Programming Language Support for Platform Testing

Areen Vaghasiya
IMT2022048
areen.vaghasiya@iiitb.ac.in

Aryan Vaghasiya
IMT2022046
aryan.vaghasiya@iiitb.ac.in

Siddharth Palod
IMT2022002
Siddharth.Palod@iiitb.ac.in

May 7, 2025

Chapter 1

Problem Description

Modern software platforms require rigorous testing methodologies to ensure reliability, especially in critical systems. However, symbolic and concolic execution tools often struggle to support newer programming constructs or annotations like JML (Java Modeling Language), resulting in inefficiencies in test case generation and code coverage analysis.

1.1 Introduction to Platform Testing

1.1.1 Current Issues

Modern platform software exhibits a high degree of architectural complexity, often composed of numerous interdependent components and diverse programming constructs. This complexity poses several challenges for conventional testing methodologies:

- **Limited Support for Annotations:** Symbolic and concolic execution tools frequently lack compatibility with specification languages such as JML (Java Modeling Language). This reduces their effectiveness in accurately interpreting the intended behavior of Java programs.
- **Manual Effort and Inefficiency:** Traditional test case generation requires significant manual effort to identify paths, generate input sets, and verify outputs—particularly in large-scale Java applications. This results in time-consuming and error-prone processes.

- **Lack of Integration with Programming Constructs:** Newer programming paradigms and custom annotations (like those provided by JML) are not well-supported in mainstream tools. This limits automated test generation from annotated code.

1.1.2 Focus areas

We intend to explore automated test case generation for platforms software. We aim to implement a Java based code generator for testing software, e.g., image processing, library, smart TV and gaming stations (e.g. Sony Play Station).

1.2 Core Learning Objectives

- Understand how programming languages support automated test generation.
- Explore symbolic and concolic testing using JPF and JML.
- Learn the basics of Design by Contract and JML annotations.
- Study AST manipulation and annotation compatibility in Java.

1.3 Intended Outcomes

- Evaluate the integration of JML with AST-based code traversal.
- Develop a reusable testing pipeline for Java libraries.
- Automated generation of test inputs through symbolic execution.

Chapter 2

Solution Outline

2.1 Overall Solution

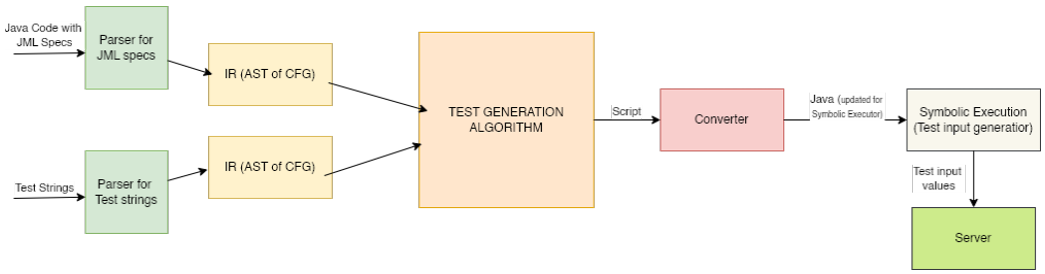

Figure 2.1: Test Generation Flow

We have divided the project into 6 main phases

- Phase 1: Finding JML specifications with DbC (Design by contract) and Test Input Strings for the Platform application
- Phase 2: Utilizing a parser which will provide us an Abstract syntax tree (AST) for given JML specifications and Test Input strings as we know from program analysis that Context free grammar (CFG) can be represented as a AST.
- Phase 3: Figure our how to traverse the AST for creation of a script which will include all *function calls*, *variables* and *automatic variable creation* based on requirements following DbC and Visitor Design pattern via Black box testing.

- Phase 4: Converting the script into a java code which can be given to the symbolic executing engine for test inputs.
- Phase 5: Finding a symbolic execution engine for test generation (added bonus for concolic execution) and utilizing it to run the java code and gain insights of path coverage and test inputs
- Phase 6: Effectively test the test inputs generated on the server side of the Platform application via White box testing and future endeavors include converting the flow into mock generation functions for thorough black box testing.

2.2 Deliverables

Our deliverables are structured as follows:

- **Comprehensive Report** — A detailed document covering the algorithm design, adopted methodologies, explored tools, and key insights. It also includes workflow rationale and impact analysis.
- **Annotated Codebase** — A manually crafted Java test script integrated with symbolic execution, accompanied by in-line documentation to explain behavior and integration points.
- **Test Generation Scripts** — Drafted test scripts and a conceptual design of an AST- and JML-based algorithm for automated test generation, intended for symbolic execution.
- **Architecture Models** —
  - Diagram outlining the complete algorithm with modular segmentation for individual development.
  - Example-driven diagram showcasing the workflow and execution path of the algorithm.
  - Visual representation of AST traversal strategy and intermediate stages.

2.3 Approach

Our solution for enabling platform-specific automated testing in Java revolves around a six-phase framework that integrates formal specifications, AST-based code analysis, and symbolic execution. The approach is modular and extensible, addressing key language-specific testing challenges through the following phases:

2.3.1 Phase 1: Specification and Input Extraction

The initial phase of our testing framework focuses on extracting formal behavioral specifications from Java source code using **Java Modeling Language (JML)** annotations. This step is essential for driving automated test case generation, particularly in black-box testing scenarios where internal implementation details are hidden.

1.1 Design by Contract (DbC)

Our approach is grounded in the principle of *Design by Contract (DbC)*, a software correctness methodology in which functions (methods) are treated as parties to a contract. Each method specifies:

- **Preconditions** (`@requires`): Conditions that must be true before the method is called.
- **Postconditions** (`@ensures`): Conditions that must hold true after the method execution.
- **Invariants**: Properties that remain true throughout the lifetime of a class or object.

These contracts formalize the expected behavior of methods and are critical for generating meaningful and valid test cases.

1.2 Role of JML Annotations

JML is a formal specification language tailored for Java, which allows the inclusion of contracts directly in the source code using structured comments. For example:

```
//@ requires balance >= 0;
//@ ensures balance == \old(balance) + depositAmount;
```

The use of JML allows automated tools to interpret the intent behind the code and generate relevant test inputs. It also serves as the foundation for detecting constraint violations, making it particularly useful in safety-critical and high-assurance software environments.

1.3 Input Extraction Strategy

We extract specifications using a combination of parsing and static analysis:

- Parse JML annotations using tools like `JMLParser` to build an annotated Abstract Syntax Tree (AST).
- Identify input parameters from method signatures and link them with corresponding JML preconditions.
- Construct a formal model of input constraints that will guide the test case generator in later stages.

For instance, a method annotated with `//@ requires x > 0 && x < 10;` leads to the extraction of an input range constraint for the variable `x`, which the test generator can later use to create both valid and boundary test cases.

1.4 Importance in Specifications

Since our framework focuses on black-box testing using specifications, where internal code logic is not directly accessible, having rich specifications becomes indispensable. These specifications define the expected input domain and output behavior, enabling:

- **Formal reasoning** about input/output relationships.
- **Automated edge-case generation**, especially around boundary constraints.
- **Symbolic execution path constraints** to be correctly inferred and satisfied.

By extracting and leveraging JML annotations at this early stage, we lay the foundation for the systematic and rigorous analysis that follows in later phases.

2.3.2 Phase 2: AST Generation via Parsing

The second phase of our testing framework involves parsing Java source code—augmented with JML annotations—to construct an **Abstract Syntax Tree (AST)**. This tree serves as an intermediate representation (IR) of the code, essential for structural analysis, specification extraction, and eventual test generation.

2.1 Motivation for Using ASTs

2.2 Intermediate Representations (IR) and Code Abstraction

We treat ASTs as a form of **Intermediate Representation (IR)**. IRs abstract away syntactic noise and provide a structured, analyzable form of code. Unlike raw source code, IRs can be easily traversed, manipulated, and used in tooling pipelines for optimization, refactoring, or test generation.

2.3 Parser Selection and Usage

To construct ASTs from Java source files, we evaluated and utilized the following parsing tools:

Tool	**AST Type**	**JML Support**	**CFG/Call Graph**	**Use Case**
JavaParser	Syntactic AST	High	No	Fast parsing, input string generation
Spoon	Semantic AST (CtModel)	Moderate	Yes	Mutation and instrumentation
JMLParser	Annotated AST	Limited	No	Handling JML contracts in AST
Soot	IR-Level (Bytecode)	Low	Yes	Symbolic backend, bytecode-level control

Table 2.1: Comparison of Tools for AST Generation and Analysis

JavaParser was chosen for rapid parsing and AST construction due to its high usability and syntactic clarity. For processing JML annotations specifically, **JMLParser** was employed to integrate contracts into the AST structure.**JavaParser is just an extension of**

2.4 Parser Workflow and AST Construction

The parser processes each source file and produces an AST where:

- **Class declarations** are captured as top-level nodes.
- **Method declarations and parameters** become child nodes.
- **JML annotations** are associated with corresponding method nodes.
- **Control structures** like conditionals and loops are recursively structured.

This tree-based representation enables modular and recursive analysis. For instance, one can extract all preconditions from a method by simply visiting its annotation nodes.

2.5 Integration with JML

A key challenge in this phase was maintaining a consistent mapping between JML annotations and the Java code structure. JMLParser helped in generating a merged AST that retains both semantic (Java) and specification (JML) information. This enriched AST is essential for the later symbolic execution phase, where path conditions rely on both code logic and specified contracts.

2.3.3 Phase 3: AST Traversal and Test Script Generation

This phase builds upon the annotated Abstract Syntax Trees (ASTs) constructed in Phase 2 to systematically generate test scripts.

3.1 Motivation and Objectives

The primary objective of this project is to automate the generation of test scripts by analyzing Java source code annotated with JML using an Abstract

Syntax Tree (AST)-based approach. The AST serves as a structured representation of the program, enabling systematic extraction and transformation.

- Use the Visitor Design Pattern to traverse the AST of Java source code, which includes JML annotations. This allows us to systematically collect variables, method definitions, function calls, and their associated preconditions and postconditions.
- Extract formal contracts defined in JML (e.g., `requires`, `ensures`) during traversal, which are essential for generating valid test cases under specified constraints.
- Represent test scenarios using an Intermediate Representation (IR) of test strings. A separate visitor is used to traverse this IR, enabling mapping between the test inputs and the code constructs identified from the AST.
- Leverage the extracted information to filter and trim unnecessary paths during symbolic execution, optimizing the testing process by focusing only on relevant and specification-conforming branches.
- Automatically generate Java-based test scripts that simulate program behavior under both normal and boundary conditions, guided by the specifications and control paths identified.

This visitor-driven analysis over both source code and test IR enables a modular, scalable, and contract-aware approach to automated test generation.

3.2 AST Traversal with Visitor Pattern

We employ the **Visitor Design Pattern** for AST traversal. This pattern separates data structure (AST) from operations performed on it (test generation logic). Each type of AST node—such as method declarations, if-statements, or annotations—has a corresponding visit method. This allows for a clean and modular traversal mechanism.

Example structure:

```
public class TestScriptVisitor extends VoidVisitorAdapter<Void> {
    @Override
```

```
    public void visit(MethodDeclaration md, Void arg) {
        // extract JML specs
        // generate parameter combinations
        // create assert/assume statements
    }
}
```

3.4 Functional Block Extraction

The traversal also detects:

- Control flow elements (if-else, loops, switch)
- Function calls and return values
- Exception paths

These blocks are modeled into corresponding test steps, preserving the execution semantics. For example, loop structures may lead to multiple test scenarios with varied input lengths or iterations.

3.5 Script Generation Output

For each method, we generate:

- Parameterized function calls
- Assertions for postconditions
- Exception catch blocks for robustness

A sample generated script may look like:

```
@Test
public void testDeposit() {
    BankAccount acc = new BankAccount();
    acc.deposit(100);
    assertEquals(100, acc.getBalance());
}
```

3.6 Current Limitations and Future Enhancements

While effective, the current traversal and generation mechanism has several limitations:

- Does not handle higher-order functions or method references.
- Pass-by-reference and pointer-like behaviors are not modeled.
- Test mocking frameworks are not yet integrated.

Future Work: We plan to integrate mocking to simulate I/O and server-side behavior, allowing for broader test automation. We also aim to support more complex control flows and state-based testing using symbolic path conditions.

3.7 Transition to Phase 4

The output of this phase—generated test scripts in intermediate form—are passed to Phase 4, where they are transformed into full-fledged Java test classes with configurations suitable for symbolic execution engines like JPF/SPF.

2.3.4 Phase 4: Script-to-Java Conversion

In this phase, the test scripts generated in intermediate form from AST traversal are transformed into fully structured and compilable **Java test classes**. This transformation is essential to make the test scripts compatible with downstream symbolic execution tools such as Java PathFinder (JPF) and Symbolic PathFinder (SPF).

4.1 Motivation

Test scripts initially generated are abstract representations of method calls and assertions. These must be converted into syntactically correct Java code, adhering to:

- Java syntax rules and type constraints.
- Framework requirements for symbolic execution (e.g., test entry points, parameter declarations).

- Integration with JML specifications and symbolic engines like SPF.

This phase bridges the gap between code analysis (AST-based generation) and actual execution on testing infrastructure.

4.2 Conversion Process

1. **Concrete Strings Replaced with Symbolic Inputs:**
   - **Before:** Used fixed strings for inputs (book titles, student names/IDs).
   - **After:** Used `Debug.makeSymbolicString()` to make inputs symbolic for automated exploration.
2. **Valid Preconditions Replaced with Symbolic Assumptions:**
   - **Before:** Checked for nulls or initialized objects manually.
   - **After:** Used `@assume` clauses (e.g., `@assume book != null;`) to constrain input space.
3. **Explicit Book Addition Abstracted:**
   - **Before:** Added each book explicitly via `addBookFunction()`.
   - **After:** Used symbolic functions and assumptions to infer book presence.
4. **Manual Book Return Flow Replaced with Symbolic Path:**
   - **Before:** Manually configured book with return details.
   - **After:** Simulated return flow with symbolic calls and assumptions on state.
5. **Shared Instances Replaced with Symbolic Structures:**
   - **After:** Replaced collections with symbolic arrays or structures to let the symbolic engine track changes.

4.4 Integration with Symbolic Executors

To integrate with tools like SPF or JPF, we generate configuration files and symbolic setup:

- **.jpf configuration files**: Specify the main class, classpath, and property settings for symbolic analysis (e.g., symbolic variables, listeners).
- **Symbolic Declarations**: SPF requires marking symbolic variables via command-line or configuration (e.g., `symbolic.method=BankAccount.deposit(sym#int)`).

4.5 Challenges and Considerations

- **Simulating External State**: Real-world programs often interact with files, databases, or networks. We isolate such dependencies using mocking or abstraction layers.
- **Handling Non-Determinism**: Input-dependent behavior or randomization must be controlled to enable deterministic symbolic execution.
- **Maintaining Specification Semantics**: While converting, we ensure that the logic defined by JML contracts is preserved and directly testable.

4.6 Output of Phase 4

The output is a set of Java test classes and configuration files that:

- Conform to Java syntax and symbolic execution tool standards.
- Encapsulate test logic defined by formal specifications (JML).
- Are fully ready for execution under tools like JPF/SPF.

This phase finalizes the test code structure, enabling it to be fed directly into the symbolic analysis framework in the next phase.

2.3.5 Phase 5: Symbolic Execution Engine Integration

This phase focuses on the core of our test automation framework—leveraging symbolic execution engines to analyze Java programs with JML specifications. Symbolic execution is a powerful technique that allows systematic exploration of program paths by treating inputs as symbolic variables rather than concrete values.

5.1 What is Symbolic Execution?

Symbolic execution is a program analysis technique where program inputs are treated as *symbols* rather than actual values. As the program executes, expressions involving these symbols are maintained, and a **path condition (PC)** is built to represent the constraints required to follow that execution path.

Example:

```
if (x > 5) {
   y = x + 1;
}
```

In symbolic execution, the engine forks:

- Path 1: $x > 5$, with $y = x + 1$
- Path 2: $x \leq 5$, block skipped

Each path maintains a unique PC, and satisfiability solvers (e.g., SMT solvers) are used to determine concrete inputs that meet those constraints, thus providing automated test inputs.

5.2 Why Symbolic Execution for Platform Testing?

Symbolic execution is particularly suitable for:

- **Black-box testing** guided by formal specifications (JML contracts).
- **High coverage testing** through exploration of all feasible code paths.
- **Edge-case detection**, especially those that might not be discovered through conventional testing.

For platform-specific systems (e.g., embedded, graphical apps), symbolic execution helps ensure correctness in constrained, unpredictable environments.

5.3 Symbolic and Concolic Execution

We use a hybrid approach:

- **Symbolic Execution**: Pure symbolic analysis without actual values.
- **Concolic Execution (Concrete + Symbolic)**: Executes the program concretely while tracking symbolic expressions.

Concolic testing helps overcome path explosion and supports analysis in environments with partial observability or external dependencies.

5.4 Tools Explored

We evaluated several tools for symbolic execution of Java programs:

- **KLEE**: A symbolic executor for C/C++ (via LLVM); discarded due to limited native Java support.
- **JCUTE**, **JDART**, **JBSE**: Java-based tools with varying support for symbolic execution and JML.
- **Java PathFinder (JPF)**: A model checker and execution engine for Java bytecode.
- **Symbolic PathFinder (SPF)**: An extension of JPF with support for symbolic and concolic execution.

5.5 Why JPF/SPF?

We selected **SPF** as the primary execution engine due to:

- **Native Java support**: Works directly on Java bytecode.
- **JML contract integration**: Supports `@requires`, `@ensures`, and `@assert` annotations.

- **Symbolic + Concrete modes**: Enables hybrid testing and precise constraint modeling.
- **Extensibility and community support**: Backed by NASA, with active forks supporting Gradle and Maven.

5.6 Configuration and Execution Flow

The symbolic execution setup includes:

- **Test driver Java class**: Assembled from Phase 4.
- **Configuration file (.jpf)**: Specifies symbolic variables, target class, and listeners.
- **SPF runtime invocation**: Executes the Java bytecode, explores all feasible paths, and logs path conditions.

Sample .jpf configuration:

```
target=BankTest
classpath=./bin
symbolic.method=BankAccount.deposit(sym#int)
listener=gov.nasa.jpf.listener.SymbolicListener
```

5.7 Output of Symbolic Execution

Symbolic execution yields:

- **Path summaries**: Conditions under which each path is taken.
- **Concrete test inputs**: Generated using SMT solvers (e.g., Z3) for each path.
- **Error traces**: For contract violations, null dereferencing, or unhandled exceptions.

5.8 Challenges and Limitations

Symbolic execution is powerful but has known challenges:

- **Path explosion**: Exponential growth in number of paths for complex code.
- **External dependencies**: Cannot handle native methods, file I/O, or platform-specific calls without modeling.
- **Limited pointer/reference support**: Handling mutable state and aliasing remains a challenge.

5.9 Summary

This phase automates deep semantic analysis of Java code guided by formal contracts. By integrating with JPF/SPF, we achieve:

- Automated and test input generation.
- Detection of edge-case and contract violations.
- Validation of platform-dependent behaviors under symbolic constraints.

The results of this phase are used for final validation on real server-side environments in Phase 6.

2.3.6 Phase 6: Server-Side Testing and Future Work

After symbolic execution identifies viable input paths and constraints, the final phase focuses on **executing these test inputs in a real or simulated server-side environment**. This is essential to validate the behavior of the system under realistic deployment conditions, especially for platform-specific software where runtime behavior can depend on system configurations or hardware contexts.

6.1 Server-Side Testing Objectives

While symbolic execution ensures high code coverage and contract validation, it operates on an abstract model of the program. To bridge the gap between formal analysis and practical deployment, this phase aims to:

- Verify that test inputs generated symbolically execute correctly on the target system.
- Validate state changes, side effects, and outputs against JML specifications.
- Test integration with system-level APIs, I/O operations, and platform dependencies.

This step ensures that the theoretical guarantees from symbolic analysis hold under actual execution conditions.

6.3 Output Validation and Assertion Checking

During execution:

- Input-output behavior is monitored.
- Internal assertions (e.g., those from JML `@ensures` clauses) are checked.
- Logs are captured for any deviations or failures.

Test results are compared against expected outcomes derived from formal specifications. Any violation—such as invariant breakage or incorrect state transition—is flagged and logged.

6.4 Mocking System Dependencies

In many cases, symbolic and server-side testing cannot fully capture system interactions (e.g., file reads, network I/O, device state). To address this:

- Mocking frameworks (e.g., Mockito, JMock) are proposed to simulate external dependencies.
- Dummy inputs or simulated I/O streams are used to mimic platform-specific interactions.
- In future work, these mocks will be automatically generated based on symbolic path summaries.

This abstraction allows testing in isolation, reducing dependency on actual hardware or external services.

6.5 Limitations and Real-World Constraints

Despite automation and formalization, server-side testing introduces new challenges:

- Execution environments can vary, leading to inconsistent behavior across platforms.
- Native methods and hardware-specific APIs are not symbolically analyzable.
- Timing-related behaviors (e.g., concurrency, latency) are hard to model and test symbolically.

These constraints highlight the need for a hybrid strategy—symbolic execution for logic validation and server testing for environmental verification.

6.6 Future Work

To extend the current framework's robustness and automation:

- **Mock Integration**: Automate generation of mocks for file I/O, network calls, and platform-specific modules.
- **Support for Advanced Java Grammar**: Currently its limited to a basic grammar without any pointers or memory location so we can improve in that sector.
- **Cross-platform Execution**: Expand testing to Android emulators, embedded JVMs, and smart devices.

6.7 Summary

This final phase validates the practical execution of symbolically generated tests in a real or simulated environment. By doing so, we close the loop from formal specification to actual platform behavior. The combination of symbolic reasoning and server-side verification ensures comprehensive and reliable test coverage, forming a robust foundation for platform-specific software quality assurance.

Chapter 3

Team's Progress

Phase	Description
Initial Research	Explored symbolic execution, concolic testing, Design by contract principles, JML annotations and AST manipulation.
Tool Trials	Tested tools like Java Path Finder(JPF), Symbolic Path Finder(SPF), JCute, JDart, KLEE; finalized JPF + SPF.
AST Exploration	Compared Spoon, JavaParser,JMLParser and Soot; finalized JMLParser.
JML Integration	Added `assume`/`assert` annotations in a sample Java library (Library management system).
Test script	Manually created mock script for our algorithm with Test strings for our Java library (Library management system).
Test input gen	Used SPF on test cases from the test script to analyze symbolic paths and inputs.
Phase 1	Found JML-parser and understood concepts of JML specifications and Design by contract.
Phase 2	Explored AST and grammar generated by JML-parser and say how traversal uses visitor design pattern.
Phase 3	Created a manual test script for a Java library with limited grammar for understanding.
Phase 4	Manually converted the script to be suitable for Symbolic execution also changed other parts of code for the same.
Phase 5	Used SPF as Symbolic executor and tested the manually converted script and checked verification of the script via results.

3.1 Additional Work and More Information

3.1.1 In-depth JPF Exploration & Looping Behavior

Investigated how JPF handles program execution under loop conditions and symbolic constraints. By studying gov.nasa.jpf libraries and experimenting with test cases, we discovered how infinite loops are processed in JPF and how symbolic conditions are evaluated. The research offered valuable insight into limitations during symbolic execution and methods to mitigate unbounded paths.

3.1.2 Find Java Frontend Library with Program Analysis Framework + Generate Code, Analyze AST, Rename Variables, etc

Evaluated multiple Java program analysis frameworks to support frontend code analysis. Initially tried the Soot framework for AST and control flow generation (including Code Property Graphs), but JavaParser offered better integration for our needs. Used it to reconstruct Java code from AST, count classes/variables, and perform basic transformations like variable renaming—paving the way for custom code mutation and refactoring capabilities during test generation.

3.1.3 Write JML Specs, Identify Test Strings, and Provide Manual Test Inputs for Java Library

Wrote JML specifications (using assume, assert, and ensures) for key functions in the Library Management System, our selected test Java library. Identified relevant test strings and created corresponding manual test inputs to simulate the expected symbolic behavior. This helped in aligning the actual output behavior with the symbolic tester's constraints and formed the groundwork for further automation of test generation.

3.1.4 Pseudo Output of the Algorithm with JML-Annotated Test Functions and Corrected Test Inputs

Designed and implemented pseudo test functions for symbolic execution using manually declared variables and corresponding JML annotations (**assume**, **assert**, **ensures**). Each test string was constructed to simulate a distinct execution path and constraint setup. Multiple test string variations were explored, refining the input structure and better aligning them with symbolic test coverage goals using **jpf-symbc**. This exercise served as a prototype for validating our end-to-end test generation pipeline.

3.1.5 JML Grammar and AST via Parser Script

Investigated the availability of JML grammar and AST generation tools. Identified and analyzed a **jml-parser** that includes both the formal grammar and AST structure, which will serve as the basis for our algorithm's AST traversal logic. This avoids the need to write a new grammar from scratch and ensures compatibility with standard JML annotations and contracts.

3.1.6 Test the Manual Script and Explore GUI Testing Tools

Executed the manually created test generation script with SPF and obtained successful symbolic test inputs. Parallelly, explored tools used in the industry for GUI testing of Java/Kotlin desktop and mobile applications. Identified tools like Appium, UI Automator, WinAppDriver, BrowserStack, JPF-Android, and more. These tools may help extend the testing pipeline to interface-level testing. Also began reviewing how JPF handles output formatting to improve readability and integrate future visual test reports.

Chapter 4

Work Distribution

The contributions done by the members are as follows:

Areen Vaghasiya:

- Researched symbolic testing tools: explored Java PathFinder (JPF), Symbolic PathFinder (SPF) in depth, AST traversals,contributed to the integration of JML specifications into the test generation pipeline,manually wrote test strings for testing, generated SPF runnable code for testing Java libs with appropriate JML assert statements.

Aryan Vaghasiya:

- Made test input mapping with JML, helped in JML integration. Implemented AST generation for Java src files using **JavaParser lib**, with focus on parsing & analyzing files containing JML. Configured parser to enable JML processing with help of jmlparser lib and integrated dynamic testing using JUnit. Ensured parser correctly stores tokens and JML comments for further analysis.

Siddharth Palod:

- Researched symbolic testing tools(JCute, JDart, KLEE), arranged the Tool setup (JPF, SPF), Manual Test script writing,Found Java library for testing (LMS),Found AST in form of JML parser along with its CFG (Context-free grammar) for analysis ,architectural diagrams, documentation explaining the workflow and course of action.

Chapter 5

Conclusion

This project extended programming language tooling to enable automated test generation by integrating symbolic execution with JML annotations and AST analysis. We developed a test generation pipeline, including an AST explorer and translation of test scripts into Symbolic PathFinder-compatible formats.

The results show that combining static annotation analysis with symbolic execution can effectively automate test creation.

Chapter 6

References

The citations used in the report have been listed here.

- Păsăreanu, Corina S., and Neha Rungta. "Symbolic PathFinder: symbolic execution of Java bytecode." Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering. 2010.
- A Survey of Symbolic Execution Techniques
- Aho, Alfred V., Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. Compilers: Principles, Techniques, and Tools. 2nd ed., Pearson, 2006. (Chapter 2: Syntax Analysis – Abstract Syntax Trees)
- Leavens, Gary T., and Cheon, Yoonsik. "Design by Contract with JML." University of Central Florida, Technical Report, 2006.
- Leavens, Gary T., et al. "JML: A Notation for Detailed Design." Behavioral Specifications of Businesses and Systems, Springer, 1999.
- Qu, Xiao, and Brian Robinson. "A case study of concolic testing tools and their limitations." 2011 international symposium on empirical software engineering and measurement. IEEE, 2011.
- DART and CUTE: Concolic Testing