Team 7

Programming Language Support for Platform Testing

- Areen Vaghasiya IMT2022048
- Aryan Vaghasiya IMT2022046
- Siddharth Palod IMT2022002

Problem Description

Testing platform-specific software like image processing libraries or gaming systems requires automated and efficient test case generation using rigorous testing methodologies to ensure reliability, especially in critical systems. This project focuses on developing a Java test code generator, addressing language-specific challenges like processing JML (Java Modeling Language) in test automation. Future extensions aim to incorporate advanced algorithms to enhance testing effectiveness and coverage.

Real-world Importance

Modern platforms require rigorous testing for reliability.

Symbolic/concolic tools struggle with modern language features.

Poor support for annotations like JML reduces testing effectiveness.

Leads to inefficiencies in test generation and code coverage.

What is Platform Testing?

A **software platform** is the underlying environment (like an OS or framework) that supports and runs software applications.

Platform testing is the process of verifying that software works correctly across specific hardware or software platform-specific environments

Key goals include:

- Validating cross-platform compatibility.
- Ensuring robust performance under platform constraints.
- Automating tests for complex, platform-dependent features.
- Detecting issues tied to hardware-specific behavior or system configurations.

Challenges Addressed

Manual Test Case Effort
High time and error risk in traditional test creation.

Limited Annotation Support
Tools lack support for Java Modeling Language (JML).

Language-Agnostic Tools Fall Short
Generic tools miss Java-specific features like exceptions and OOP.

Complex Platform Architectures
Systems like smart TVs and gaming consoles demand scalable test generation.

Low Input Coverage & Traversal
Symbolic tools often fail to cover all paths without AST analysis.

Spec Language Gaps in Other Languages
C++ lacks a JML-equivalent, justifying Java focus.

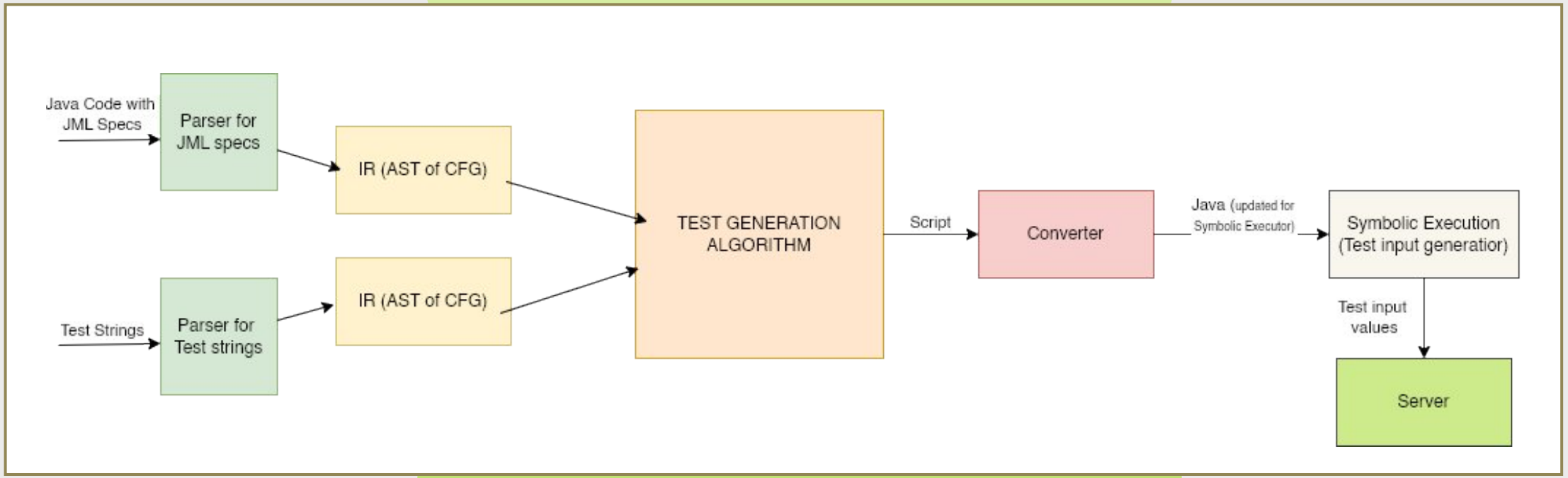
Our Solution

Utilize an automated testing framework

Our Deliverables

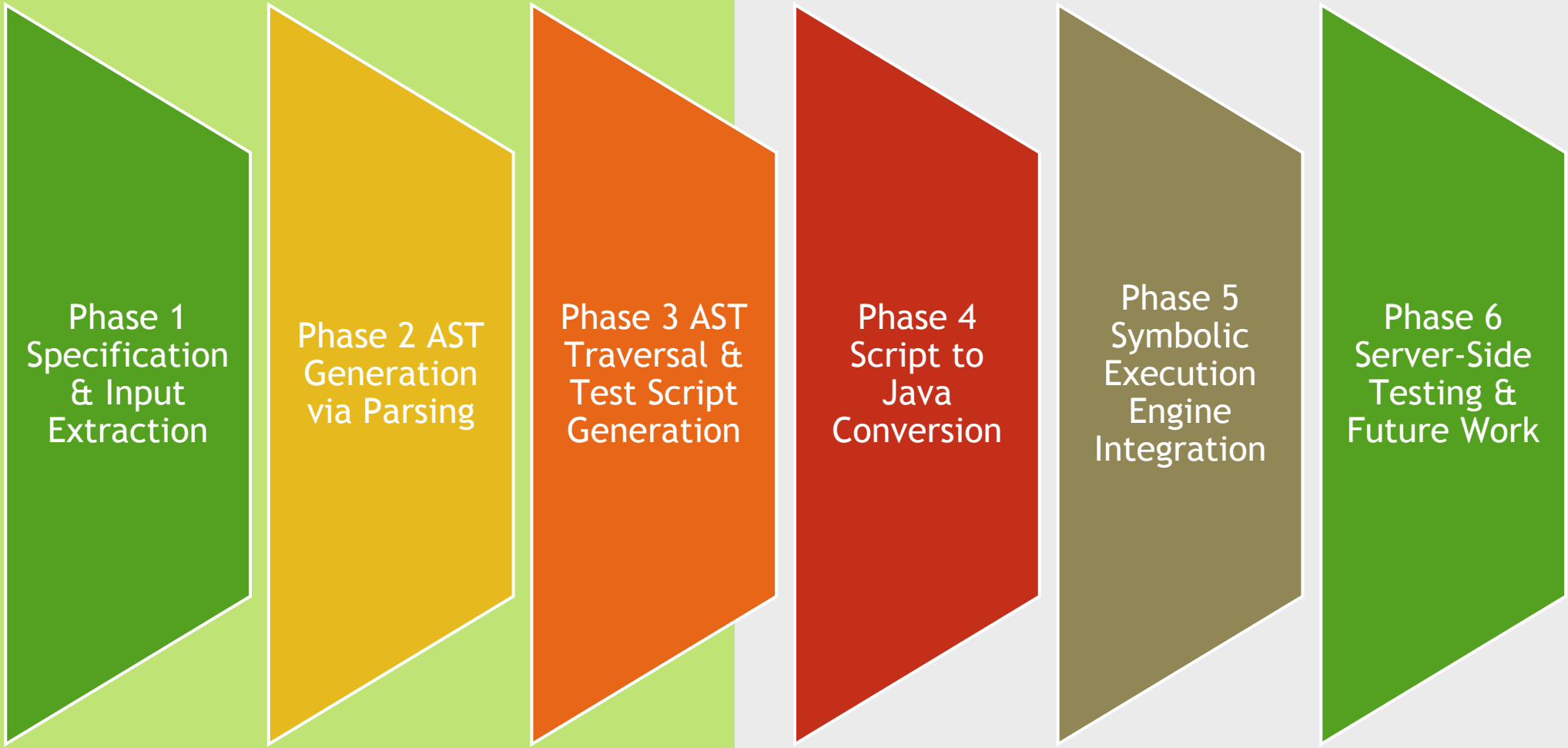
- **Comprehensive Report**
  - Documentation of algorithm designed, methods utilized,
  - Research papers analysed
  - Tools explored
  - Methodology of the workflow and impacts.
- **Annotated Codebase**
  - Manual test script for a Java platform application with integration of symbolic execution.
- **Test Generation Scripts**
  - Write Manual test scripts draft concept to write algorithm leveraging AST + JML for creation of test-generation algorithm which will be send for symbolic execution.
- **Architecture Models**
  - Diagram representing whole algorithm with each section segregated as a independent unit to develop
  - Diagram for example usage and working of algorithms
  - Diagram for indicating AST traversals methodology.

Testing Framework



Don’t worry we will explore it in great detail

Let's Break the Framework into Parts



Phase 1 Specification & Input Extraction

- Goal: Identify JML specifications and Test Input Strings
- Based on Design by Contract (DbC)
- Extract constraints and input expectations for the platform app

Why we need Specifications ?

1. In black-box testing, we do not access the internal logic of functions — we rely solely on what is specified.
2. Specifications (like JML annotations) define expected behaviour using preconditions, postconditions, and invariants that help generate valid test inputs and expected outputs.
3. Without specifications, test generation tools cannot determine correct or incorrect behaviour.

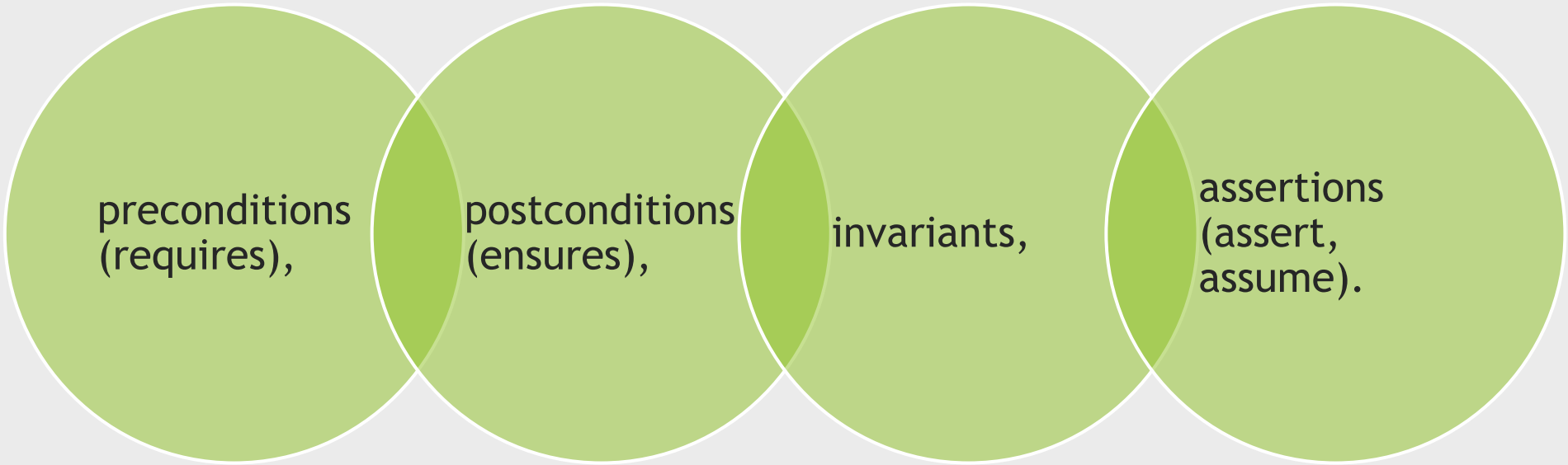
Why Is This Important for Server-Side Testing?

1. On the server side, we often only interact through APIs or exposed methods — we can't see the implementation.
2. Formal specifications act as contracts between the client (test generator) and the server logic.
3. Test generation tools use these contracts to derive input constraints and expected outcomes.

What is JML Annotations ?

JML (Java Modeling Language) is a behavioural interface specification language for Java programs.

It allows developers to annotate Java classes and methods with formal specifications such as:



JML specifications help developers understand expected behavior and automate the testing of edge cases.

Example:

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

Leavens, Gary T., et al. “JML: A Notation for Detailed Design.” Behavioral Specifications of Businesses and Systems, Springer, 1999.

What is DbC(Design by contract)?

Design by Contract (DbC) is a software correctness methodology where components act as entities that communicate via well-defined contracts, just like legal agreements.

The contracts formally define:

- *true before calling method*
- *true after method finishes*
- *always true for the class or object*

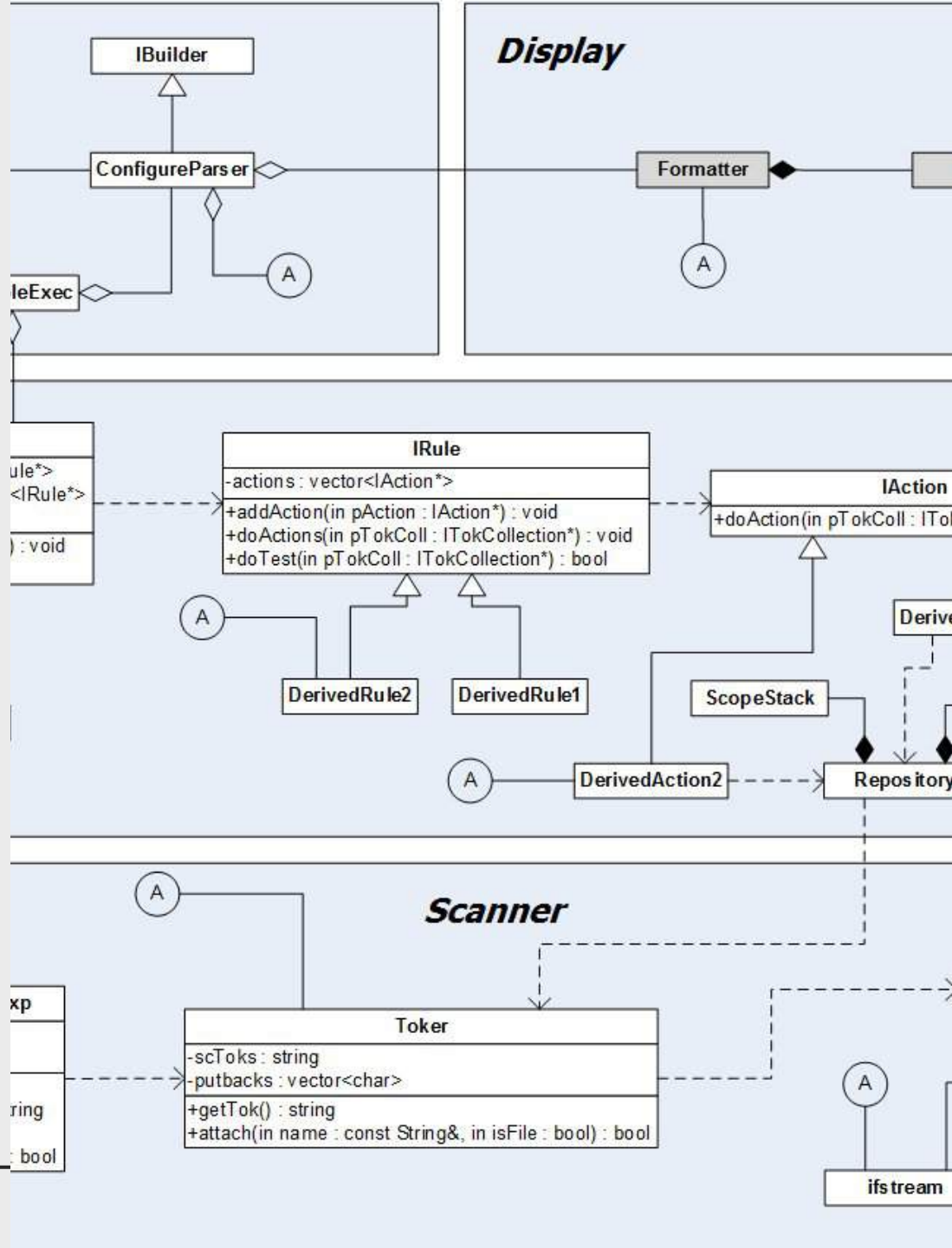
Why Use DbC?

1. Encourages correct-by-construction programming
2. Helps detect bugs earlier through formal conditions
3. Improves readability and maintainability of code
4. Enables formal reasoning and test generation

Leavens, Gary T., and Cheon, Yoonsik. “Design by Contract with JML.” University of Central Florida, Technical Report, 2006.

Phase 2 – AST Generation via Parsing

- Goal: Identify JML specifications and Test Input Strings
- Based on Design by Contract (DbC)
- Extract constraints and input expectations for the platform app

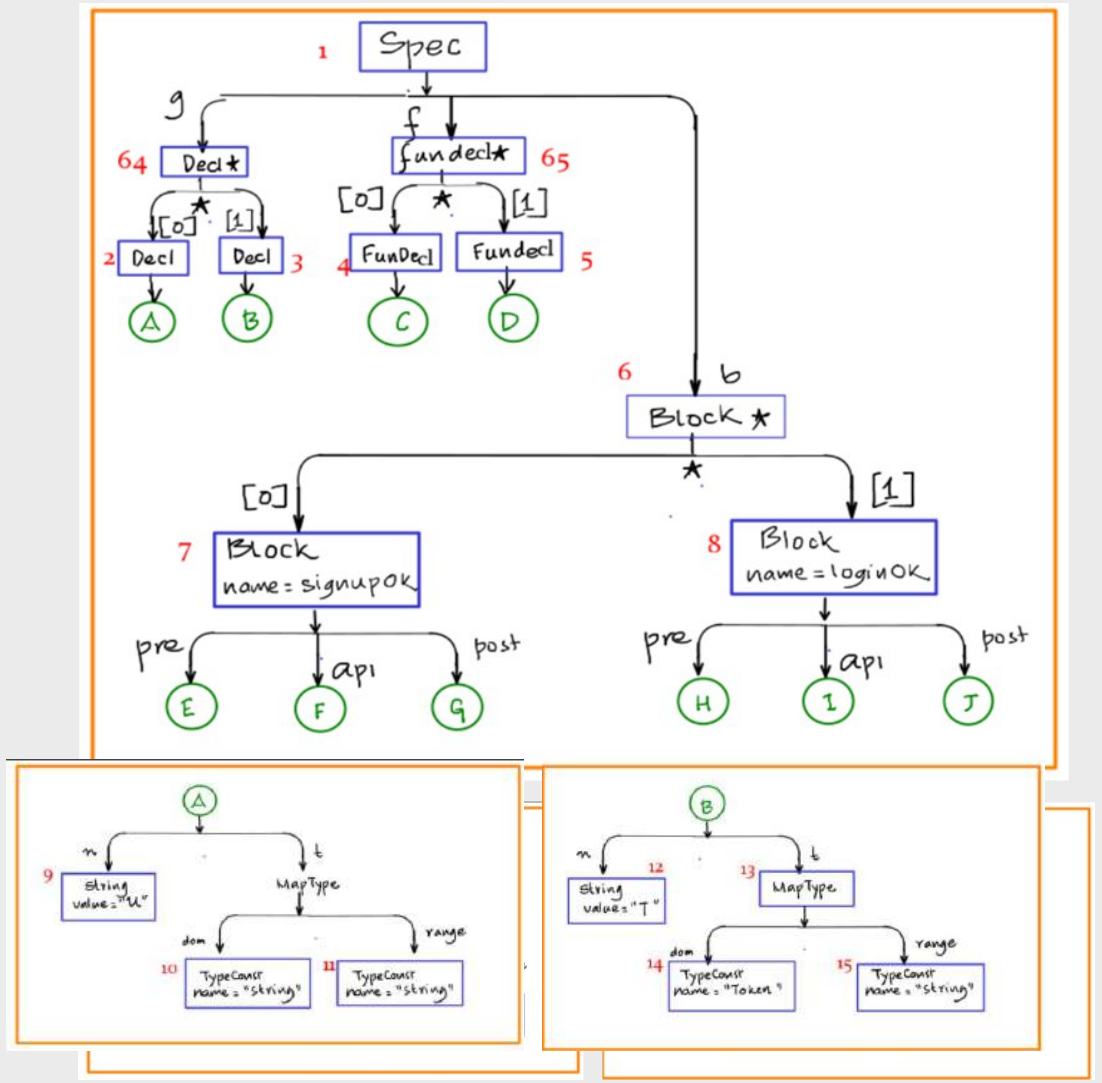


What is IR and Why Use AST ?

- IR is abstraction of source code used during program analysis and transformation.
- Simplifies complex programming constructs into form that tools can easily access.

Abstract Syntax Tree (AST)

- Tree-structured **Intermediate Representation** that captures the **syntactic structure** of code based on its grammar.
- Crucial for understanding **program flow**, **dependencies**, and **logical segments** like methods, variables, loops, and annotations.



Example AST from **API Testing grammar in languages.pdf**

Abstract Syntax Tree (AST)

AST enables:

- **Traversal and transformation** of code using visitor patterns.
- Easy **extraction of JML annotations** for contract-based testing.
- Generation of scripts or test cases by identifying functional blocks in code.

In the Project,

- We use AST to represent Java + JML code in a intermediate representation.
- AST acts as a bridge between raw code and automated test generation logic.
- Enables systematic code analysis and **symbolic path extraction** through **visitor design patterns**.

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)

Why use Parser in making AST?

A parser is a tool that helps construct the Abstract Syntax Tree (AST), which structurally represents source code for automated analysis and transformation.

Why use Parser for AST generation?

Converts code into AST that reflects program's syntax

ASTs are essential for tasks like test generation, code optimization, refactoring, and symbolic execution.

Tools like JavaParser, Spoon, and JMLParser help extract function definitions, variables, etc for further analysis.

Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). Compilers: Principles, Techniques, and Tools (2nd ed.).

Tools explored for Parser and AST generation

Tool	Language Level	AST type	JML support	Code Transformation	CFG/call graph support	Ease of Use	Use Case
JavaParser	Source (Java)	Syntactic AST	✗	✓ (Simple edits)	✗	✓ High	Fast parsing, test string generation
Spoon	Source (Java)	Semantic AST (CtModel)	✗	✓ (Advanced edits)	✗	⚠ Moderate	Mutation, instrumentation, AST traversal
JMLParser	Source (Java+JML)	Annotated AST with JML nodes	✓	Limited	✗	⚠ Moderate	Handling JML annotations in AST
Soot	Bytecode	IR	✗	✓ (IR-level)	✓ (CFG, call graphs)	✗ Low	Deep analysis, symbolic engine backends

Some more tools: ANTLR,Eclipse AST

Phase 3 – AST Traversal & Script Generation

- Goal: Identify JML specifications and Test Input Strings
- Based on Design by Contract (DbC)
- Extract constraints and input expectations for the platform app

Visitor Design Pattern

Why Visitor Pattern?

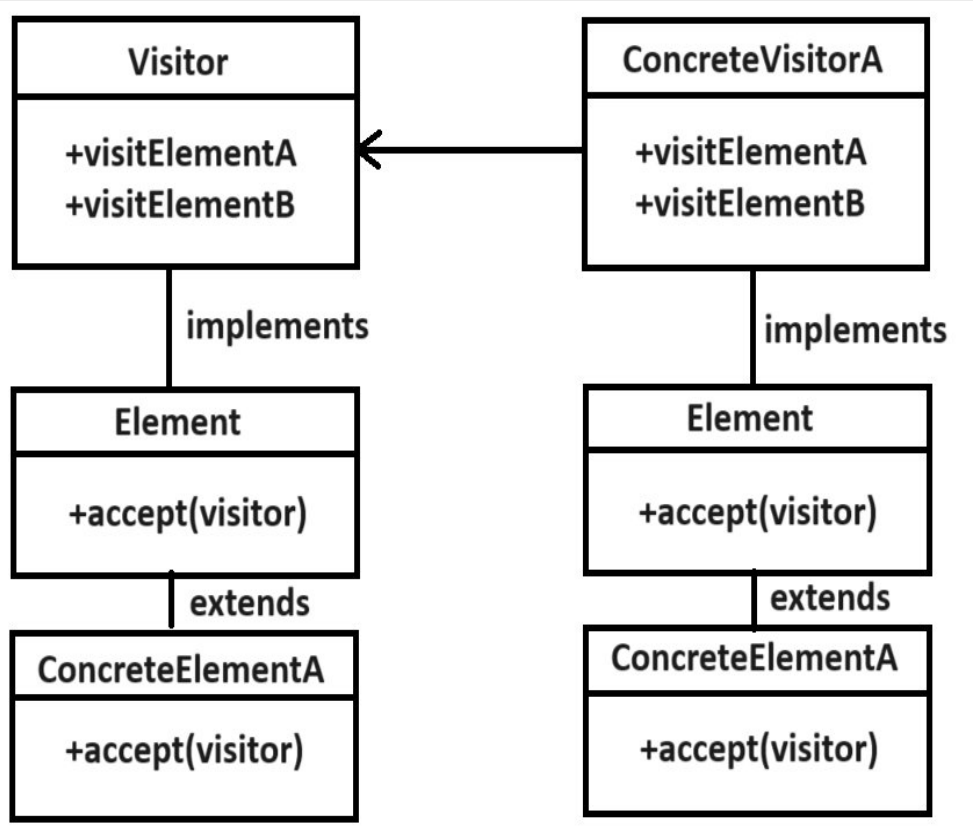
1. It provides a clean separation between data structure (e.g., AST nodes) and operations (e.g., test generation, traversal logic).
2. Makes it easy to add new operations without modifying AST node classes.
3. Ideal for symbolic testing where different logic (e.g., constraint generation, code instrumentation) is applied to nodes.

How It Works:

1. Each AST node (Element) implements an accept(visitor) method.
2. The Visitor defines visit methods for different node types (e.g., visitFunctionDecl, visitVariable).
3. Concrete visitors implement specific logic while traversing the AST.

Usage in Our Project:

1. Used Visitor to walk through Java ASTs generated by JMLParser.
2. At each node, generated corresponding test strings or symbolic constraints.
3. Helps modularize code and scale easily for additional node types or test features.

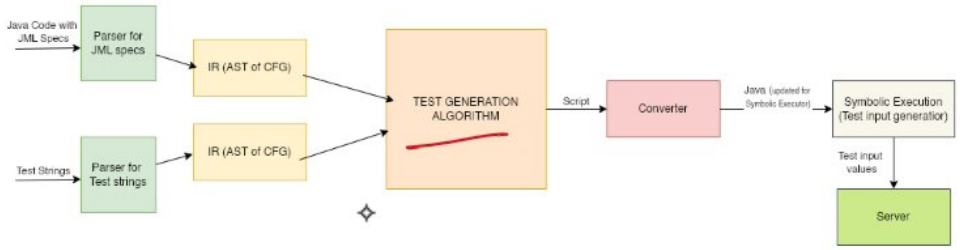


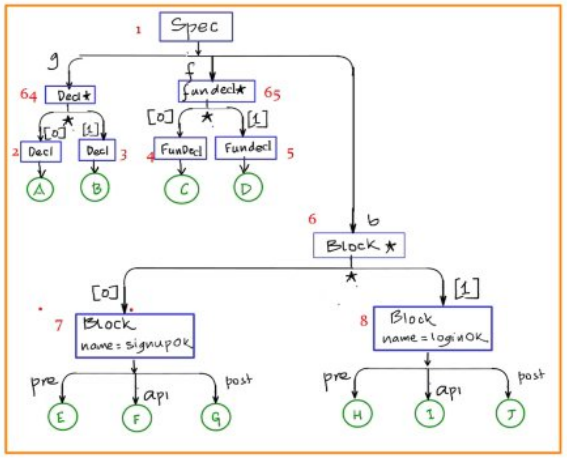
https://www.geeksforgeeks.org/visitor-pattern-javascript-design-patterns/

Current Test Generation Script

Sample Test Gen Algorithm

30 April 2025 22:21





Issues in Script for Future Work

- **Current Limitations & Opportunities for Improvement:**

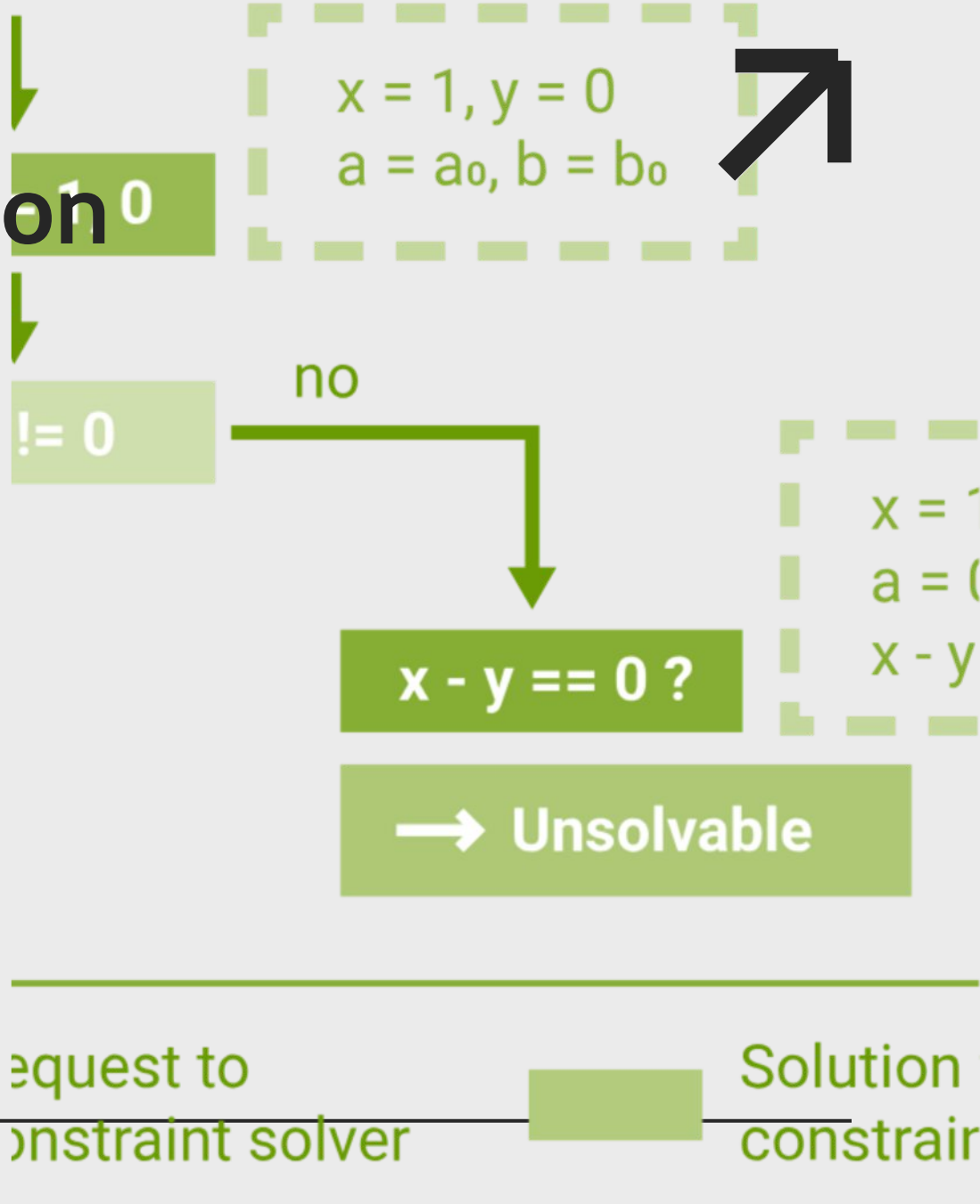
1. Does not handle pass-by-reference or pointer-based data (common in native or low-level logic).
2. Not robust for higher-order functions (HOFs) or functions passed as arguments.
3. Replace direct test function calls with mocking frameworks (e.g., Mockito, EasyMock) to simulate behavior.

Phase 4 - Script to Java Conversion

- Convert the script into imputable Java code for Symbolic tester
- Making all functions symbolic (Instead of reading a actual file simulate mock file reading)
- Create a specification file for symbolic executor settings (like if using Java Path finder create a .jpf file)

Phase 5 – Symbolic Execution Engine Integration

- Goal: Identify JML specifications and Test Input Strings
- Based on Design by Contract (DbC)
- Extract constraints and input expectations for the platform app



What is Symbolic Execution

- Symbolic Execution runs a program using **symbols instead of concrete values** for inputs.
- Inputs are treated as symbolic variables (eg- x, y)
- Program paths are explored conditionally:

  if (x > 5) ⟶ Execution forks:
  - **Path 1:** $x > 5$
  - **Path 2:** $x \leq 5$

- A path condition is built for each path.
- SMT solvers are used to generate actual inputs satisfying these conditions.

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.

What is Symbolic and Concolic Testing

Concolic =Concrete + Symbolic

A **hybrid testing technique** that combines:

- Concrete Execution (running with actual inputs)
- Symbolic Execution (tracking expressions symbolically)

Essential for **black-box testing** where expected behavior is known via specifications (JML)

WHY?

- **Manual test writing is inefficient** and error-prone, especially in large-scale Java applications.

Symbolic execution enables:

- Automated Test generation
- High code and path coverage
- Detection of edge cases that manual testing may miss

Tools Explored for Symbolic Execution

u Klee (via Docker image)- Used for C/C++ has Java support by translating but as we were using JML Specs of Java dropped for Java based tool

u Papers: CUTE, DART

u JCUTE,JDART,

u JBSE,

u JPF (Java Path finder)

u SPF (Symbolic Java Path finder)

References

- Cristian Cadar, Daniel Dunbar, and Dawson Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." *OSDI*, 2008.
- Koushik Sen, Darko Marinov, and Gul Agha. "CUTE: A concolic unit testing engine for C." *ESEC/FSE*, 2005.
- Patrice Godefroid, Nils Klarlund, and Koushik Sen. "DART: Directed Automated Random Testing." *PLDI*, 2005.
- Willem Visser, Klaus Havelund, Guillaume Brat, and Seungjoon Park. "Model Checking Programs." *ASE*, 2000.
- Willem Visser, Corina S. Păsăreanu, Sarfraz Khurshid. "Test Input Generation with Java PathFinder." *ISSTA*, 2004.

Why SPF and JPF selected ?

Native Support for Java

- Directly works on Java bytecode – no need for translation.
- Ideal for testing Java applications with formal specs (JML)

Integration with JML-based Contracts

- Supports annotations like @requires, @ensures, @assert
- Aligns with our DbC testing approach.

Symbolic + Concolic Capabilities

- SPF handles both symbolic path reasoning and concrete execution modes.

Rich Ecosystem

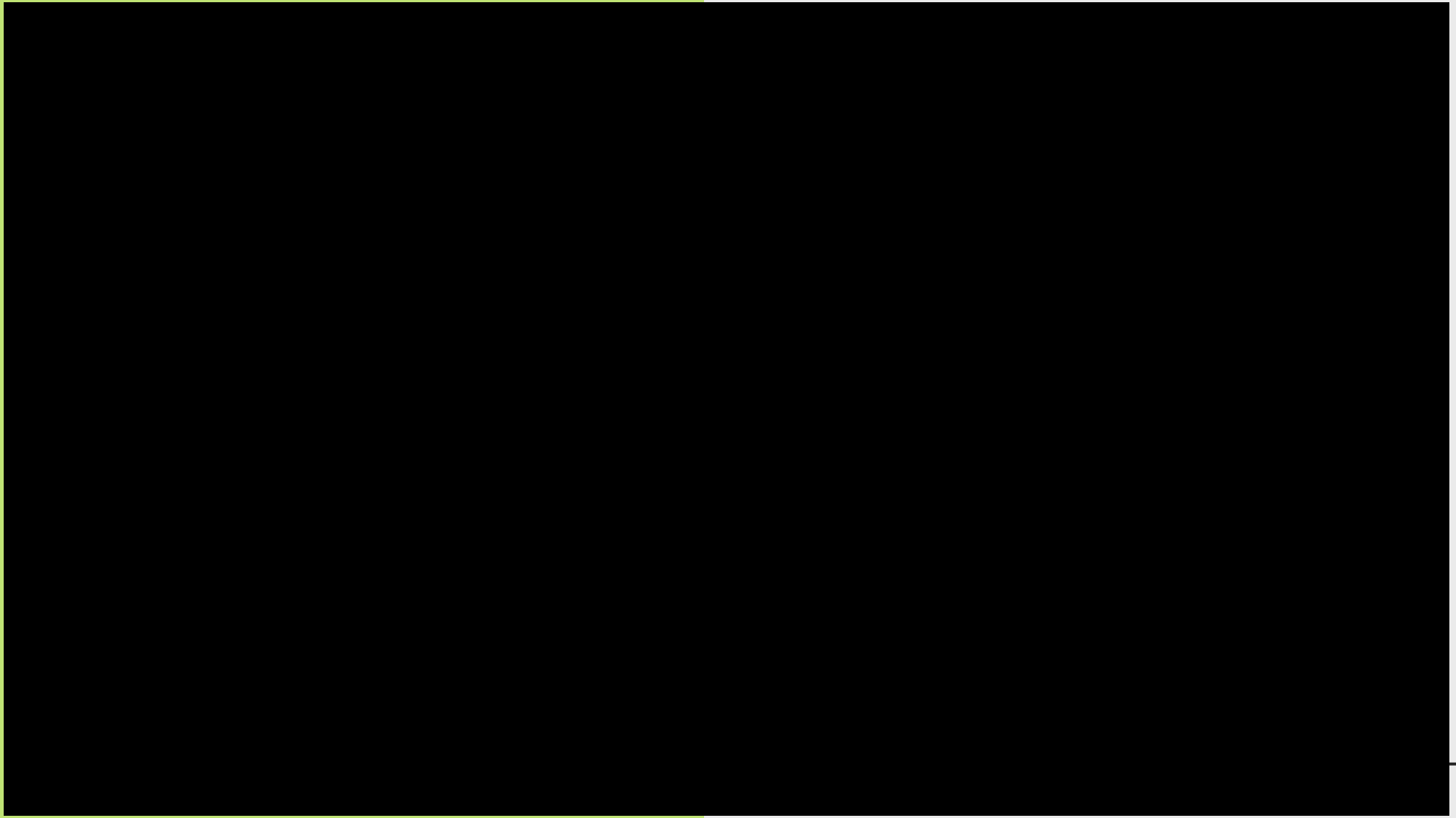
- Multi- mode configurable architecture
- Backed by NASA and actively maintained forks with Gradle and Java versions support

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.

Explore Codebase

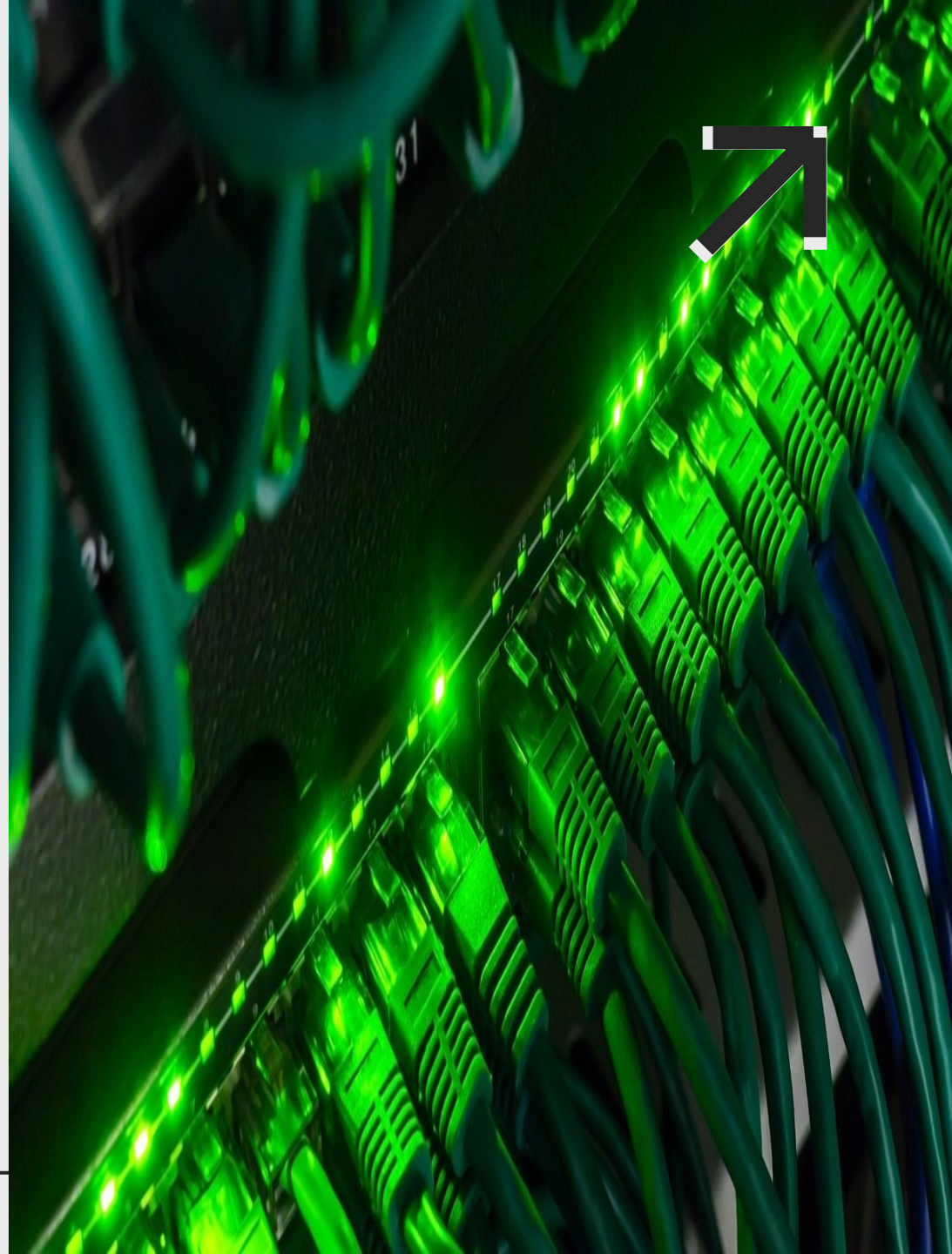
We created a manual test script for a Library application written in Java.

We created JPF code for the manual test script and also changed the dependancies to work on symbolic execution i.e. Remove actual IO operation into Symbolic Yes/No agreements



Phase 6 – Server-Side Testing & Future Work

- Now that we have test input values we should test it on our actual server for verification
- In Test generation as you have seen for most of the code we don’t really need server except this case so can we reduce our dependency here too; For this we introduce mocking in server-side which can be touched upon in future
- Extract constraints and input expectations for the platform app




Thank you