilation status. This involves the ability to compile uncompiled Java source code automatically. (Implemented as part of Fig. 3 Step 2)
- **JBMC Integration:** It is critical to verify the JBMC installation path. If the wrapper encounters any discrepancies or errors in the specified JBMC location, it should revert to a predefined default path. This ensures the wrapper maintains its operational integrity, irrespective of the user's system configuration.
- **Unwind Parameter Interaction:** Users must have the option to specify an 'unwind' parameter value. If the user fails to provide this value after three prompts, the system should default to a value of 10. This feature is essential for controlling the depth of loop unwinding within JBMC, impacting both the comprehensiveness and runtime of the analysis. (Implemented as part of Fig. 3 Step 1)
- **Execution and Analysis:** The wrapper should execute JBMC on each method found in the Java input, applying the user-defined or default unwind value. It should leverage CPROVER library specifications to conduct a thorough analysis. (Implemented as part of Fig. 3 Step 3 and 4)
- **Result Interpretation and Counterexample Generation:** Upon receiving the JBMC output (in XML format), the wrapper should systematically generate executable Java files. Each file should replicate the specific circumstances under which a bug was identified, enabling direct bug reproduction at the method level within the original Java application. Each counterexample would be produced in the location of the input file. (Implemented as part of Fig. 3 Step 5 and 6)
- **Method-specific Counterexample Production:** For every analysed method of the input file, the wrapper is mandated to produce distinct, detailed counterexamples. This ensures that each potential issue is addressed with a targeted, method-specific counterexample, enhancing the debugging process. (Implemented as part of Fig. 3 Step 6 and 7)
- **Bug Explanation:** Each generated counterexample file should include a concise comment at the beginning detailing the nature of the detected bug based on JBMC's analysis. This feature aims to provide users with immediate context and understanding of each identified issue. (Implemented as part of Fig. 3 Step 7)

- **Sensible Running Time:** The runtime efficiency of the wrapper is paramount. The goal is to ensure that the wrapper's execution time is competitive with direct JBMC usage on equivalent Java code. This efficiency is crucial for user adoption and practical utility in real-world scenarios.

3.2 Application Architecture and Components

The wrapper is designed to automate the verification of Java programs, identifying potential issues and generating counterexamples to aid debugging. The selection of Python and XML for this tool is based on their respective strengths in handling the tasks required by the wrapper.

XML is chosen for its ability to structure complex and hierarchical data effectively. This is particularly important for representing the output of code analysis, which can include nested structures reflecting the program's execution paths, variable states, and potential error points[30]. The format's ubiquity and the availability of numerous parsing tools across different platforms ensure that XML data can be easily accessed and manipulated, providing flexibility and interoperability for the JBMC wrapper[6]. Moreover, the JBMC supports producing the output in an XML format, which aids in the process of manipulating the output [31].

Python offers robust XML parsing capabilities [32], which are essential for processing the detailed output JBMC generates. Its libraries, such as xml.etree.ElementTree, simplifies the task of navigating and extracting data from XML files. Furthermore, Python's syntax is clear and concise, facilitating the maintenance and enhancement of the tool over time [33].

3.2.1 Code Flow

In the code flow of the wrapper, each operation is methodically structured to ensure an exhaustive analysis and effective generation of counterexamples. Follow the description below and Fig. 3 for the code flow.

The process begins with converting the Java source code into byte-code (Step 1 of Fig. 3), an essential step as JBMC conducts its verification at the byte-code level. This transformation is foundational, preparing the code for in-depth analysis by JBMC. Following this, the wrapper interacts with the user to obtain the unwind parameter, a critical setting that determines the depth of the analysis (Step 2 of Fig. 3). This engagement ensures that the analysis depth is aligned with user expectations and the specific context of the application being examined.

The subsequent phase involves isolating individual methods (Step 3 of Fig. 3) from the byte-code. This methodical extraction enables a focused analysis, ensuring each method is scrutinised independently. JBMC processes each method utilising the unwind parameter and the CPROVER libraries, which facilitates a granular examination of the code, uncovering potential issues within distinct execution paths (Step 4 of Fig. 3).

After JBMC concludes its analysis, the wrapper gets the parsed XML output. This step is pivotal as the XML contains detailed information on the execution paths, variable states, and any discrepancies or anomalies detected during analysis, in a structured and clear manner—the parsing extracts and processes this information, preparing it for the final stage. The information from the XML tree is examined so that the variables, their types, values, and structures can be extracted, which would help in the generation of the counterexample (Step 5 of Fig. 3).

In the wrapper's code flow, after extracting input variables from the JBMC XML trace, an essential validation process ensues to ensure the integrity and correctness of the data. This validation, encapsulated in the ”Input Types Check” step (Step 6 of Fig. 3), involves scrutinising each input variable's type to confirm its adherence to expected Java types. Leveraging a series of helper functions, the wrapper meticulously examines whether the extracted input types align with predefined primitive types, array types, string types, or class types. By employing this rigorous type-checking mechanism, potential inconsistencies or mismatches in input variable types are swiftly identified, enabling preemptive resolution before proceeding further in the analysis pipeline. This meticulous validation process not only enhances the robustness of subsequent analyses but also fosters greater confidence in the reliability of the generated counterexamples.

The culmination of the code flow is the generation of counterexamples (Step 7 of Fig. 3). This stage leverages the information extracted from the XML data to construct Java code snippets that emulate the conditions under which specific issues were identified. These snippets are crafted to precisely mirror the problematic scenarios, providing developers with a direct means to observe, understand, and rectify the detected issues.

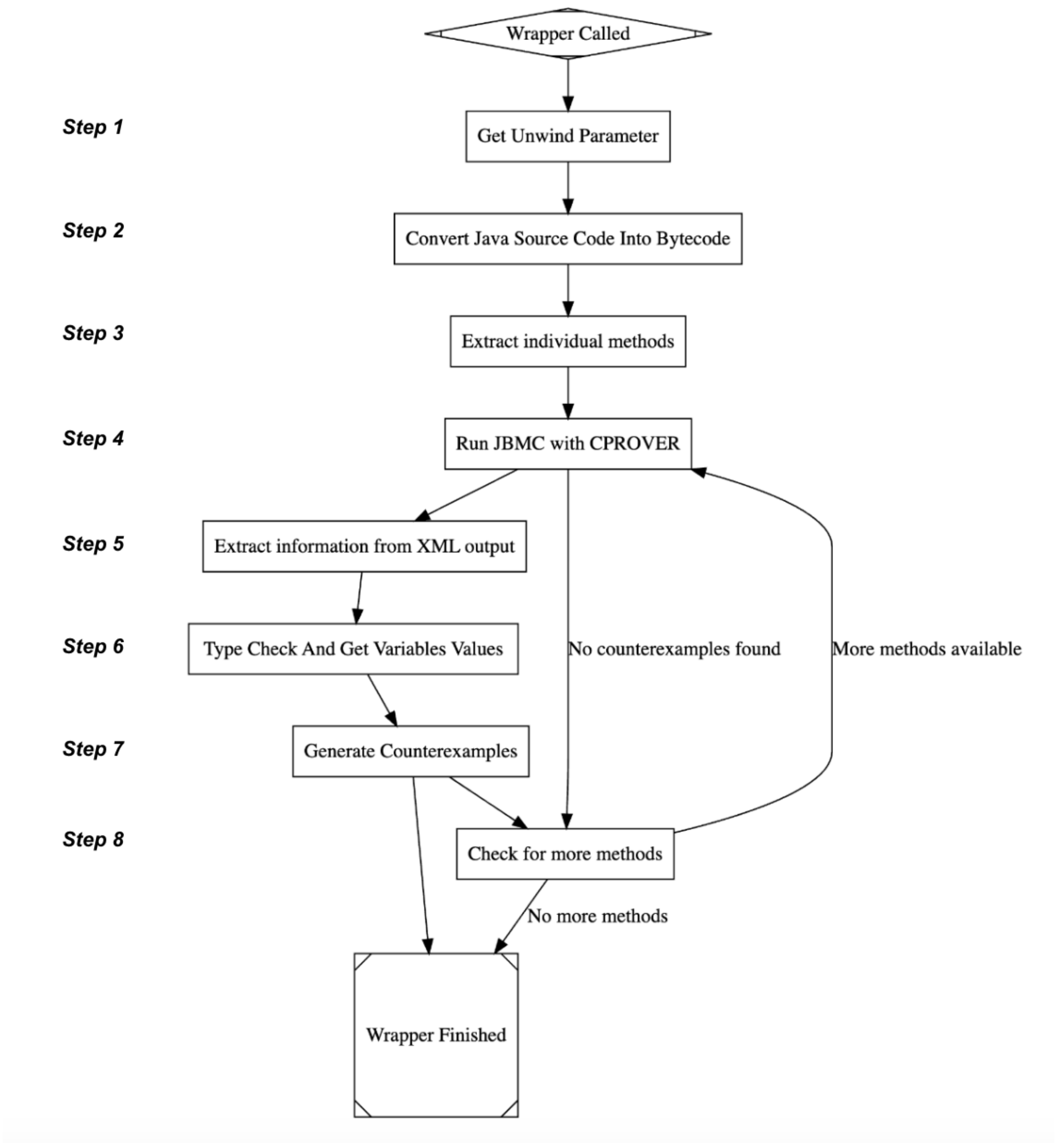


Fig. 3. Wrapper Flow

3.2.2 Running the Wrapper

To execute the wrapper, ensure that your input Java code is located within the `code_verification/<YourAppName>` directory in the wrapper's repository. Then, run the script using the following command:

```
python3 ../../src/jbmc-counterexample.py <jbmc_full_path> <YourJavaFile.java>
```

Replace `<jbmc_full_path>` with the path to your JBMC installation and `<YourJavaFile.java>` with the name of your Java file. The unwind option allows for either manual input upon unwind invocation or a default value of 10 if skipped. Additionally, the program verifies the JBMC location, prompting the user to correct it if necessary. Upon execution, counterexample Java source files are generated in the `code_verification/<YourAppName>` directory, named as `CounterExample<N>.java`.

3.2.3 XML Stack Trace Handing

In the XML Stack Trace Handling mechanism, the process initiates with the extraction of execution traces from JBMC, employing the `--xml-ui` option. This option instructs JBMC to render its analysis results in XML format. Execution of this step is orchestrated through `get_trace_xml` functions within `jbmc_runner.py` and `java_helpers.py`. These functions facilitate the construction and execution of command-line operations to initiate JBMC against specified Java classes, ensuring precise formatting of output for subsequent analysis phases. The pseudocode below illustrates the simplified logic employed to execute the JBMC tool and capture its XML output.

```
# Pseudocode for get_trace_xml function
def get_trace_xml(jbmc_path, class_name, method_name):
    cmd = [jbmc_path, f'{class_name}.{method_name}', '--xml-ui']
    return run(cmd, capture_output=True, text=True)
```

Once the XML trace is retrieved, attention transitions to parsing activities within `input_parser.py` through the `get_inputs` function. This function rigorously parses the XML to distil essential information concerning program execution paths, variable states, and conditions under which failures were observed, segregating data according to variable types and contextual deployment within the execution sequence (Step 6 of Fig. 3).

Identification of variable types is pivotal in parsing, supported by `input_type_checker.py` through functions like `is_array_type`, `is_class_type`, `is_primitive_type`, and `is_string_type`. These functions systematically categorise variables, which is foundational for reconstructing program states accurately during failure instances. The detailed parsing and categorisation process is facilitated by methods such as `get_input_type` and `get_input_value`, ensuring precise understanding and representation of variable states within the execution trace. Fig. 4 represents the flow described above, and the order in which the XML trace is processed if a bug flag was raised by JBMC.

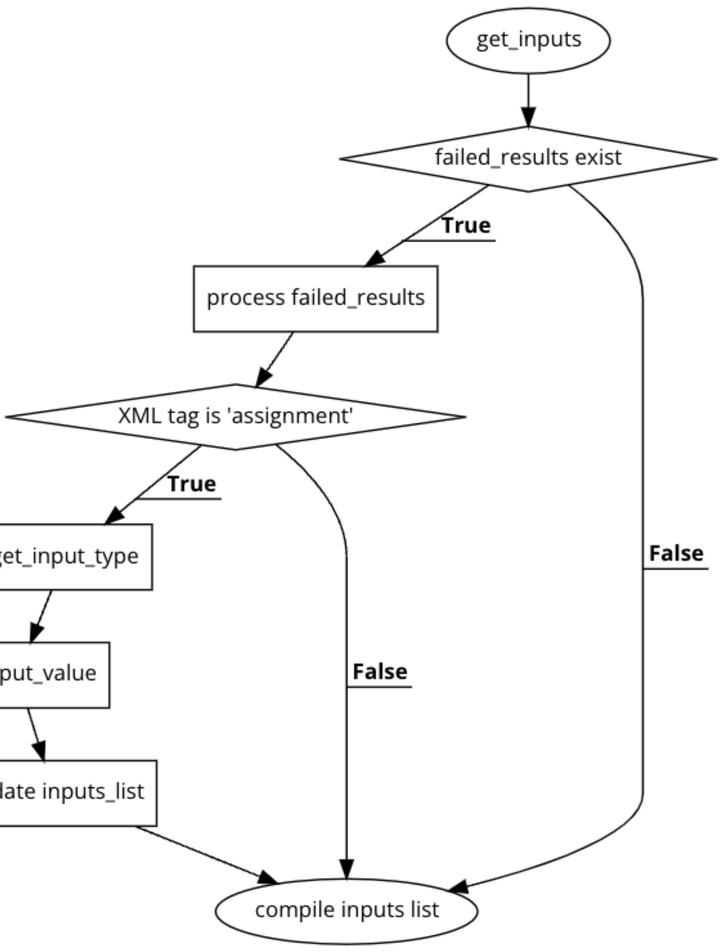


Fig. 4. XML Trace Processing Based on Type

This parsing mechanism extends to decode complex data structures, meticulously resolving nested arrays and class instances to accurately portray their states as recorded in the trace. To further illustrate this process, an expanded view into the parsing mechanism is provided through additional pseudocode below, showcasing the advanced parsing capabilities designed to handle complex data types within the XML trace.

```
# Pseudocode for identifying and parsing data types in input_parser.py
function parse_complex_data(trace_element):
    if is_array_type(trace_element.type):
        return parse_array(trace_element)
    elif is_class_type(trace_element.type):
        return parse_object(trace_element)
    else:
        return parse_primitive(trace_element)

function parse_array(array_element):
    initialise array_data
    for each element in array_element:
        append parse_complex_data(element) to array_data
    return array_data

function parse_object(object_element):
    initialise object_data
    for each field in object_element:
        object_data[field.name] = parse_complex_data(field)
    return object_data

function parse_primitive(primitive_element):
    return primitive_element.value
```

In addition to the pseudocode above, and to Fig. 4, Fig. 5 represents the logic on how every input type is determined, during the flow of processing the bug from the XML output produced by the JBMC. This is an essential step to ensure that the assignment is made to the correct type, so that the bug can be adequately replicated.

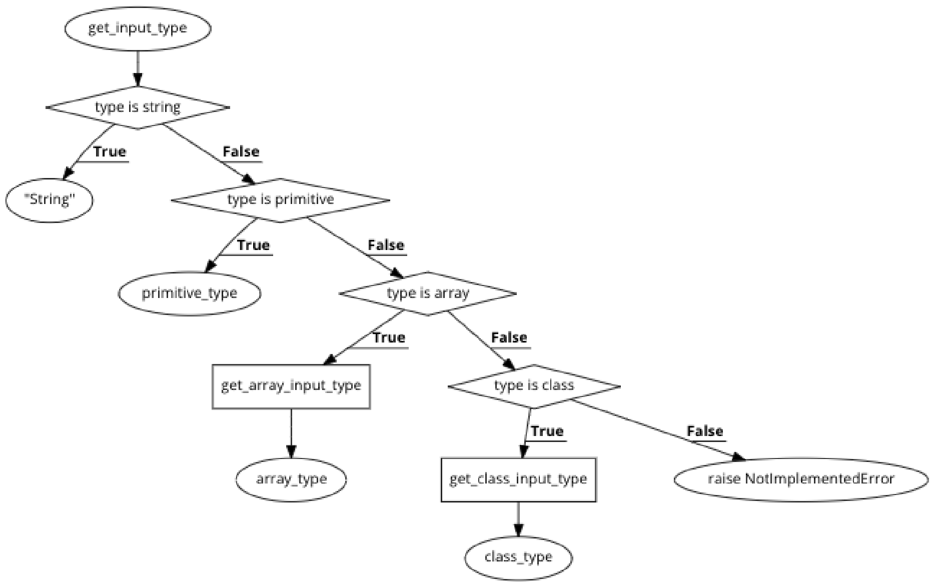


Fig. 5. Get Input Type Logic

3.2.4 Counterexample Generation

After the XML trace and go-to trace from JBMC had been extracted, the input type was checked and the value parsed; a dictionary was built containing all that information, the method name where the bug was identified, and the reason why the bug was detected. This dictionary is built along the type and value checking explained in the sub-section above. This dictionary is then passed to a series of methods and helpers that would extract the information and then build more intuitive counterexamples (Step 7 of Fig. 3).

A dedicated function facilitates the dynamic creation of Java source code based on the abovementioned parameters. It begins by initialising an empty list to store lines of Java code. Subsequently, a comment is added to add the reason for the counterexample (the bug explanation). The function then constructs the Java class, including its primary method, which serves as the program's entry point. During the iteration through the provided counterexample inputs, details such as variable names, types, and values are extracted (Fig. 4 and Fig. 5). Depending on the input type, an appropriate initialisation code is generated. For primitives, assignment expressions are used, while objects and arrays trigger helper functions to create the necessary code blocks. After input processing, an argument list for invoking the associated test method is constructed. Finally, the function completes the Java source code by closing the primary method and the class, joining all lines of code into a single string for output. This process streamlines the generation of Java code tailored to specific scenarios, aiding in testing and debugging efforts. The below pseudocode exemplifies the actual code flow explained and is an excellent simplified illustration of the actual implementation.

```
# Pseudocode for generating Java source code
function generate_java_source(test_class_name, out_class_name, counterexample_inputs,
    reason, method_name):
    source_code = []  # Initialise empty list to store lines of Java code
    source_code.append("// Counterexample for: " + reason)  # Add comment indicating
        reason for counterexample
    source_code.append("class " + out_class_name + " {")  # Begin generating Java class
    source_code.append("\tpublic static void main(String[] args) {")  # Begin main method
    var_name_base = "classVar"  # Initialize base variable name
    var_index = 0  # Initialize variable index
    for var_name, var_info in counterexample_inputs.items():  # Iterate through
        counterexample inputs
        var_type = var_info['type']  # Extract variable type
        var_value = var_info['value']  # Extract variable value
        if isinstance(var_value, dict):  # Check if variable is an object
            generate_object_initialization(var_name, var_value, indent=2)
            # Generate object initialisation code
        elif isinstance(var_value, list):  # Check if variable is an array
            generate_array_initialization(var_name, var_value, indent=2)
            # Generate array initialisation code
        else:  # Variable is a primitive
            assignment_expr = var_type + " " + var_name + " = " + var_value
            # Construct assignment expression
            source_code.append("\t\t" + assignment_expr + ";")
            # Add assignment expression to source code
    arg_list = ", ".join(counterexample_inputs.keys())
    # Generate argument list for test method call
    source_code.append("\t\t" + test_class_name + "." + method_name
        + "(" + arg_list + ");")  # Add test method call
    source_code.append("\t}")  # Close main method
    source_code.append("}")  # Close class
    return "\n".join(source_code)  # Join lines of code and return as string
```

The above happens for every method identified in the program that the JBMC has flagged as having bugs. The wrapper generates a separate Counterexample file for every bug identified in the Input file directory. The Counterexamples are enumerated from one to N, where N is the number of total bugs determined by the JBMC check.

If the JBMC does not raise any bugs, the trace is not processed further, and no counterexample is produced.

3.2.5 User-friendly Code-base and Output

The wrapper is equipped with structured output, notably the progress indicators during the JBMC execution and clear messaging on the generation of counterexamples or success notifications. These provide users with real-time feedback on the tool's processes, providing clear notes in case of success or possible issues while running the wrapper. This transparency helps users measure progress and understand the outcomes of their actions with the tool.

Interactive components, such as user prompts to enter the JBMC path or unwind limit, exemplify the code's adaptability to user input, making it more responsive and adaptable. These prompts ensure that users are engaged in the execution process, providing opportunities to correct errors or adjust parameters dynamically.

The code-base is thoroughly written, incorporating clear and concise documentation, with comments and doc-strings explaining the purpose and parameters of each function, as well as explaining the whole of every method. This would help, especially if a different developer wants to contribute further to the wrapper and needs to understand the code-base and its flow. Moreover, the repository was equipped with a README.md file that explains the prerequisites required to run the wrapper, as well as the file organisation and the commands needed.

3.3 Limitations

As the wrapper is developed, it is built on top of the JBMC, and its implementation will naturally inherit all the limitations of the JBMC outlined in the Background Chapter. Moreover, there are some implementation-specific limitations due to its implementation and XML handling. All these limitations were examined and discovered during the testing phase, which is described in a detailed manner in Chapter Four.

3.3.1 Inherited Limitations From JBMC

One notable limitation of JBMC is its scalability [4], [5], [7]. Therefore, the wrapper is affected by the complexity of the analysed program, as the number of execution paths that need to be examined would proportionally grow. For larger-scale codes, the developed wrapper can suffer from the state explosion problem when running the JBMC, where the resources required to analyse a program grow exponentially with the size of the program. This can make the analysis of large or complex Java applications impractical, as the time and computational resources required can become prohibitive [5], [7], [22].

Another constraint is related to the handling of dynamic features inherent to Java. While JBMC can effectively analyse a wide range of Java applications, it may struggle with dynamic language features such as reflection and dynamic class loading, making the counterexample generation impractical for our wrapper. These features can introduce non-determinism and obscure the program structure, complicating the model-checking process[4], [5].

Additionally, JBMC's effectiveness is dependent on the quality and completeness of the specifications or properties being verified. If the specifications are incomplete or incorrect, JBMC might either fail to detect actual bugs or report issues that are not genuine [4], [7], [22], undermining the confidence in the verification process. Therefore, our wrapper depends on the number of unwinds specified by the user, as well as the CPROVER library checks.

In the context of enhancing JBMC with a wrapper, it is crucial to address specific limitations inherent to JBMC that could affect its integration and effectiveness. Concurrency presents a notable challenge, as JBMC's capacity to model and analyse complex interactions in multi-threaded applications can be limited, potentially missing critical issues such as race conditions or deadlocks. [4]–[7], [22] The tool's reliance on precise modelling of the Java standard library is another limitation, where inaccuracies can undermine verification results, necessitating continuous updates to align with the evolving Java ecosystem. Furthermore, the effective utilisation of JBMC, especially when extended with a wrapper, demands substantial user expertise in formal methods and the tool's specifics.

Therefore, the wrapper strictly depends on the bugs the JBMC determines. If the JBMC does not flag a given bug, it cannot generate an XML trace for it, and therefore, the wrapper cannot build a counterexample for it.

Therefore, as the wrapper relies on running the JBMC, the generation of more understandable bugs would grow with the development of the JBMC - new types of bugs detected by the JBMC would be incorporated in the wrapper, as long as the wrapper would be called with the latest version of the JBMC.

3.3.2 Implementation-Specific Limitations

One of the main limitations of the developed wrapper is that it runs based on the CPROVER library but does not allow for further libraries or JBMC specifications to be defined by the user so that JBMC can be applied to those.

Moreover, the wrapper builds the counterexamples based on the XML output provided by the JBMC check. Parsing XML can be computationally intensive [34], especially for large files. The complexity of parsing processes can impact the performance of large-scale systems, increasing the time required for data processing. Moreover, XML is known for its verbosity [34], [35], which can lead to significantly larger file sizes compared to other formats. This can be a critical issue for large systems where data transmission and storage efficiency are crucial. This is why, in several cases, depending on the scale of the bug and its complexity, the wrapper can struggle with time complexity and generate correct counterexamples, especially in cases such as very nested classes/methods or complex dynamic structures.

Moreover, due to the nature of the implementation, the wrapper is effective at producing illustrative counterexamples for Java methods that have entry point attributes, and the correctness of those entry points depends on the correctness of the method. In other words, if the JBMC flags any potential errors on a Java method that does not rely on any method's attributes, it will generate a counterexample, which will invoke the method. Therefore, this would not necessarily facilitate the debugging process, as the wrapper would converge just to a JBMC check, not adding any extra information on top of the check.

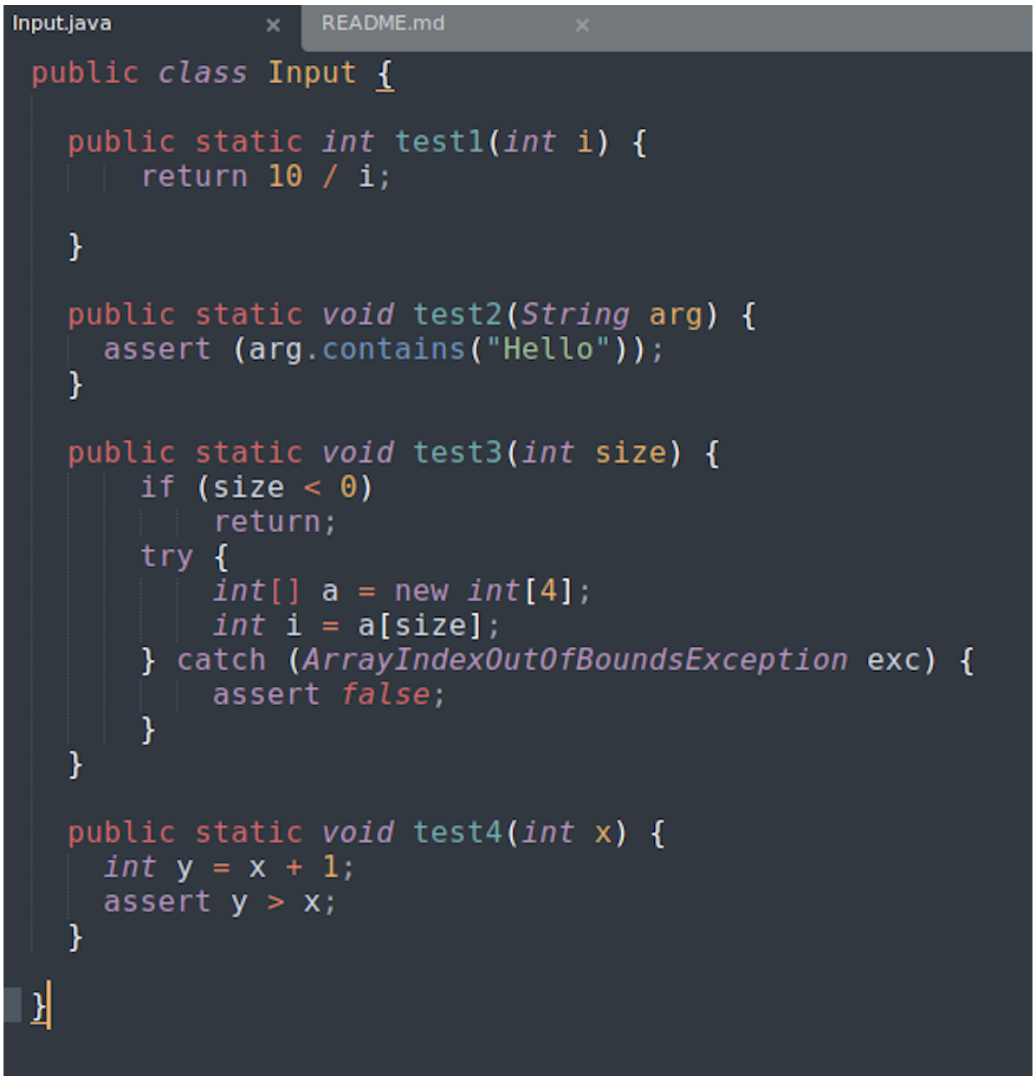
Furthermore, the name of the variables with which the method was called was not preserved and noted as "attrX", where X is a number from 0 to N, N being the maximum number of attributes for a given method analysed with flagged problems by the JBMC. As this is not a direct limitation, it can make the counterexample easier to understand and less intuitive, and the base code needs to be compared with the counterexamples to understand each of the method's attributes.

Proper compliance with the instructions in the README.md file is essential when configuring the file structure and invoking the wrapper, mainly because the CPROVER libraries are integral to the project's framework. The application must be well-informed of the precise location of these libraries within the repository's structure to ensure effective integration and operation, representing a limitation for where the Input file and project can be placed and called from.

The potential to overcome these limitations will be discussed in the Sixth Chapter for future implementations of the wrapper.

3.4 Illustrative Example

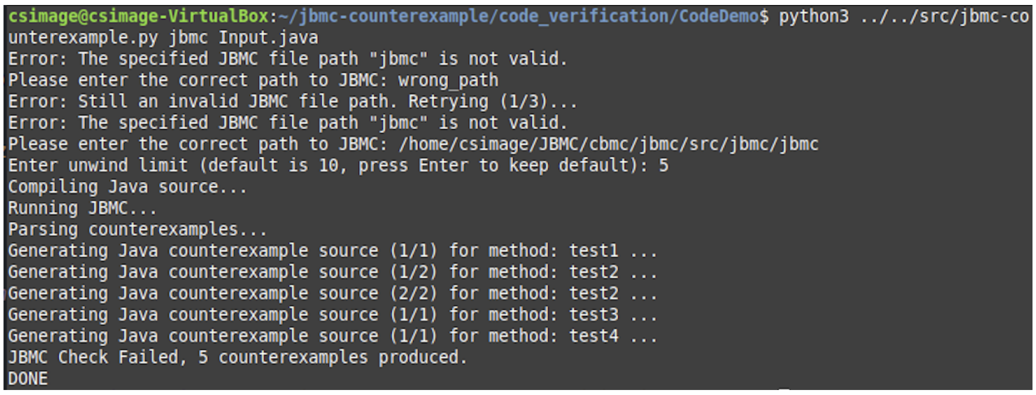
This section aims to illustrate the application's functionality through a practical demonstration. For this purpose, a concise input file is utilised, encompassing four distinct methods that engage with arrays, signed number overflow, strings, and integer arithmetic. This diversity in examples showcases the broad spectrum of capabilities inherent in the developed wrapper. The referenced input file is delineated in Fig. 6.



```java
public class Input {

  public static int test1(int i) {
      return 10 / i;

  }

  public static void test2(String arg) {
    assert (arg.contains("Hello"));
  }

  public static void test3(int size) {
      if (size < 0)
          return;
      try {
          int[] a = new int[4];
          int i = a[size];
      } catch (ArrayIndexOutOfBoundsException exc) {
          assert false;
      }
  }

  public static void test4(int x) {
    int y = x + 1;
    assert y > x;
  }

}
```

Fig. 6. Java File Chosen as Input

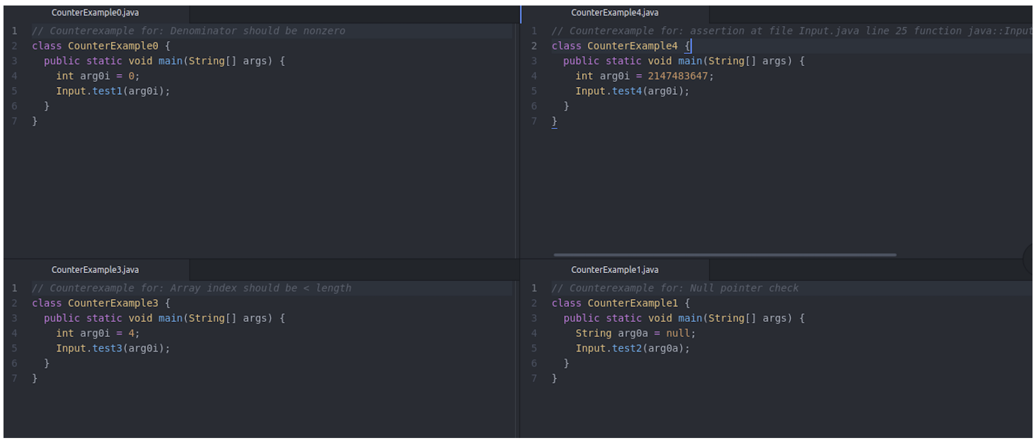
As depicted in Fig. 7, the execution of the wrapper commences from a predefined location, adhering to the file structure delineated in previous sections and detailed within the README.md documentation. The initial operation of the wrapper entails prompting the user to verify or specify the correct JBMC installation path. Should the path not align with the one anticipated by the wrapper invocation, a user-friendly error notification is generated. Upon establishing the accurate JBMC path, the user is then prompted to input an unwind value or to press enter to default to an unwind value of 10, as illustrated in Fig. 7. Following this input, the wrapper compiles the Java source file specified by the user and initiates the JBMC analysis, which is predicated on the provided unwind value and leverages the necessary CPROVER libraries. Subsequent to the analysis, a message is relayed to the user, elucidating any issues identified by JBMC, the quantity of generated counterexamples, and the specific methods they pertain to, as detailed in Fig. 7.



```
csimage@csimage-VirtualBox:~/jbmc-counterexample/code_verification/CodeDemo$ python3 ../../src/jbmc-co
unterexample.py jbmc Input.java
Error: The specified JBMC file path "jbmc" is not valid.
Please enter the correct path to JBMC: wrong_path
Error: Still an invalid JBMC file path. Retrying (1/3)...
Error: The specified JBMC file path "jbmc" is not valid.
Please enter the correct path to JBMC: /home/csimage/JBMC/cbmc/jbmc/src/jbmc/jbmc
Enter unwind limit (default is 10, press Enter to keep default): 5
Compiling Java source...
Running JBMC...
Parsing counterexamples...
Generating Java counterexample source (1/1) for method: test1 ...
Generating Java counterexample source (1/2) for method: test2 ...
Generating Java counterexample source (2/2) for method: test2 ...
Generating Java counterexample source (1/1) for method: test3 ...
Generating Java counterexample source (1/1) for method: test4 ...
JBMC Check Failed, 5 counterexamples produced.
DONE
```

Fig. 7. Calling the Wrapper - The Flow

Figure Fig. 8 shows the resultant counterexamples produced by the wrapper based on the preliminary input. Each counterexample file is prefaced with a comment that encapsulates the nature of the detected issue. Furthermore, these counterexamples are executable, enabling the replication of the discovered issues by invoking the original input file, thereby facilitating a tangible demonstration of the identified bugs.



CounterExample0.java

```java
// Counterexample for: Denominator should be nonzero
class CounterExample0 {
  public static void main(String[] args) {
    int arg0i = 0;
    Input.test1(arg0i);
  }
}
```

CounterExample4.java

```java
// Counterexample for: assertion at file Input.java line 25 function java::Input
class CounterExample4 {
  public static void main(String[] args) {
    int arg0i = 2147483647;
    Input.test4(arg0i);
  }
}
```

CounterExample3.java

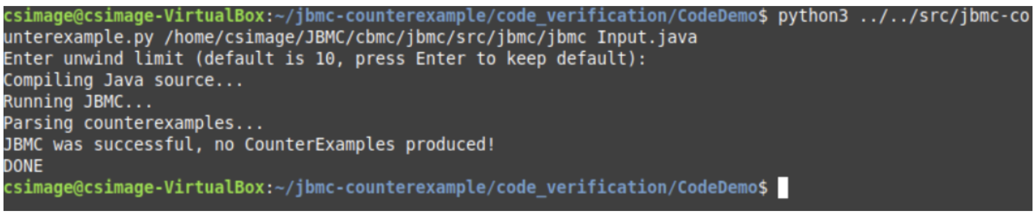
```java
// Counterexample for: Array index should be < length
class CounterExample3 {
  public static void main(String[] args) {
    int arg0i = 4;
    Input.test3(arg0i);
  }
}
```

CounterExample1.java

```java
// Counterexample for: Null pointer check
class CounterExample1 {
  public static void main(String[] args) {
    String arg0a = null;
    Input.test2(arg0a);
  }
}
```

Fig. 8. Calling the Wrapper - The Flow

Fig. 9 illustrates the wrapper's response when invoked with an input file that JBMC, following the given specifications, identifies as having no issues, thereby not generating any counterexamples. In such instances, the wrapper proactively communicates a clear message confirming the absence of detectable problems. This feedback mechanism is integral to the wrapper's design, ensuring that users receive an immediate and unambiguous confirmation when their analysed code exhibits no issues as per the JBMC's verification process.



```
csimage@csimage-VirtualBox:~/jbmc-counterexample/code_verification/CodeDemo$ python3 ../../src/jbmc-co
unterexample.py /home/csimage/JBMC/cbmc/jbmc/src/jbmc/jbmc Input.java
Enter unwind limit (default is 10, press Enter to keep default):
Compiling Java source...
Running JBMC...
Parsing counterexamples...
JBMC was successful, no CounterExamples produced!
DONE
csimage@csimage-VirtualBox:~/jbmc-counterexample/code_verification/CodeDemo$ 
```

Fig. 9. Calling the Wrapper - No Counterexamples

4 Testing and Evaluation

This section will explain and present the testing methodologies used to test the performance and functionality of the developed wrapper. These would include a suite of manually created simple cases, some test cases inspired by the benchmarks used by the SV-COMP competitions [8], [9], [23], and testing on a few Java files of open-source projects. This section will look into the functionality, time performance, and readability of the counterexamples produced by the wrapper.

4.1 Overview

4.1.1 Experimental Setup

The experimental setup is meticulously designed to simulate a variety of debugging scenarios that Java developers commonly encounter. It encompasses the following specific setups:

- **Simple Test Cases:** A series of basic Java programs are devised to examine the wrapper's precision in detecting and elaborating straightforward bugs. This set includes arithmetic errors, string manipulations, array handling, and algorithmic logic faults. Each case is crafted to reveal the wrapper's responsiveness to distinct bug types, offering a granular understanding of its diagnostic capabilities.
- **SV-COMP Inspired Cases:** Leveraging benchmarks inspired by the SV-COMP competition, this setup intensifies the complexity, testing the wrapper's performance against more intricate and nuanced Java code scenarios. The cases are selected to cover a spectrum of verification challenges, including control structures, data manipulation, and concurrency issues, reflecting real-world application complexities.
- **Open-Source Application Analysis:** Two prominent Java classes, `StringUtils` and `ArrayUtils` from the Apache Commons Lang library, are analysed under two conditions: their original, bug-free state and a modified state with injected errors. This analysis aims to assess the wrapper's effectiveness in a realistic development environment, evaluating its ability to detect and illustrate bugs within widely used Java libraries.

4.1.2 Objectives

The core objectives of this evaluation chapter are meticulously defined to ascertain the comprehensive capabilities of the developed wrapper:

- **Accuracy and Precision:** To determine the wrapper's effectiveness in accurately identifying bugs and generating corresponding counterexamples that are executable and illustrative of the issue at hand.

- **Efficiency and Performance:** Evaluate the wrapper's time performance, especially in comparison to direct JBMC executions, ensuring that the added utility does not come at a high computational cost.
- **Scalability and Robustness:** To challenge the wrapper with a broad array of test cases, gauging its ability to maintain consistent performance and reliability across straightforward to complex Java applications.
- **User Experience and Practicality:** To critically analyse the counterexamples' clarity and usefulness from a developer's perspective, assessing whether they provide actionable insights that facilitate an expedited debugging process.

4.2 Simple Cases

The first step in the evaluation of the method was to create simple, straightforward cases based on the capabilities of JBMC run with CPROVER [4]–[6] so that a clear overview of the quality of the counterexamples can be observed, as well as the running time and possible limitations of the developed wrapper.

Based on the JBMC abilities, these simple cases were written to test, but were not limited to, the following functionalities and potential problems:

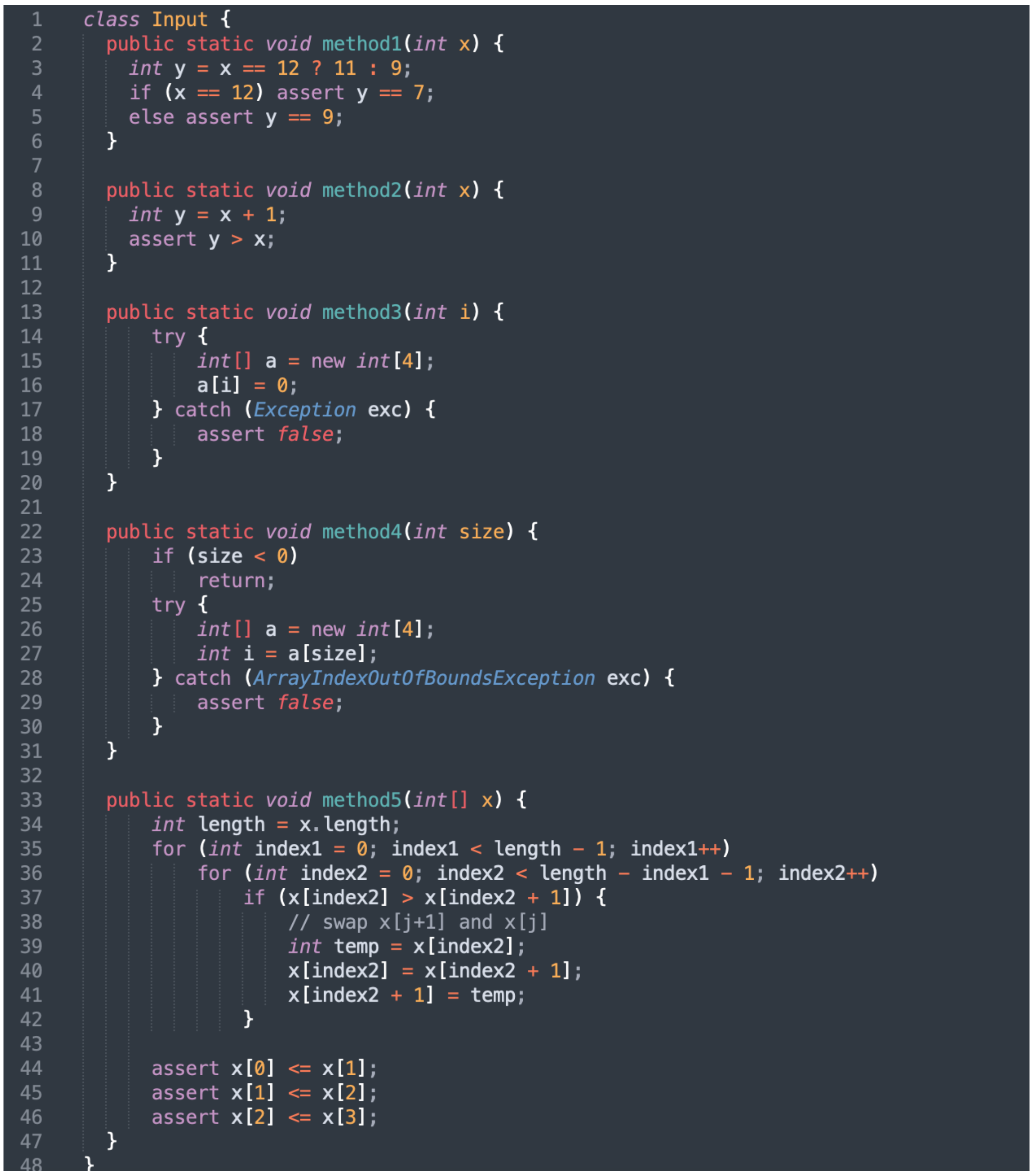
- **Arithmetic operations:**
  - division with null
  - division with zero
  - number overflowing
  - modulo by zero
  - integer underflow
  - loss of precision in integer division
  - floating-point precision issues
  - incorrect bit-wise operations
  - overflow in exponentiation
  - unanticipated Math function behaviour
  - division by near-zero values
  - erroneous algebraic simplifications
  - improper shift operations

- **Strings:**
  - contains ”asserts” not matching
  - string index out of bounds
  - comparing strings using ’==’ instead of ’equals()’
  - null string operations leading to NullPointerException
  - improper use of string format specifiers
  - concatenating strings in loops without StringBuilder
  - encoding issues when converting strings to bytes
  - regular expression pattern errors causing PatternSyntaxException
  - incorrect substring operations leading to unexpected results
  - misuse of string methods leading to case-insensitive comparison issues
  - string internment misuse affecting memory consumption
  - excessive string length causing memory issues
- **Arrays:**
  - array index out of bounds
  - initialising an array with null size
  - accessing array with negative index
  - modifying array size at runtime
  - comparing arrays using ’==’ instead of ’Arrays.equals()’
  - cloning arrays shallowly when a deep copy is needed
  - using incorrect length in array copy operations
  - neglecting potential ’null’ values within arrays
  - concurrent modification of an array during iteration
- **Known Algorithms (with some unchecked exceptions introduced):**
  - Bubble Sort
  - Insertion Sort
  - Quick Sort
  - Binary Search Algorithm

  - Breadth First Search (BFS) Algorithm
  - Depth First Search (DFS) Algorithm
  - Merge Sort Algorithm
  - Quicksort Algorithm
  - Kruskal's Algorithm
  - Floyd Warshall Algorithm
  - Dijkstra's Algorithm
  - Bellman Ford Algorithm
  - Insertion Sort Algorithm
  - Selection Sort Algorithm
  - Counting Sort Algorithm
  - Heap Sort Algorithm
  - Linear Search
  - Euclid's Algorithm
- **Assertions:**
  - asserting a given value is of a set value
- **If-else statements with potential issues on at least one logical branch**
- **Nested Classes**

4.2.1 Methodology

Initially, distinct Java files were devised, each encapsulating a single method to represent a specific case, as mentioned earlier, thereby ensuring clarity and simplicity in the demonstration. Subsequent to this preliminary phase, the scenarios were combined, escalating the complexity of the test files through an augmentation in the number of methods per class. This progression aimed to facilitate a rigorous evaluation of the wrapper's efficacy in processing more intricate input files and structures. An illustrative example of such an integrated case, featuring multiple methods within a single file, is depicted in Fig. 10.



```java
class Input {
  public static void method1(int x) {
    int y = x == 12 ? 11 : 9;
    if (x == 12) assert y == 7;
    else assert y == 9;
  }

  public static void method2(int x) {
    int y = x + 1;
    assert y > x;
  }

  public static void method3(int i) {
      try {
          int[] a = new int[4];
          a[i] = 0;
      } catch (Exception exc) {
          assert false;
      }
  }

  public static void method4(int size) {
      if (size < 0)
          return;
      try {
          int[] a = new int[4];
          int i = a[size];
      } catch (ArrayIndexOutOfBoundsException exc) {
          assert false;
      }
  }

  public static void method5(int[] x) {
      int length = x.length;
      for (int index1 = 0; index1 < length - 1; index1++)
          for (int index2 = 0; index2 < length - index1 - 1; index2++)
              if (x[index2] > x[index2 + 1]) {
                  // swap x[j+1] and x[j]
                  int temp = x[index2];
                  x[index2] = x[index2 + 1];
                  x[index2 + 1] = temp;
              }

      assert x[0] <= x[1];
      assert x[1] <= x[2];
      assert x[2] <= x[3];
  }
}
```

Fig. 10. Input File Combining Several Testing Functionalities

It should be noted that all methods aim to have at least one attribute entry point that would contribute to the correctness of the program analysed by the JBMC. Predominantly, an unwind value of 10 was employed, aligning with the wrapper's default setting.

In terms of output analysis, attention was focused on scrutinising the generated counterexamples and gauging the wrapper's execution duration. The execution time evaluation encompassed running the test programs via the wrapper alongside a separate execution through JBMC to ascertain a comparative understanding of the temporal demands associated with XML analysis and counterexample generation. This dual-pronged approach yielded insights into the operational efficiency and analytical depth of the wrapper when confronted with a spectrum of test cases.

Furthermore, to test the wrapper's performance, a series of Java programs that were not identified as having bugs was introduced by running the JBMC with an unwind value of 10.

4.2.2 Observations

The wrapper demonstrated commendable efficacy in this analysis, generating counterexamples that aligned with expectations. A pertinent illustration of this success is depicted in Fig. 9, showcasing counterexamples for relatively straightforward scenarios. These counterexamples are characterised by their clarity and intelligibility, incorporating variable assignments for each method's attributes and the corresponding method invocations.

However, the evaluation also uncovered certain limitations within the wrapper. Among 80 assessed cases, five discrepancies were noted, particularly in the context of string functionalities—specifically, the '.contains()' method's behaviour when comparing two distinct strings. Additionally, issues were identified in dynamic object assignments within array checks. Another point of concern is the occasional ambiguity in the counterexamples' explanatory comments, which may lack intuitive clarity regarding the nature of the detected bugs (e.g., // Counterexample for: assertion at file Input.java line 25 function java::Input.test4:(I)V bytecode-index 12). Particularly in string-related tests, there were instances where generated strings in counterexamples extended to approximately 1500 characters, potentially complicating their interpretation.

From a performance perspective, the analysis across 80 test scenarios revealed that the additional computational overhead incurred by the wrapper, in comparison to direct JBMC execution, ranged from 5-10%. Despite this increment, the execution time remained below 6 seconds across all cases, with an average duration approximating 2.5 seconds, as demonstrated in Fig. 11. This metric underscores the wrapper's efficiency, maintaining a reasonable time performance while extending JBMC's functionality.

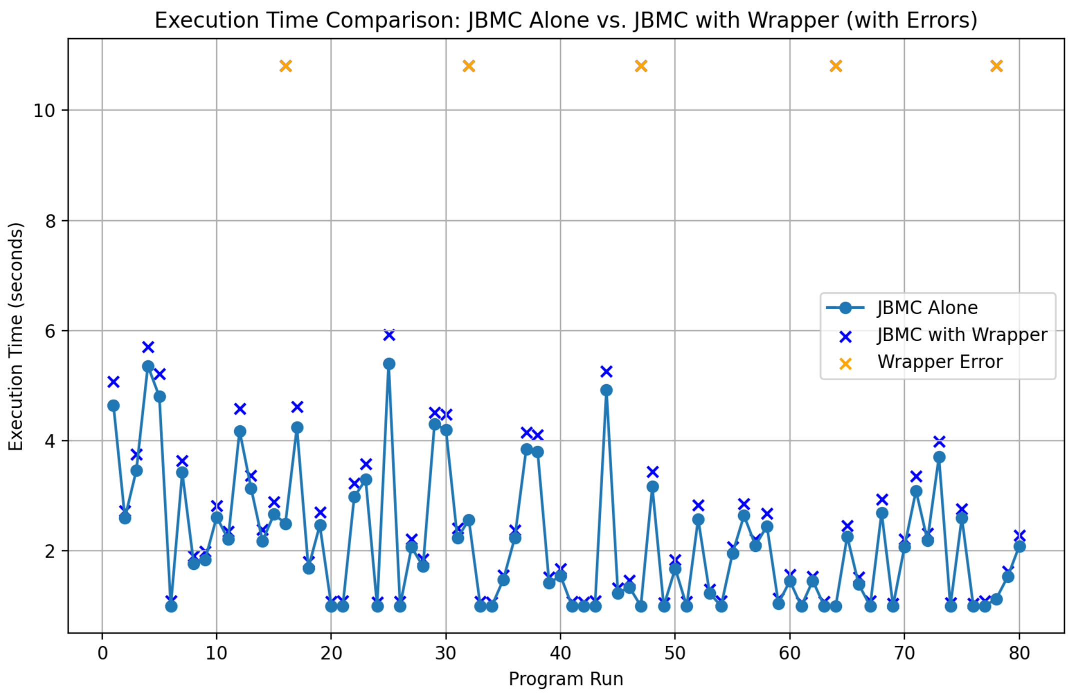


Fig. 11. JBMC vs Wrapper - Execution Performance Over 80 Cases

4.3 SV-COMP Inspired Testing

The methodology adopted involves employing a selected set of benchmark programs inspired by the SV-COMP[8], [9], [23] annual challenge, tailored to assess the capabilities of the developed wrapper. The benchmarks were chosen to reflect a variety of verification challenges akin to those encountered in the SV-COMP, ensuring a broad evaluation spectrum. The direct running of SV-COMP was not possible due to the wrapper's design limitations. The repository from where the inspiration when developing the cases was taken can be found at `https://github.com/sosy-lab/sv-benchmarks/tree/master/java`.

4.3.1 Methodology

The testing process commenced with the application of the wrapper on 30 benchmark programs. The selected 30 benchmark programs were crafted to span a broad spectrum of verification scenarios, ensuring a comprehensive assessment that parallels the diversity encountered in SV-COMP challenges. This selection aimed to test various aspects of the wrapper, from handling basic control structures to more complex algorithmic logic, thereby providing a well-rounded perspective on its efficacy and robustness. Detailed measurements of execution times were meticulously recorded, emphasising the identification of any significant deviations or anomalies. For each benchmark, the wrapper's overhead was quantitatively assessed by comparing the increase in execution time relative to JBMC's standalone performance, establishing a baseline for the evaluation of its computational impact.

4.3.2 Observations

The observations drawn from the Fig. 12, illustrated in the subsequent graph, reveal critical insights into the wrapper's performance. Notably, the average execution time for JBMC alone across these programs was approximately 210 seconds, with the wrapper introducing a 10-15% increase in execution time, aligning with the anticipated overhead. However, it's pertinent to highlight instances of wrapper errors observed in 4 out of the 30 cases, where execution times significantly exceeded the average, likely attributable to specific complexities or inefficiencies in handling those particular cases.

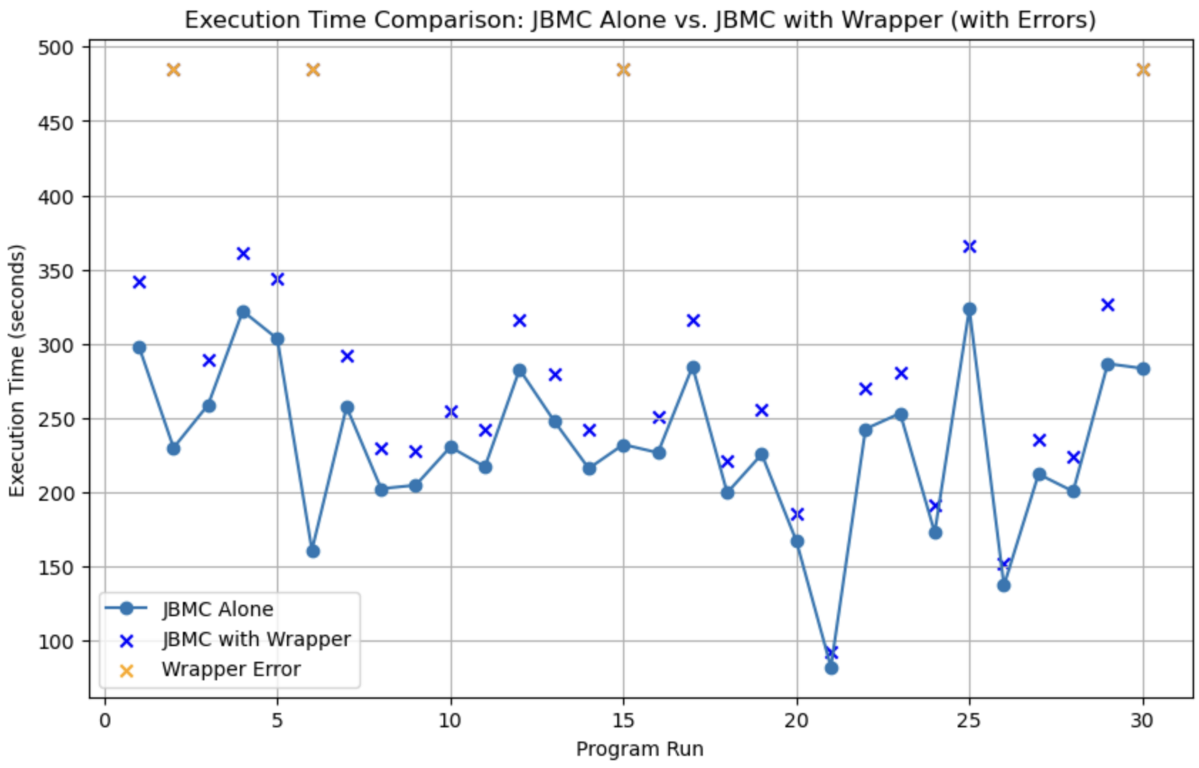


Fig. 12. JBMC vs Wrapper - Execution Performance Over 30 Cases

Reflecting on the broader context, it's essential to consider the performance benchmark set by JBMC in CV-COMP 2022 [9], which averaged at 0.44 hours. Extrapolating from our observations, if the wrapper design were to be integrated and aligned with the CV-COMP benchmarks, a similar 10-15% increase in execution time could be expected. This incremental analysis provides a foundational understanding, suggesting that while the wrapper introduces a modest overhead, it maintains a reasonable efficiency margin, reflecting its practical applicability in enhancing JBMC's verification process.

4.4 Testing on Open-Source Code: StringUtils.java and ArrayUtils.java

In this section, we explore the application of our developed wrapper, operating JBMC, on two well-regarded and widely-utilised Java classes from the Apache Commons Lang library: `StringUtils.java` and `ArrayUtils.java`. This exercise aims to validate the wrapper's effectiveness in detecting and reporting potential bugs under both standard and manipulated conditions in more extensive, well-known code bases. For all the tests, the default 10 for unwind was used.

Initially, our tests focused on `StringUtils.java` and `ArrayUtils.java` without any modifications. These classes are part of the Apache Commons Lang library, which is known for its robustness and reliability, given its extensive use in numerous Java applications. The respective files can be accessed via the following links: `StringUtils.java`: `https://github.com/apache/commons-lang/blob/master/src/main/java/org/apache/commons/lang3/StringUtils.java` and `ArrayUtils.java`: `https://github.com/apache/commons-lang/blob/master/src/main/java/org/apache/commons/lang3/ArrayUtils.java`

As anticipated, no bugs were flagged when the wrapper, in conjunction with JBMC, was applied to the pristine versions of these files. This outcome was expected given the high quality and widespread scrutiny these classes have undergone, reinforcing their reliability in various applications.

Subsequently, to further probe the capabilities of our wrapper, we deliberately introduced errors into random methods within these classes. For instance, in `StringUtils.java`, we removed null check conditions in methods such as `isEmpty` or altered the index check logic in `substring`. Similarly, in `ArrayUtils.java`, we modified the boundary-checking logic in `getLength` and removed some null pointer checks in `add`. Specific examples include the omission of a null check in `StringUtils.isEmpty`, where the method should return true for a null input but, due to our modification, proceeded to evaluate the input's length, leading to a potential `NullPointerException`. In `ArrayUtils`, altering the boundary check in `getLength` could cause an `ArrayIndexOutOfBoundsException` under certain conditions. Our wrapper correctly flagged these scenarios, illustrating its utility in capturing and presenting such common yet critical errors.

These manipulations were intended to simulate common programming oversights [36], such as neglecting null checks or mishandling array bounds. Upon reapplying our wrapper and JBMC to these modified versions, the tools successfully identified the newly introduced flaws, generating counterexamples for each detected issue. These findings underscore the efficacy of our wrapper in conjunction with JBMC for detecting subtle yet significant errors within widely used utility classes.

4.5 Conclusions

The evaluation of the wrapper demonstrates its effective integration with JBMC, providing enhanced error detection and analysis capabilities while maintaining reasonable performance. In simple cases, the wrapper successfully identified various issues, with performance overheads remaining minimal. The SV-COMP-inspired tests further substantiated the wrapper's capability to handle complex verification tasks, albeit with a slight increase in execution time. Finally, testing with the Apache Commons Lang library confirmed the wrapper's practical utility in detecting potential bugs in widely used code bases.

Despite its success, some limitations were noted, including discrepancies in string handling and minor performance overhead. However, the benefits, particularly in terms of improved error detection and analysis depth, suggest that the wrapper represents a valuable tool for enhancing the efficacy of Java program verification with JBMC. The incremental overhead is justified by the extended capabilities and insights provided by the wrapper, highlighting its potential for broader adoption in software testing and verification workflows.

4.5.1 Results

The testing and evaluation of the wrapper, designed to augment the JBMC were conducted across various dimensions, including simple case analyses, SV-COMP-inspired benchmarks, and real-world applications using open-source code. In simple cases, the wrapper demonstrated a robust ability to identify and articulate common Java programming issues, such as string mishandling and arithmetic errors, by generating intuitive counterexamples.

When applied to well-tested open-source code-bases — specifically StringUtils.java and ArrayUtils.java - the wrapper proved its utility in identifying artificially introduced errors. This capability instils confidence in its real-world application, mirroring potential programming oversights.

The wrapper, despite introducing a consistent overhead of 10-15% in execution time, effectively communicated the absence of issues in bug-free code contexts. This reliability instils a sense of security, making the overhead acceptable, given the additional analytical depth and user-friendly output provided by the wrapper.

4.5.2 Threats to Validity

While the results underscore the wrapper's potential in enhancing JBMC's debugging capabilities, several threats to validity warrant attention. The observed limitations stem partly from the inherent constraints of JBMC, such as handling dynamic features and complex data structures, which the wrapper inherits. Additionally, implementation-specific limitations were noted, especially in parsing extensive string operations and managing complex dynamic object assignments in arrays, potentially affecting the granularity and utility of generated counterexamples.

Although thorough, the performance assessment was confined to a controlled set of test scenarios and may not fully extrapolate to all real-world applications, especially those with significantly larger code-bases or higher complexity. Moreover, the testing process did not encompass a comprehensive spectrum of Java's features, particularly more advanced constructs and the latest language updates, which could further challenge the wrapper's effectiveness.

5 Case Studies

5.1 JBMC Wrapper - Debugging Tool for Java Engineers

5.1.1 Overview

This case study aims to analyse whether the developed wrapper can be helpful as a debugging tool for Java Software Engineers in the industry. For this task, two Java Files of medium length and complexity were created to be used for the experiment. The experiments were conducted on five Java Backend Software Engineers with a range of two to ten years of experience (part of my engineering team at my workplace) at a large technology company.

5.1.2 Approach

The two Java Files created, found at Appendix 13 and at Appendix 14, were designed to be similar in complexity and contain similar types of bugs. Each file would contain eight methods, out of which five would contain bugs detectable by the JBMC when run with CPROVER and a default unwind value of 10, one of them would contain bugs that cannot be detected by the JBMC check, and the two remaining methods would contain no bugs.

The experiment would ask every engineer, individually, to first try to indicate which methods have potential bugs and what those bugs are for one of the Java files created without having access to the wrapper's results and to record their results. The time elapsed for this task would be documented for analysis purposes. For the second part of the task, the engineer would have access to the developed wrapper outputs based on the JBMC check and again should outline any possible bugs and their types based on the other Java file, recording their findings and the time that elapsed.

The Engineers were not told how many methods contained bugs or were correct to avoid biases and the introduction of additional knowledge.

5.1.3 Results

The results were collected in Table 1 and Table 2. Table 1 represents the data collected for the case when the Engineer had no access to the wrapper's output or any JBMC check. It can clearly be seen that one of the JBMC bugs posed a challenge to fully be undefined to 3 of the 5 Engineers, as well as another JBMC bug was not identified by one Engineer and only partially by another, while the non-JBMC bug had one partially correct identification, and the rest correct. The remaining were all fully correctly identified, and the average time spent on the exercise was 9 minutes.

	JBMC Bugs					Non-JBMC Bug	Correct		
	M1	M2	M3	M4	M5	M6	M7	M8	Total Time
Engineer 1	✓	✓	○	×	✓	✓	✓	✓	8.5 min
Engineer 2	✓	✓	✓	✓	✓	✓	✓	✓	9.5 min
Engineer 3	✓	✓	○	○	✓	✓	✓	✓	7.5 min
Engineer 4	✓	✓	○	✓	✓	○	✓	✓	8.5 min
Engineer 5	✓	✓	✓	✓	✓	✓	✓	✓	11 min
Score	5/5	5/5	3.5/5	4/5	5/5	4.5/5	5/5	5/5	Avg. Time: 9 min

Table 1. Bug identification effectiveness and time analysis with no input from the wrapper. Symbols represent: ✓ - fully identified (score 1), ○ - partially identified (score 0.5), × - not identified (score 0). Abbreviations: MN represents the Nth Method part of the file. Time approximated at 0.5 minutes

For the case when the Engineers were presented with the wrapper's output to aid the debugging process, it can clearly be seen in the Table 2 that the time taken to complete the task was significantly reduced, averaging at 3 minutes. Moreover, all the JBMC-related bugs were identified. Interestingly, the non-JBMC bug had a lesser identification score than the previous case.

	JBMC Bugs					Non-JBMC Bug	Correct		
	M1	M2	M3	M4	M5	M6	M7	M8	Total Time
Engineer 1	✓	✓	✓	✓	✓	×	✓	✓	3 min
Engineer 2	✓	✓	✓	✓	✓	✓	✓	✓	2 min
Engineer 3	✓	✓	✓	✓	✓	✓	✓	✓	3.5 min
Engineer 4	✓	✓	✓	✓	✓	○	✓	✓	3 min
Engineer 5	✓	✓	✓	✓	✓	✓	✓	✓	4 min
Score	5/5	5/5	5/5	5/5	5/5	3.5/5	5/5	5/5	Avg. Time: 3 min

Table 2. Bug identification effectiveness and time analysis with input from the wrapper. Symbols represent: ✓ - fully identified (score 1), ○ - partially identified (score 0.5), × - not identified (score 0). Abbreviations: MN represents the Nth Method part of the file. Time approximated at 0.5 minutes

5.1.4 Observations

Based on the above experimentation, it is clear that tools such as JBMC, equipped with an illustrative counterexample producer, can aid the debugging process for Java Engineers, produce better bug-identification accuracy, and shorten the analysis time by around a third.

However, also based on the analyses, it can be seen that when tools such as JBMC are used, the bugs not identified by the checker can be overlooked by the Engineer. Therefore, such checking bugs should be utilised with care, and the user should make sure to understand the capacities of the check and its limitations. The methods not flagged by the JBMC checker should not be automatically regarded as being correct. Therefore, the user should still check for potential logical or undetected bugs to ensure the program's correctness and not fully rely on the correctness of the JBMC equipped with the wrapper.

5.2 JBMC Wrapper - Debugging Tool for Students

5.2.1 Overview

The purpose of this case study is to investigate if the developed wrapper can help the debugging process for Computer Science Students with Java experience only through University exposure, not industrial experience with this programming language. The experiments were conducted on twelve Third-Year Students who completed a Java course in the first year of study but had limited to no Java experience outside that course unit scope.

5.2.2 Approach

The 80 Java Input files used as part of the Simple Cases for Testing and Evaluation Chapter were utilised, with an additional 10 programs that contain logical errors that the JBMC does not flag. Out of the 90 programs, pools of 10 programs were extracted randomly, just making sure that out of the 10 Java files, three contained non-JBMC bugs. Each participant is presented with three such pool programs, an ad for which they should identify the bug and measure the time elapsed for completing the task for a given pool of 10 programs. For the first ten programs, the participant would not have any access to any JBMC check; for the second pool of 10 programs, just the output of doing the JBMC check with ten unwind options would be available; and for the last pool, the wrapper's output with the counterexamples produced - ran with the default settings for each of the program.

Moreover, at the end of the task completion, the participant is presented with a small questionnaire for the whole set of 3 pools of programs.

5.2.3 Results

Based on the data collected in Table 3, it can be observed that the accuracy increased from pool to pool, with a major increment between pools two and three. Moreover, the time taken to analyse each pool decreases, as expected, and the difference between the first and third pools is reduced by two-thirds.

Moreover, in terms of the questionnaire, which can be found fully at Appendix B, the data collected shows that 75% of respondents chose either Very Helpful or Extremely Helpful when asked, ”How much did the JBMC check help compared to not having any formal verification tool in place?”, while for the question ”How beneficial was the presence of counterexamples compared to just having the JBMC output without them?”, 91% responded with either Very Beneficial or Extremely Beneficial. The full results of the questionnaire can be observed in Table 4.

	Pool 1				Pool 2 - JBMC Check				Pool 3 - Wrapper Check			
S	JBMC	Other	Acc.	Dur.	JBMC	Otehr	Acc.	Dur.	JBMC	Other	Acc.	Dur.
S1	4/7	1/3	50%	12	5/7	1/3	60%	10	7/7	1/3	80%	5
S2	3/7	2/3	50%	16	6/7	2/3	80%	14	7/7	2/3	90%	4
S3	2/7	0/3	20%	21	4/7	0/3	40%	8	7/7	2/3	90%	6
S4	5/7	2/3	70%	20	2/7	1/3	30%	12	7/7	0/3	70%	7
S5	6/7	1/3	70%	18	3/7	2/3	50%	15	7/7	3/3	100%	3
S6	2/7	1/3	30%	19	7/7	1/3	80%	6	7/7	0/3	70%	5
S7	3/7	2/3	50%	14	5/7	0/3	50%	9	5/7	0/3	50%	6
S8	5/7	3/3	80%	11	4/7	2/3	60%	16	7/7	1/3	80%	7
S9	4/7	0/3	40%	17	6/7	0/3	60%	13	7/7	1/3	80%	4
S10	3/7	3/3	60%	9	5/7	1/3	60%	11	7/7	2/3	90%	3
S11	2/7	1/3	30%	10	3/7	2/3	50%	7	5/7	1/3	60%	6
S12	6/7	2/3	80%	13	7/7	1/3	80%	16	6/7	1/3	70%	7
Avg	-	-	52.5%	15	-	-	58%	11	-	-	77.5%	5

Table 3. Analysis of student performance across the three pools. Abbreviations: SN represents the Nth Student part of the experiment. Other refers to programs that are not flagged by the JBMC. Acc. represent the accuracy and Dur. the duration. Time measured in minuted approximated at nearest whole minute.

Question	Not at all/Slightly	Moderately	Very	Extremely	Total
JBMC Help	0	3	5	4	12
Counterexamples Benefit	0	1	5	6	12
Understanding Enhancement	1	2	4	5	12
Future Use Likelihood	0	2	3	7	12

Table 4. Detailed summary of questionnaire responses.

5.2.4 Observations

Based on the results of code analyses and the questionnaire, it can clearly be seen that the use of the JBMC checker positively impacted students, increasing the accuracy and improving the time elapsed in completing the task. Notably, having the Counterexamples produced by the wrapper had an even greater impact on the statistics, as the counterexamples, in their intuitive and straightforward nature, helped the students identify the bug better and faster while understanding it based on the counterexample.

However, as the other Case Study also suggested, as the students were not familiar with the capacities and limitations of the JBMC checks, they did not double-check the programs that were not flagged by the JBMC, and the accuracy of that program's bug identification decreased. Therefore, the use of the JBMC alongside the developed wrapper has beneficial outcomes but should be used with caution and knowledge of the tool, and further manual checks should still be carried on to secure the code's correctness logically and syntactically.

5.3 Experimental Questions

This section is designed to bridge the theoretical frameworks and developmental narratives presented earlier with the conclusive insights we aim to establish. By addressing key experimental questions, these case studies provide a critical evaluation of the wrapper's performance, elucidating its strengths, potential limitations, and its overarching value to the Java programming community.

Q1: Does the JBMC wrapper improve the accuracy of bug detection in Java applications compared to traditional debugging methods?
Answer: Yes, the JBMC wrapper enhances bug detection accuracy. By converting JBMC's complex output into executable counterexamples, developers receive more precise insights into bugs. This methodological advancement enables more precise bug identification, particularly in complex code scenarios where traditional debugging might overlook subtle issues.

Q2: How does the JBMC wrapper affect the debugging time for Java developers?
Answer: The JBMC wrapper significantly reduces Java developers' debugging time. Providing intuitive and executable counterexamples allows developers to reproduce and analyse bugs quickly, thus streamlining the debugging process. In contrast to conventional methods that require extensive manual inspection, this automated and focused approach accelerates resolution times.

Q3: Can the JBMC wrapper aid in the educational context, particularly in helping students understand and fix bugs in Java programs?
Answer: Absolutely, the JBMC wrapper proves to be a valuable educational tool. It demystifies the debugging process for students by offering tangible examples of how and why specific bugs occur. This hands-on learning aid fosters a deeper understanding of Java programming nuances and debugging strategies, effectively supplementing theoretical instruction.

Q4: Are there any scalability limitations when using the JBMC wrapper for debugging larger Java applications?
Answer: While the JBMC wrapper is highly effective for various applications, scalability can pose challenges, especially for very large Java applications. The resource-intensive nature of model checking may lead to longer analysis times as application complexity grows. However, the wrapper maintains its efficacy and efficiency for most standard-sized applications.

Q5: How does the introduction of the JBMC wrapper influence the overall software development life-cycle regarding bug detection and resolution?
Answer: The JBMC wrapper introduces significant improvements in the software development lifecycle by facilitating earlier and more accurate bug detection, contributing to more reliable and robust software products. Reducing the time and effort required for debugging allows development teams to allocate more resources to feature development and optimisation, thus enhancing productivity and product quality.

5.4 Conclusions

Overall, the case studies affirm the value of the JBMC wrapper as a potent enhancement to debugging processes in both educational and professional environments. The wrapper accelerates bug identification and improves the precision of these identifications, facilitating a more thorough and informed debugging process. However, these studies also caution against over-reliance on automated tools; engineers and students tended to ignore potential errors in code sections deemed error-free by JBMC. This highlights the importance of complementary manual checks and a comprehensive understanding of the tool's scope and limitations.

In conclusion, while the JBMC wrapper is an effective aid in debugging, it should be integrated into broader verification and validation strategies that include manual code review and an awareness of the limitations inherent in automated debugging tools. This balanced approach ensures that software engineers and students benefit from the efficiency and accuracy enhancements provided by such tools and maintain rigorous standards for code quality and reliability.

6 Summary and Conclusions

6.1 Summary of the Work

The project embarked on developing a Java-based wrapper designed to augment the Java Bounded Model Checker (JBMC) [6] by generating intuitive counterexamples for detected bugs, thereby aiding in the debugging process. The primary focus was to enhance the usability of JBMC outputs, making the debugging process more accessible and efficient for Java developers. The wrapper was meticulously crafted to process Java files, interface seamlessly with JBMC, and output counterexamples that provide clear, actionable insights into the nature and context of detected bugs in a user-friendly manner. The counterexamples generated by the wrapper, processing the XML Trace produced by the JBMC checks, are runnable Java programs that are able to call the input file and simulate the detected bug.

Throughout the project, the wrapper was subjected to rigorous testing using a diverse array of test cases, including simple scenarios, SV-COMP-inspired benchmarks [8], [9], and real-world open-source code, such as Apache Commons Lang library. These evaluations were aimed at assessing the wrapper's effectiveness, its ability to generate meaningful counterexamples, and its performance in terms of execution time and resource utilisation.

Additionally, the project undertook two distinct case studies to explore the wrapper's utility in practical settings: one focusing on Java engineers in the industry and another on computer science students. These studies provided valuable insights into the wrapper's applicability, effectiveness, and potential areas for enhancement.

6.2 Achievements

The core achievement of this project is the development of the wrapper, which significantly strengthens the utility and applicability of the JBMC for debugging purposes. Here's a more detailed discussion of the specific characteristics of the wrapper's development and the achievements of this project:

- **Intuitive Counterexample Generation:** At the heart of the wrapper's functionality is its ability to transform the intricate output of JBMC into user-friendly, executable counterexamples. These counterexamples illustrate the exact conditions under which bugs occur, providing developers with a direct pathway to replicate and investigate the issues within their code-base. By extracting vital data such as input types, values, and structures from JBMC's XML output, the wrapper constructs detailed scenarios that demonstrate each bug, enhancing the comprehensibility of the debugging process.
- **Enhanced Usability:** The wrapper is designed with user experience in mind, featuring a straightforward interface that guides users through its operation. The wrapper removes significant complexity by automating several processes, such as the compilation of Java files and interaction

with JBMC, making advanced debugging techniques more accessible to a broader range of developers. Its interactive prompts and clear output messages ensure that users can easily navigate its features, adjust settings, and interpret results.

- **Adaptability and Flexibility:** The wrapper is built to accommodate various user preferences and requirements, acknowledging the diverse environments and scenarios in which Java debugging occurs. It offers customisation options, such as the ability to set the ’unwind’ parameter, which influences the depth of the analysis performed by JBMC.

- **Performance Optimisation:** While augmenting JBMC’s output with utilised counterexamples, the wrapper focuses on performance optimisation to minimise the overhead introduced during the analysis process. Rigorous testing and optimisation efforts have ensured that the wrapper operates efficiently, balancing its advanced functionality with optimisation and resource conservation in a debugging tool.

- **Robust Testing and Validation:** The wrapper has undergone extensive testing using a variety of benchmarks and real-world code samples. This rigorous evaluation demonstrates its reliability and effectiveness across different scenarios and provides a comprehensive assessment of its capabilities, highlighting areas where it excels and identifying opportunities for future enhancements.

- **Practical Application Studies:** The case studies provided concrete evidence of the wrapper’s utility in real-world scenarios. The wrapper expedited the bug identification process for Java engineers, enabling more efficient debugging. In the academic context, students found the wrapper to be an invaluable tool for understanding and resolving programming bugs, which underscored the wrapper’s educational potential.

- **Documentation:** The wrapper comes with comprehensive documentation detailing its installation, configuration, and usage. This resource is invaluable for users seeking to leverage the wrapper’s capacities.

6.3 Reflection

Upon reflecting on this project, it is evident that the developed wrapper significantly enhances JBMC’s utility, providing a more intuitive interface for debugging Java programs. The ability to generate clear, executable counterexamples marks a substantial advancement, offering developers immediate and comprehensible insights into the bugs within their code. While the wrapper has demonstrated robustness and utility across various tests, this reflection also acknowledges the challenges encountered, particularly in translating complex JBMC outputs into user-friendly formats. The endeavour has underscored the intricate balance between extending functionality and maintaining user accessibility.

Moreover, as a Java Developer myself, this project not only helped me understand the Java bytecode and underlying logic better but also helped me develop a tool that I can use in my day-to-day job and tasks. This tool would make the debugging process easy in some cases, as well established by the JBMC capacities and capabilities.

6.3.1 Main Challenges

One of the main challenges was taking the JBMC output and processing and manipulating it. As the JBMC tool does not have extensive documentation of the options that the tool can be run with, a significant amount of time was spent on attempting various forms of the JBMC's output until I discovered its capabilities to deliver an XML-format output.

The wrapper's initial development revolved around writing the code in Java as well. However, this resulted in a key challenge regarding the implementation and XML trace processing. Therefore, keeping in mind Python's capacities around XML [33], [34] Trace, I switched to developing the wrapper in Python, having to translate the initial functionality of user specification settings and compile the Input from Java to Python.

6.4 Future Work

Even if the wrapper itself has an enhancement impact on the JBMC checks, as demonstrated through the testing methodologies and case studies, there are still areas of improvement that would take the implementation to the next level.

Firstly, the implementation could contain even more flexibility in specifying the way that a wrapper can be run. A few of these features include, on the one hand, the ability to run off, for example, the CPROVER library, in case the user opts out of using it. On the other hand, as the JBMC supports a variety of settings, the project can ask the user to promote the settings with which the JBMC would be running, as well as to build a "–help" command, which would help the user navigate through all potential settings.

Secondly, for further clarity and intuitive counterexamples, the original names of the variables that have to be initialised by the counterexamples can be expressed by further exploring the XML trace. This would aid the user in directly mapping every variable from the counterexamples to the program.

Thirdly, in terms of main functionality, the wrapper can evolve in a way that accommodates better the String class checks and its generation of counterexamples, as well as making sure to provide a clear bug explanation on top of every counterexample generated.

Furthermore, the wrapper can be improved by enhancing its file structure implementation and design to accommodate all project structures even more. As this tool is in its initial setup and the programs that it checked were mostly mono-files or a small number of separate classes, the current implementation sufficed. However, for larger projects with intricate file structures, the wrapper could allow for more flexibility on where the checked Inputs should be placed and how the wrapper should be run.

Lastly, to further improve its usability and user-friendly approach, the wrapper can be transformed into either a Graphical User Interface (GUI) application for the use of smaller files, in order to enhance the user experience, or as an importable tool for the leading Java IDEs such as IntelliJ or Eclipse. This tool would be an automatic one, being able to run the JBMC check on a given class based on some settings set in either pom.xml or the environment-specific settings, flagging any potential issues with a given code if the user runs it, and being able to generate, at a set location, counterexamples which can be compared and further analysed to understand the bug. Even if this task is more complex to implement, it will transform the wrapper into an integral tool for Java Verification, providing not only possible flagging of the bugs but also tangible counterexamples in the project's environment.

References

[1] L. S. Vailshery. “Most widely utilized programming languages among developers worldwide 2023.” Accessed: 29/02/2024. (Jan. 2024), [Online]. Available: `https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/` (cited on p. 6).

[2] J. Gosling, “Java™: An overview,” *Recuperado d e http://www. cs. dartmouth. edu/~ mckeeman/cs118/references/OriginalJavaWhitep aper. pdf*, 1995 (cited on p. 6).

[3] K.-h. Chang, V. Bertacco, and I. L. Markov, “Simulation-based bug trace minimization with bmc-based refinement,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 1, pp. 152–165, 2006 (cited on p. 6).

[4] R. Brenguier, L. Cordeiro, D. Kroening, and P. Schrammel, “Jbmc: A bounded model checking tool for java bytecode,” *arXiv preprint arXiv:2302.02381*, 2023 (cited on pp. 6, 11–14, 24, 30).

[5] P. Kesseli and M. Trtik, “Jbmc: A bounded model checking tool for verifying java bytecode,” 2018 (cited on pp. 7, 11–14, 24, 30).

[6] L. Cordeiro, P. Kesseli, D. Kroening, P. Schrammel, and M. Trtik, “Jbmc: A bounded model checking tool for verifying java bytecode,” in *International Conference on Computer Aided Verification*, Springer, 2018, pp. 183–190 (cited on pp. 7, 10–14, 16, 24, 30, 45).

[7] L. Cordeiro, D. Kroening, and P. Schrammel, “Jbmc: Bounded model checking for java bytecode: (competition contribution),” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2019, pp. 219–223 (cited on pp. 7, 10–14, 24).

[8] D. Beyer, “Software verification: 10th comparative evaluation (sv-comp 2021),” in *Tools and Algorithms for the Construction and Analysis of Systems: 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27–April 1, 2021, Proceedings, Part II 27*, Springer, 2021, pp. 401–422 (cited on pp. 7, 12, 29, 35, 45).

[9] D. Beyer, “Progress on software verification: Sv-comp 2022,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2022, pp. 375–402 (cited on pp. 7, 12, 29, 35, 36, 45).

[10] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, and Y. Zhu, “Bounded model checking.,” *Handbook of satisfiability*, vol. 185, no. 99, pp. 457–481, 2009 (cited on pp. 8, 10).

[11] K. Heljanko and T. Junttila, “Advanced tutorial on bounded model checking (bmc),” (cited on p. 8).

[12] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, “Bounded model checking using satisfiability solving,” *Formal methods in system design*, vol. 19, pp. 7–34, 2001 (cited on pp. 8, 9).

[13] A. Armando, J. Mantovani, and L. Platania, “Bounded model checking of software using smt solvers instead of sat solvers,” *International Journal on Software Tools for Technology Transfer*, vol. 11, pp. 69–83, 2009 (cited on pp. 8, 9).

[14] K. Heljanko and I. Niemelä, “Bounded ltl model checking with stable models,” *Theory and Practice of Logic Programming*, vol. 3, no. 4-5, pp. 519–550, 2003 (cited on pp. 8, 9).

[15] G. Cabodi, A. Kondratyev, L. Lavagno, S. Nocco, S. Quer, and Y. Watanabe, “A bmc-based formulation for the scheduling problem of hardware systems,” *International Journal on Software Tools for Technology Transfer*, vol. 7, pp. 102–117, 2005 (cited on pp. 8–10).

[16] A. Derhab, M. Guerroumi, M. Belaoued, and O. Cheikhrouhou, “Bmc-sdn: Blockchain-based multicontroller architecture for secure software-defined networks,” *Wireless Communications and Mobile Computing*, vol. 2021, pp. 1–12, 2021 (cited on pp. 8–10).

[17] B. Dutertre and L. De Moura, “The yices smt solver,” *Tool paper at http://yices. csl. sri. com/tool-paper. pdf*, vol. 2, no. 2, pp. 1–2, 2006 (cited on p. 9).

[18] K. Y. Rozier, “Linear temporal logic symbolic model checking,” *Computer Science Review*, vol. 5, no. 2, pp. 163–203, 2011 (cited on p. 9).

[19] V. Rybakov, “Linear temporal logic with until and next, logical consecutions,” *Annals of Pure and Applied Logic*, vol. 155, no. 1, pp. 32–45, 2008 (cited on p. 9).

[20] A. Biere, K. Heljanko, T. Junttila, T. Latvala, and V. Schuppan, “Linear encodings of bounded ltl model checking,” *Logical Methods in Computer Science*, vol. 2, 2006 (cited on p. 10).

[21] D. Kroening and M. Tautschnig, “Cbmc–c bounded model checker: (competition contribution),” in *Tools and Algorithms for the Construction and Analysis of Systems: 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings 20*, Springer, 2014, pp. 389–391 (cited on pp. 10, 11).

[22] L. Cordeiro, D. Kroening, and P. Schrammel, “Benchmarking of java verification tools at the software verification competition (sv-comp),” *arXiv preprint arXiv:1809.03739*, 2018 (cited on pp. 11, 12, 14, 24).

[23] D. Beyer, “Competition on software verification and witness validation: Sv-comp 2023,” in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2023, pp. 495–522 (cited on pp. 12, 29, 35).

[24] J. Gennari, A. Gurfinkel, T. Kahsai, J. A. Navas, and E. J. Schwartz, “Executable counterexamples in software model checking,” in *Verified Software. Theories, Tools, and Experiments: 10th International Conference, VSTTE 2018, Oxford, UK, July 18–19, 2018, Revised Selected Papers 10*, Springer, 2018, pp. 17–37 (cited on pp. 12, 13).

[25] A. Groce and D. Kroening, “Making the most of bmc counterexamples,” *Electronic Notes in Theoretical Computer Science*, vol. 119, no. 2, pp. 67–81, 2005 (cited on pp. 12, 13).

[26] A. P. Kaleeswaran, A. Nordmann, T. Vogel, and L. Grunske, “A systematic literature review on counterexample explanation,” *Information and Software Technology*, vol. 145, p. 106 800, 2022 (cited on pp. 12, 13).

[27] H. Maruyama, *XML and Java: developing Web applications*. Addison-Wesley Professional, 2002 (cited on p. 13).

[28] J. R. Gardner, J. R. Gardner, and Z. L. Rendon, *XSLT and XPATH: a Guide to XML Transformations*. Prentice Hall Professional, 2002 (cited on p. 13).

[29] T. M. Chester, “Cross-platform integration with xml and soap,” *IT Professional*, vol. 3, no. 5, pp. 26–34, 2001 (cited on p. 13).

[30] B. Marchal, *XML by Example*. Que Publishing, 2002 (cited on p. 16).

[31] T. Wu, “Develop and evaluate a security analyser for finding vulnerabilities in java programs,” (cited on p. 16).

[32] C. A. Jones and F. L. Drake Jr, *Python & XML: XML Processing with Python*. ” O’Reilly Media, Inc.”, 2001 (cited on p. 16).

[33] W. Python, “Python,” *Python releases for windows*, vol. 24, 2021 (cited on pp. 16, 47).

[34] I. Mlỳnková, “Xml benchmarking: Limitations and opportunities,” *Technical Report, Department of Software Engineering, Charles University*, 2008 (cited on pp. 25, 47).

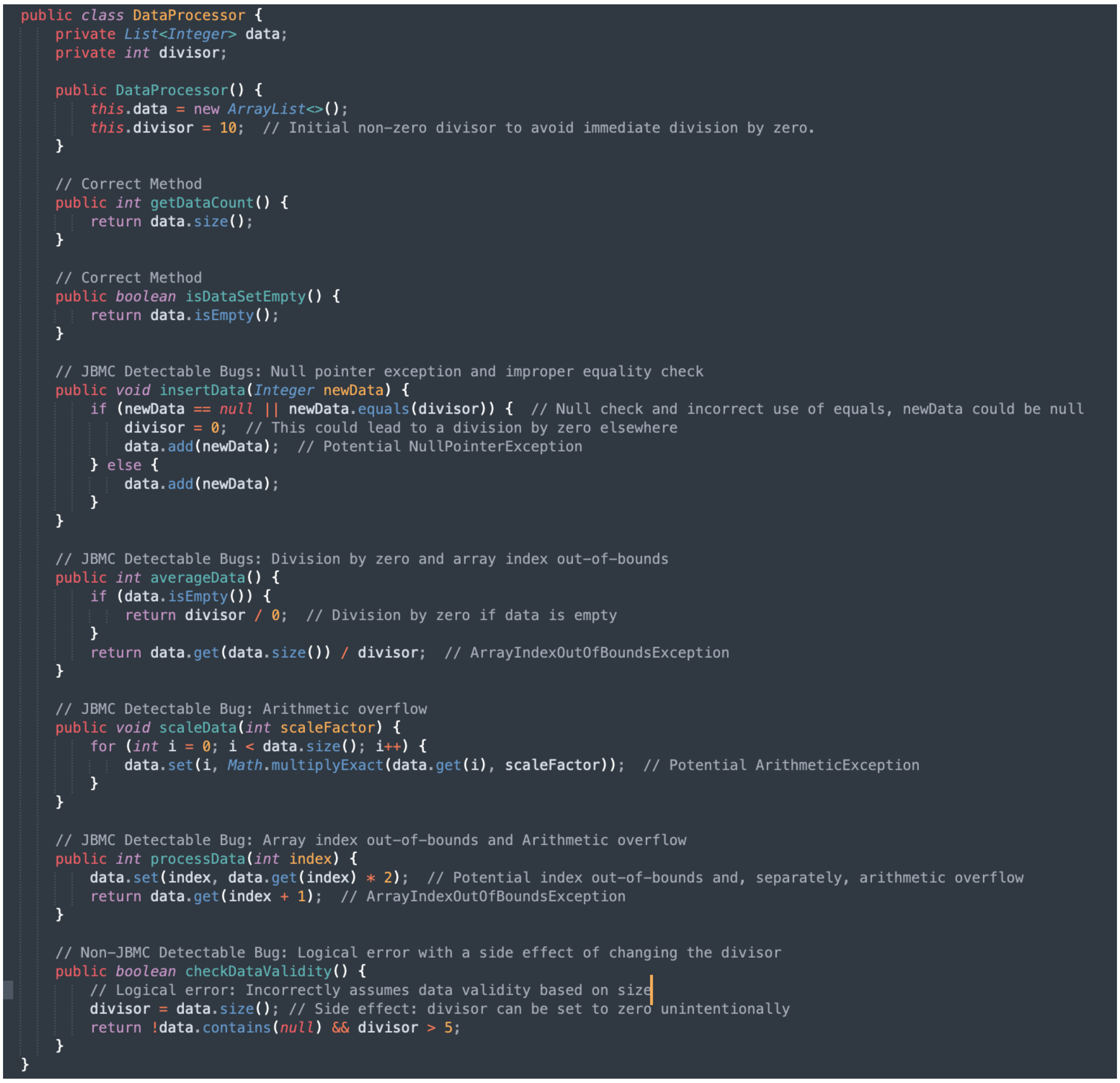
[35] M. Měchura, “Better than xml: Towards a lexicographic markup language,” *Data & Knowledge Engineering*, vol. 146, p. 102 196, 2023 (cited on p. 25).

[36] R. K. Saha, Y. Lyu, W. Lam, H. Yoshida, and M. R. Prasad, “Bugs. jar: A large-scale, diverse dataset of real-world java bugs,” in *Proceedings of the 15th international conference on mining software repositories*, 2018, pp. 10–13 (cited on p. 37).

Appendices

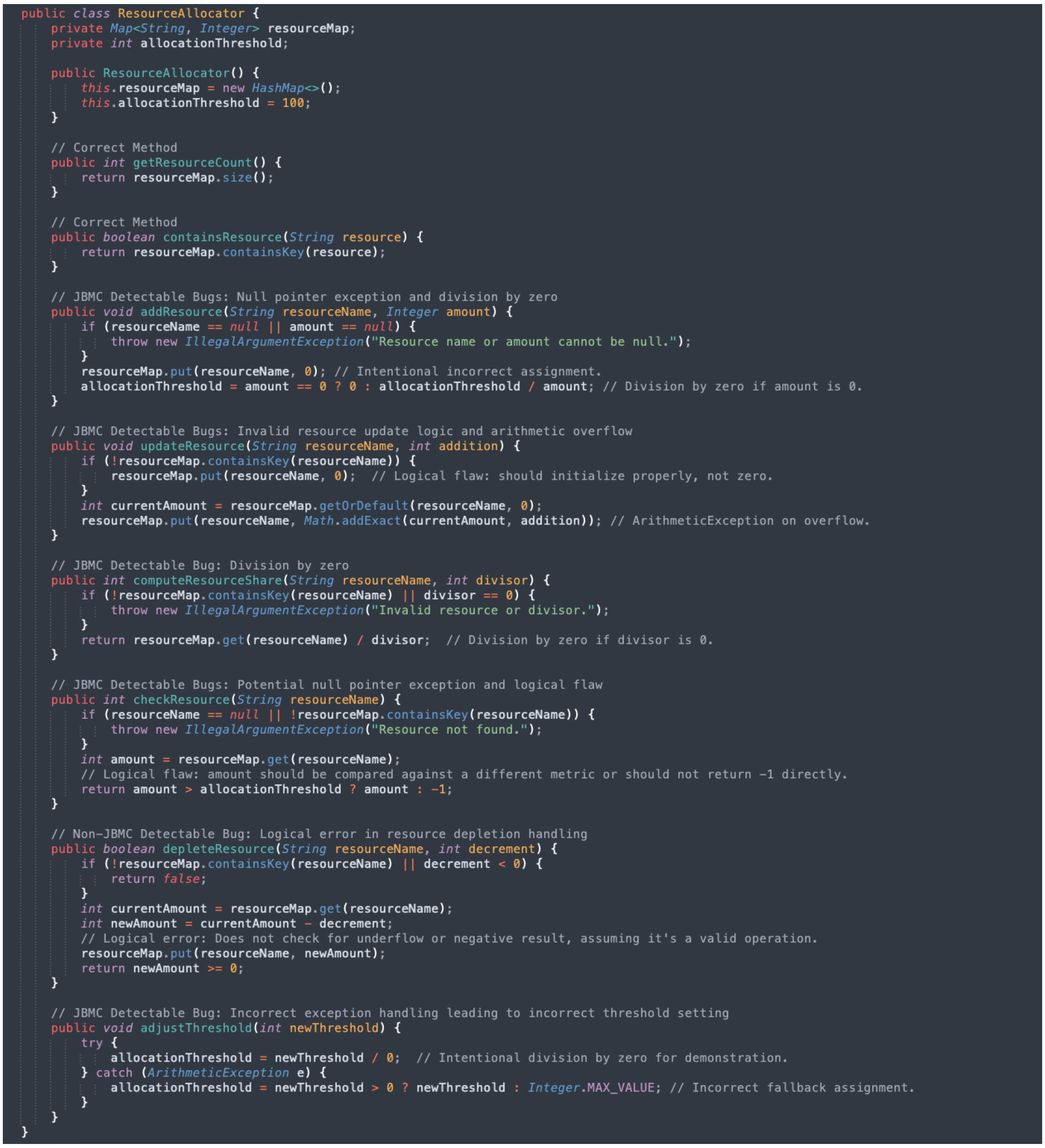
A JBMC Wrapper - Debugging Tool for Java Engineers - Code Snippets

It is to be noted that these Java files were modified with comments in order to aid the understanding of the Experiment. During the case study, the Engineer was presented with the methods with no comments to avoid biases and help correct analyses of the output.



```java
public class DataProcessor {
    private List<Integer> data;
    private int divisor;

    public DataProcessor() {
        this.data = new ArrayList<>();
        this.divisor = 10;  // Initial non-zero divisor to avoid immediate division by zero.
    }

    // Correct Method
    public int getDataCount() {
        return data.size();
    }

    // Correct Method
    public boolean isDataSetEmpty() {
        return data.isEmpty();
    }

    // JBMC Detectable Bugs: Null pointer exception and improper equality check
    public void insertData(Integer newData) {
        if (newData == null || newData.equals(divisor)) {  // Null check and incorrect use of equals, newData could be null
            divisor = 0;  // This could lead to a division by zero elsewhere
            data.add(newData);  // Potential NullPointerException
        } else {
            data.add(newData);
        }
    }

    // JBMC Detectable Bugs: Division by zero and array index out-of-bounds
    public int averageData() {
        if (data.isEmpty()) {
            return divisor / 0;  // Division by zero if data is empty
        }
        return data.get(data.size()) / divisor;  // ArrayIndexOutOfBoundsException
    }

    // JBMC Detectable Bug: Arithmetic overflow
    public void scaleData(int scaleFactor) {
        for (int i = 0; i < data.size(); i++) {
            data.set(i, Math.multiplyExact(data.get(i), scaleFactor));  // Potential ArithmeticException
        }
    }

    // JBMC Detectable Bug: Array index out-of-bounds and Arithmetic overflow
    public int processData(int index) {
        data.set(index, data.get(index) * 2);  // Potential index out-of-bounds and, separately, arithmetic overflow
        return data.get(index + 1);  // ArrayIndexOutOfBoundsException
    }

    // Non-JBMC Detectable Bug: Logical error with a side effect of changing the divisor
    public boolean checkDataValidity() {
        // Logical error: Incorrectly assumes data validity based on size
        divisor = data.size(); // Side effect: divisor can be set to zero unintentionally
        return !data.contains(null) && divisor > 5;
    }
}
```

Fig. 13. Code Snippet 1 used for Experimentation



```java
public class ResourceAllocator {
    private Map<String, Integer> resourceMap;
    private int allocationThreshold;

    public ResourceAllocator() {
        this.resourceMap = new HashMap<>();
        this.allocationThreshold = 100;
    }

    // Correct Method
    public int getResourceCount() {
        return resourceMap.size();
    }

    // Correct Method
    public boolean containsResource(String resource) {
        return resourceMap.containsKey(resource);
    }

    // JBMC Detectable Bugs: Null pointer exception and division by zero
    public void addResource(String resourceName, Integer amount) {
        if (resourceName == null || amount == null) {
            throw new IllegalArgumentException("Resource name or amount cannot be null.");
        }
        resourceMap.put(resourceName, 0); // Intentional incorrect assignment.
        allocationThreshold = amount == 0 ? 0 : allocationThreshold / amount; // Division by zero if amount is 0.
    }

    // JBMC Detectable Bugs: Invalid resource update logic and arithmetic overflow
    public void updateResource(String resourceName, int addition) {
        if (!resourceMap.containsKey(resourceName)) {
            resourceMap.put(resourceName, 0);  // Logical flaw: should initialize properly, not zero.
        }
        int currentAmount = resourceMap.getOrDefault(resourceName, 0);
        resourceMap.put(resourceName, Math.addExact(currentAmount, addition)); // ArithmeticException on overflow.
    }

    // JBMC Detectable Bug: Division by zero
    public int computeResourceShare(String resourceName, int divisor) {
        if (!resourceMap.containsKey(resourceName) || divisor == 0) {
            throw new IllegalArgumentException("Invalid resource or divisor.");
        }
        return resourceMap.get(resourceName) / divisor;  // Division by zero if divisor is 0.
    }

    // JBMC Detectable Bugs: Potential null pointer exception and logical flaw
    public int checkResource(String resourceName) {
        if (resourceName == null || !resourceMap.containsKey(resourceName)) {
            throw new IllegalArgumentException("Resource not found.");
        }
        int amount = resourceMap.get(resourceName);
        // Logical flaw: amount should be compared against a different metric or should not return -1 directly.
        return amount > allocationThreshold ? amount : -1;
    }

    // Non-JBMC Detectable Bug: Logical error in resource depletion handling
    public boolean depleteResource(String resourceName, int decrement) {
        if (!resourceMap.containsKey(resourceName) || decrement < 0) {
            return false;
        }
        int currentAmount = resourceMap.get(resourceName);
        int newAmount = currentAmount - decrement;
        // Logical error: Does not check for underflow or negative result, assuming it's a valid operation.
        resourceMap.put(resourceName, newAmount);
        return newAmount >= 0;
    }

    // JBMC Detectable Bug: Incorrect exception handling leading to incorrect threshold setting
    public void adjustThreshold(int newThreshold) {
        try {
            allocationThreshold = newThreshold / 0;  // Intentional division by zero for demonstration.
        } catch (ArithmeticException e) {
            allocationThreshold = newThreshold > 0 ? newThreshold : Integer.MAX_VALUE; // Incorrect fallback assignment.
        }
    }
}
```

Fig. 14. Code Snippet 2 used for Experimentation

B JBMC Wrapper - Debugging Tool for Students - Questionnaire

Please indicate your level of agreement with the following statements regarding the JBMC checks conducted during the exercise:

Q1. How much did the JBMC check help compared to not having any formal verification tool in place?

☐ Not at all helpful

☐ A bit helpful

☐ Moderately helpful

☐ Very helpful

☐ Extremely helpful

Q2. How beneficial was the presence of counterexamples compared to just having the JBMC output without them?

☐ Not Beneficial

☐ Slightly beneficial

☐ Moderately beneficial

☐ Very beneficial

☐ Extremely beneficial

Q3. To what extent did the JBMC checks enhance your understanding of the software's behaviour?

☐ Did not enhance at all

☐ Enhanced a little

☐ Moderately enhanced

☐ Significantly enhanced

☐ Fully enhanced my understanding

Q4. How likely are you to use JBMC or a similar formal verification tool in your future projects?

☐ Very unlikely

☐ Unlikely

☐ Neutral

☐ Likely

☐ Very Likely