Google Summer of Code

GSoC 2025 Project Proposal

The JPF Team

JPF is an Abstract Virtual Machine (VM) written in Java. It does not execute a program like a normal JVM, it systematically explores all possible execution paths to check for errors, deadlocks, and unhandled exceptions. JPF was started as a model checker around 1999 and was developed at the **NASA Ames Research Center.**

JPF also has extensions like Symbolic Pathfinder (SPF), which combines symbolic execution with model checking and constraint solving for automated test case generation and error detection in Java bytecode. In SPF, programs are executed on symbolic, rather than concrete inputs.

Github Repositories

- *jpf-core* - The *jpf-core* project constitutes the core structure of JPF.
- *jpf-symbc* - The *jpf-symbc* project constitutes SPF, a JPF extension.

Table of Contents

Support Runtime Exception In SPF

Abstract

The project aims to support and extend SPF’s symbolic execution capabilities which are being limited by certain factors, unhandled common Java runtime exceptions arising from symbolic inputs and the need to polish or fix the issues related to Java 11. This project also aims to address certain issues on SPF like additional support for strings and the impact of Java 11 on previously supported model classes in Java 8 (e.g., **Math.java, Scanner.java**). This project aims to help SPF detect, model and explore execution paths leading to runtime exceptions such as ***NullPointerException*** and **IndexOutOfBoundsException.** This will involve modifying the core components like symbolic bytecode handlers and methods summarizers (e.g., ***SymbolicStringHandler***). This work, primarily, will extend the *sv-comp* branch resulting in a better version of Symbolic PathFinder.

Introduction

This section provides a brief introduction to the necessary tools in this project.

Symbolic Execution

In particular, Symbolic Execution is a well-known program analysis technique that was proposed in the 1970's, where we execute programs on symbolic rather than concrete inputs and the idea is that these symbolic inputs represent multiple concrete inputs. The analysis maintains ***path conditions*** that encodes conditions on inputs that follow the same program paths and these path conditions are basically collected from different conditions in the code and are checked for satisfiability using ***off-the-shelf solvers*** (for e.g., Z3). The paths generated during the symbolic execution of a program are characterized by a ***symbolic execution tree***. Now, if the path condition is found to be non-satisfiable the analysis stops from that path because there are no inputs that can follow that path. So, effectively the analysis only explores feasible paths and the byproduct of this constraint solving is that the obtained solutions can be used as test inputs that exercise different paths through the code.

Symbolic Pathfinder

SPF is a JPF extension that performs symbolic execution of Java bytecode. SPF supports a mixed mode execution that combines concrete and symbolic execution. SPF is a part of the Java Pathfinder verification tool-set and as we know JPF includes ***jpf-core***, a model checker which consists of an extensible custom Java Virtual Machine (VM), state storage and backtracking capabilities, as well as listeners. In contrast to JPF, SPF replaces the concrete execution semantics of jpf-core with a non-standard symbolic interpretation using an ***instruction factory***. SPF relies on the jpf-core framework to systematically explore the different symbolic execution paths, it uses JPF also to generate and execute the ***symbolic execution tree*** of the code under analysis. SPF uses a variety of ***off-the-shelf*** decision procedures and constraint solvers (Z3, Choco, CVC3, IASolver) to solve the constraints generated by the symbolic execution of the bytecode programs.

Choice generators (PCChoiceGenerator) are used to implement non-deterministic choices when symbolically executing ***branching conditions***, and ***listeners*** are used for printing the results of the symbolic analysis.

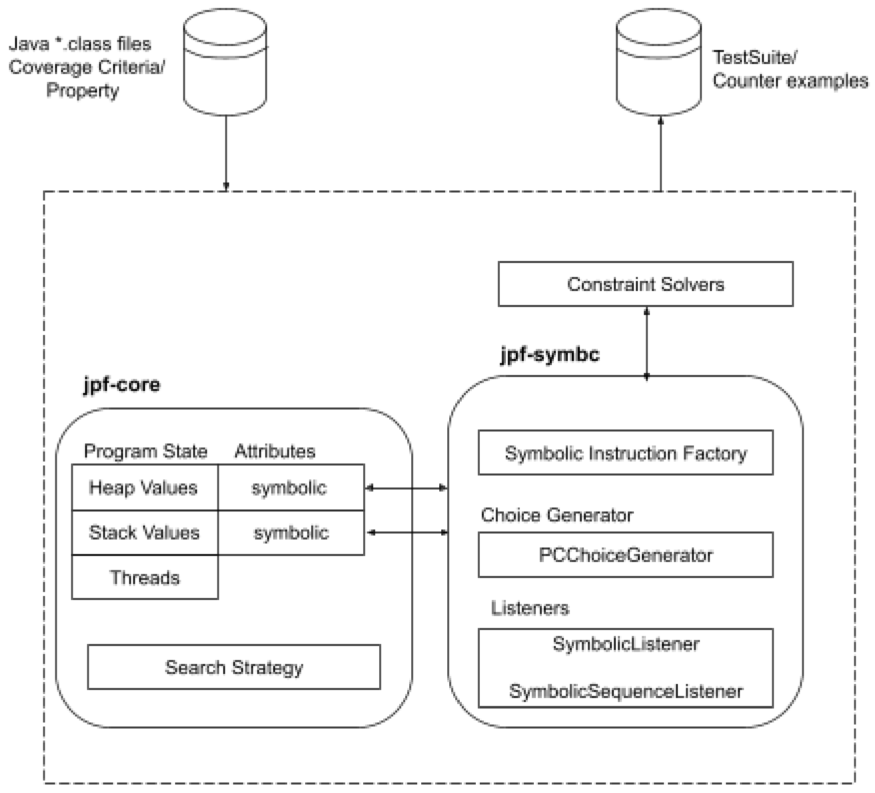


Figure 1: Architecture of SPF

Just like JPF, SPF also requires as input a .class file, a configuration file that would tell about which methods should be executed symbolically in a program, and properties to specify.

After execution is finished, it outputs possible errors and test cases for each error for example ***Null Pointer Exception*** and ***Out of Bounds,*** SPF will describe two cases one for each of them. SPF also uses jpf-core's native peers mechanism to model native libraries and any other program parts that cannot be analyzed directly with symbolic execution.

Project Description

The main goal of this project is to support throwing a runtime exception for some of the summarized functions such as ***String.substring.*** Also this project will build upon SPF's Java 11 gradle support, addressing and solving existing issues. This project will be extending the SPF branch *sv-comp*.

Difficulty: *Medium*
Scope: *350 hours*
Required Skills: *Knowledge of Symbolic Pathfinder*
Possible Mentor: *Soha Hussein*

What is sv-comp ?

It stands for ***Software Verification Competition*** which happens to be annually, that evaluates the effectiveness of software verification tools. It provides a set of verification tasks to compare different tools for security, correctness, safety, and etc.

SV-BENCHMARKS are the collection of these tasks including programs written in languages like C and JAVA. These benchmarks help in accessing the performance of verification algorithms.

Gitlab Repository

- sv-benchmarks

Symbolic String Execution

Symbolic Execution in SPF is built upon *jpf-core*, which acts as a replacement for the standard JVM, jpf-core implements its own flavour of VM. Just like the standard JVM, jpf-core's VM is also fundamentally a stack-based machine. All Java bytecode instructions (like iload, iadd, invokevirtual, etc.) have corresponding implementations within jpf-core, managed by the ***instruction factory***. This factory is responsible for creating the objects that execute the semantics of each bytecode.

SPF (jpf-symbc) extends this standard ***instruction factory*** with a **SymbolicInstructionFactory.** When JPF executes bytecode under symbolic mode, then handlers provided by this factory are used. JPF's VM operates on a stack, allowing associated attributes with each stack entry. It stores the symbolic variables or expressions that the concrete integer value on the stack might represent during the symbolic execution.

When a symbolic instruction handler from the **SymbolicInstructionFactory,** it pops the required operands from the stack and accesses the attributes of these operands typically by constructing a new node in an Abstract Syntax Tree. It pushes a concrete placeholder value back onto the stack and stores the new symbolic expression in the attributes of that placeholder. The stack manipulation must match the execution signature of the original bytecode instruction.

The same principle is extended to handle string operations. The **SymbolicInstructionFactory** provides handlers for bytecode instructions that involve string manipulation. If a result of an operation is used in a conditional branch instruction, the handler generates a logical constraint. This constraint is added to the Path Condition for the current execution path. Choice Generator mechanism is invoked at these branches to ensure feasible outcomes are explored on separate paths. The generated constraints can be solved using SMT solvers like Z3 using bitvectors/automata approaches. For very complex or native String methods, the native peer mechanism can be used as an alternative.

Symbolic Execution Example

As an illustrative example consider the code below, that computes the absolute difference between two input integers x and y.

```
int x, y;
if (x > y)
    result = x - y;
else
    result = y - x;
assert (result > 0);
```

The concrete execution of the program will follow only one path through the code, based on the values of x and y. Given the values x = 2 and y = 1, the true branch of the if statement is executed, and in turn the assertion is not violated. Symbolic execution starts with symbolic, rather than concrete, input values, x = X , y = Y , and sets the initial value of PC as true. At each branch point, the PC is updated with constraints on the inputs in order to choose between alternative paths. For example after executing line one in the code, both alternatives of the if statement are possible, and the PC is updated accordingly with the two different possibilities.

When the path condition is unsatisfiable it becomes false, which means that the path is now infeasible and symbolic execution does not continue along an infeasible path.

Now symbolic execution will explore three different feasible paths, with the following path conditions:

- $PC_1 : X > Y \wedge X - Y > 0$ for Path 1,
- $PC_2 : X \leq Y \wedge Y - X > 0$ for Path 2,
- $PC_3 : X \leq Y \wedge X - X \leq 0$ for Path 3.

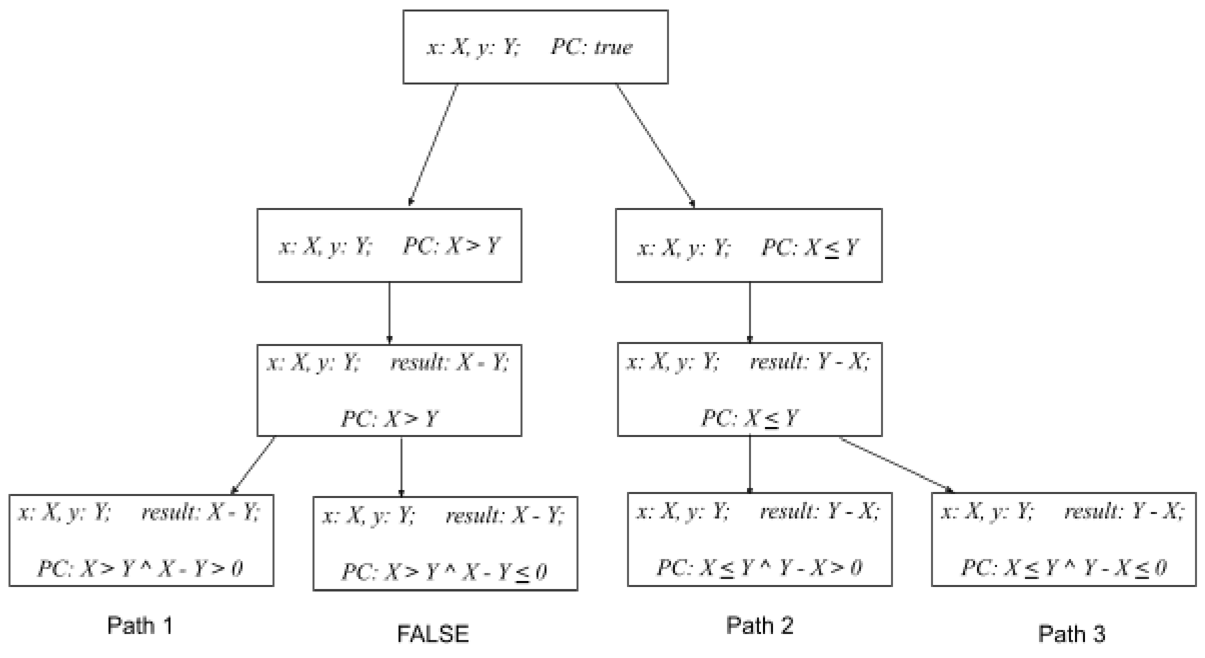

x: X, y: Y; PC: true
x: X, y: Y; PC: X > Y
x: X, y: Y; PC: X ≤ Y
x: X, y: Y; result: X - Y; PC: X > Y
x: X, y: Y; result: Y - X; PC: X ≤ Y
x: X, y: Y; result: X - Y; PC: X > Y ^ X - Y > 0
x: X, y: Y; result: X - Y; PC: X > Y ^ X - Y ≤ 0
x: X, y: Y; result: Y - X; PC: X ≤ Y ^ Y - X > 0
x: X, y: Y; result: Y - X; PC: X ≤ Y ^ Y - X ≤ 0
Path 1
FALSE
Path 2
Path 3

Figure 2: Execution Tree

PC_3 characterizes the concrete program executions that violate the assertion. This tells that the code has an unstated precondition, namely x ≠ y. For test input generation, the obtained path conditions are solved using off-the-shelf decision procedures and the solutions are used as test inputs that are generated to exercise all the paths through this code. These test cases also provide high path coverage.

Symbolic execution explores a program's behaviour using symbols and constraints allowing it to show many potential concrete executions simultaneously to find bugs.

Note that I have taken the reference of this example from the article:

"**Symbolic PathFinder: Integrating symbolic execution with model checking for Java bytecode analysis"** which was released in September 2013.

$$X = Sym_x; \quad Y = Sym_y$$

Why JPF and this project ?

My journey with JPF started when I first reached out to one of the mentors on 16 July, 2024 asking about how to contribute to JPF in the upcoming year (GSoC 2025). But my interest grew in SPF more than that of in JPF. Firstly, I was exploring JPF and how to setup it locally, which was bit hard for me, so I moved to the jpf-symbc repository and with the mentors help, I was able to setup the whole project locally. What is different in SPF is that while working on the *sv-comp*, *gradle build* and *gradle-build-java-11*, it has Gradle Multi-Project Build and Git-Submodules, introduced.

As I mostly code in Java and I always wanted to contribute to an open source organization by the means of GSoC, so I chose JPF. It is a pretty good organization with experienced mentors.

I chose this project "**Support Runtime Exception in SPF**" because it is a medium difficulty project, one should not know more than that JPF is a virtual machine and also it is a model checker along with the concept of ChoiceGenerators. What attracted me the most to this project was the support to throw a runtime exception and the opportunity to work on Java 8 and 11 along with Gradle. Additionally, I will learn about program analysis, model checking and symbolic execution—topics that have fascinated me for a long time. And the work I will do in this project will benefit my career ahead, as the successful contribution to this project will serve as a proof of my strong knowledge in Java.

As a newcomer to JPF/SPF, I chose this project because clearly it aligns with my current skills and interests, making it a great starting point for me to contribute to open-source.

The JPF community has been incredibly supportive, always responsive to my questions. I am especially grateful to *Soha Hussein,* whose help and guidance have made my learning and proposal process much easier. I truly appreciate her support.

Personal Introduction

Personal Details

Name: Saif Ali Khan

Email: saif7862254j@gmail.com | saif.khan16@outlook.com

College: Dayanand Academy of Management Studies, Kanpur

College Affiliation: C.S.J.M University, Kanpur

Program: Bachelor of Computer Applications

Github: saifk16 | f-ei8ht | f-eightProjects

LinkedIn: SAIF ALI KHAN

Twitter: SAIF ALI KHAN

Location: Kanpur (Uttar Pradesh), INDIA

Time Zone: Indian Standard Time (IST) (UTC +05:30)

Phone Number: +91 89249 51075

Preferred Language: English (both spoken and written)

About Me

I'm a final year undergraduate student at Dayanand Academy of Management Studies, currently pursuing Bachelor of Computer Applications. I have been using open-source softwares, even before I knew what they were. Whether it's using, **VLC Media Player** or **OpenJDK,** I always wanted to contribute to open-source projects in any way I can.

My open-source journey started when I learned about **Git** and **GitHub** (from a YouTube video) and the whole workflow of how open-source works i.e., how to write PRs, raise issues, contributions, collaborations, etc. This made me eager to learn about open-source development and that's when I discovered **GSoC.** It is a perfect opportunity for me to contribute to open-source projects while learning from experienced mentors.

Experience

As a first-time GSoC participant, I'm aware of my experiences and open-source involvements around different projects. I worked as a **web development intern** for 2 months, where I used HTML, CSS, JS, and Bootstrap to develop and style web pages for an organization. This was the first time I learned how multiple contributors can work on their own forks, make changes, and create pull requests to merge them into the main GitHub repository. Additionally, I have worked on different projects that have helped me explore and improve my knowledge of writing clean code.

Some of the **projects** are listed below.

1. Smart Contact Manager

Github Repository - Link

- Smart Contact Manager is a web-based contact management system that allows users to securely store and manage their contacts.
- Built using Java, Spring Boot, Hibernate, JPA, Thymeleaf, MySQL, HTML, CSS, JS, Flowbite.
- The project follows an Incremental model with Agile practices.
- Supports Google, GitHub, and Self-Provider for secure login and sign-up using OAuth 2.0.
- Contact Images are stored securely using Cloudinary.
- Users can create, update, delete, and view their contacts, also they can search and filter their saved contacts.
- This is my academic project, you can see my synopsis for this here.

2. mathJX

Github Repository - Link

- mathJX is a mathematical library written in java.
- This will include mathematical functions from fundamental concepts like prime numbers to advanced topics like interpolation and divided differences.
- This is a personal project that I started to improve my knowledge of mathematics while coding in Java.

Apart from these two projects, I had also worked on a project with two of my colleagues, which was a Shopping Management System (using java swing) where we added efficient management of users, categories, products, suppliers, and transactions. Currently I'm working on *mathJX*, but I also have a personal portfolio built using *Hugo*, an open-source static site generator written in **Go**.

These projects helped me gain valuable experience in open-source development, specifically in Java.

I had also participated in two offline coding competitions, both of them were focused on problem-solving. In one of them I secured **4th** place, and they were inter-college competitions as students from around 10 colleges participated.

Open Source Usage/Involvements

I have made a few contributions, which also include a recent pull request (PR) on *jpf-symbc.* But most of my earlier contributions are small and unrelated to JPF, SPF or Java. I would prefer not to mention them in detail as they do not align or fit with my GSoC proposal.

But small contributions can serve as a good starting point for new contributors just like me. My first contribution #4 where I just updated a flappy bird game version by adding a window title with an icon to it.

In my second contribution #5 I added support for my play station one controller to the game but I closed this pull request because key mappings can vary across different controllers.

Thus minor contributions can be a good source of learning for newcomers to open-source. Additionally, I actively use open-source tools, such as Arch, OpenJDK, LibreWolf, Brave, VLC Media Player, Obsidian, Yay and now I must include JPF/SPF as well in the list.

Contribution to SPF

ID	Name	Status
#111	Improve max/min methods to handle NaN and -0.0f/-0.0d correctly	Under Review

In this pull request I made some changes to *Math.max()* and *Math.min()* methods. It handles the following listed for **float** and **double** types:

- NaN: Not a Number (Double.Nan/Float.NaN)
- 0.0f and -0.0f/0.0d and -0.0d

I referred to the 2019 revision of IEEE Standard for Floating-Point Arithmetic (754-2008) and OpenJDK's **Math.java** to ensure correctness and avoid direct plagiarism. Implementation code snippet –

```
if (a != a) { // NaN is not equal to itself
    return a;
}
if (a == b) {
    return (1.0d / a > 0) ? a : b; // Handle 0.0 and -0.
} // Math.max() implementation of double type
```

0.0 == -0.0 will evaluate as true, and as we know that in Java floating-point literals are of double type by default, as well as 0.0f and 0.0d are default values for double and float respectively. So ***1.0d / a > 0*** will result in +Infinity which is greater than 0, hence it will return 0.0, now if we pass arguments like this **-0.0 == 0.0** this will also evaluate as true, so ***1.0d / a > 0*** will now result in -Infinity which is less than 0, hence again it will return 0.0, references were taken from **The JLS Java SE 8 Edition**.

Currently the PR is under review because I need to test my implementation, which will show the benefits of the code I had added. This will help the maintainers to validate my code and increase regression tests. Since my code does not require symbolic execution, the tests will run concretely. What is needed are only a **test** and a **.jpf** file.

Project Proposal

Motivation

The core motivation is to enhance the capabilities of Symbolic Pathfinder. Although it is being developed actively, but some of the methodologies are not yet supported or remain unimplemented.

1. **Current Limitations:** As mentioned currently SPF has limited support for runtime exceptions for String methods when the program operates on symbolic inputs.
2. **Missed Error Paths:** When an operation could potentially throw an exception depending on the symbolic input's value, SPF might not explore the specific error path. It will stop or execute the operation concretely, thus failing to detect potential bugs that only manifest under specific input conditions leading to an exception.
3. **Previous Works:** While previous works on SPF helped it in being compatible with different versions and technologies, but the efforts made need some finalization.

This work that will be done on this project will help SPF to explore critical exceptional program paths, leading to more comprehensive analysis and the detection of previously unhandled defects.

This project will also include some investigations around the migration of SPF to different versions of Java and Gradle, and will see how the core concepts change because of them. Which will make SPF a more capable and reliable version of itself, increasing its value for software verification and testing is the main motivation.

Goals

The current implementations in SPF show or present certain limitations that affect its ability. The goals outlined below are designed to achieve what it takes to make SPF a more capable tool.

1. **Runtime Exception Handling:** As mentioned, earlier this goal aims to detect and handle runtime exceptions.
    - Analyze and document the existing mechanism within SPF (on the stable sv-comp/Java 8 baseline) for handling potential ***NullPointerException*** when operations involve symbolic object reference. This includes tracing execution through relevant bytecode handlers (e.g., invokevirtual).
    - Implementing symbolic handling for **IndexOutOfBoundsException** and other such runtime exceptions within ***SymbolicStringHandler*** focusing on methods like *substring, charAt, indexOf, replace*, and etc. This involves introducing **PCChoiceGenerators** to model branching between valid execution and exception paths, which adds appropriate constraints to the ***PathConditon.***
    - Writing specific test cases using symbolic inputs to verify that SPF correctly explores paths that lead to the runtime exceptions that are not currently supported.
    - Documenting the implementation mechanisms and choices, with some usage examples to make it easy for the future developments.

2. **Finalize and Stabilize Java 11 + Gradle support for SPF:** As mentioned earlier, this goal aims to polish the remaining tasks from the Java 11 build branch.
    - Identify the pending tasks and existing issues required to complete the stable Java 11 build using Gradle, based on the work and contributions on branches like *gradle-build-java-11,* we will probably use a higher version of Gradle than the current version (6.9) in *sv-comp.*

- Investigate the impact of Java 11's module system and API changes to SPF's core functionality, particularly regarding the handling of modeled Java classes (e.g., ***java.lang.Math***, ***java.util.Scanner***). This can be a direct consequence of Project *JIGSAW* because it introduced the Java Platform Module System.
- Testing and Documenting my implementations and investigations to validate Java 11's compatibility across SPF.

Apart from these goals there is PR of mine #111, I've had long discussions with my mentor over this, furthermore I need to investigate more on the IEEE 754 standard for floating point arithmetics and recent comments that ask to what extent are my variations close to OpenJDK's implementations.

Concrete Implementation Plan

I will try to complete this project over the standard GSoC timeline, including the community bonding period and a 12 - 22 weeks coding period. The plan is structured in phases to address the goals mentioned above.

Community Bonding Period: (Approx. 4 Weeks, Pre-Coding)

- I will familiarize myself more with the internals of Symbolic PathFinder, especially the Instruction Factory, Handler, Listener, ChoiceGenerator, PathCondition and Z3.
- I will check out the SPF code base and the files and folders located under */jpf-symbc/src* (primarily *sv-comp* and relevant *gradle-build-java-11*).
- I will try to run more existing examples under the *sv-comp* branch to understand the current state of SPF more accurately.
- I will analyze the *gradle-build-java-11* branch, associated PR's (e.g, #96 and #112) and the issue (#114).
- Work on mr PR (#111) and try to cover all the changes and refactoring asked by my mentor.
- I will also explore JPF and will try to learn more about it from the JPF Wiki.
- Discuss my findings, plans and priorities with my mentor.
- Engage in the JPF community with the fellow contributors.

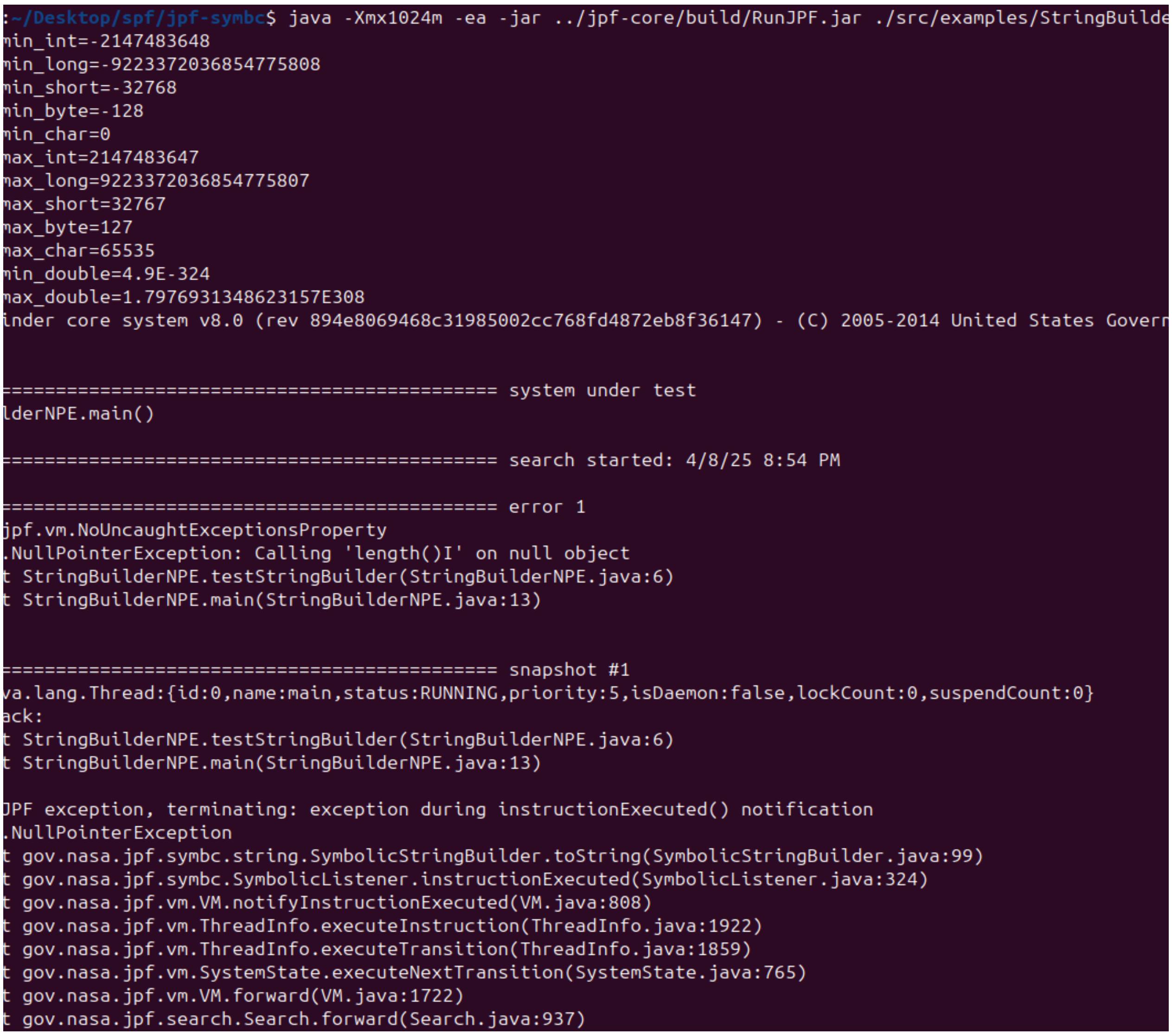
Phase 1: Support Core Exception Handling (Weeks 1 - 4)

- Let’s see how SPF checks/detects ***NullPointerException***. Below is the code I added in the examples directory of jpf-symbc.

```
import gov.nasa.jpf.symbc.Debug;

public class StringBuilderNPE {
   public static void testStringBuilder(StringBuilder sb) {
       int length = sb.length();
       System.out.println("StringBuilder length: " + length);
   }

   public static void main(String[] args) {

       StringBuilder symSB =
(StringBuilder)Debug.makeSymbolicRef("symSB", new
StringBuilder());

       testStringBuilder(symSB);
   }
}
```

```
target=StringBuilderNPE
classpath=${jpf-symbc}/build/examples
sourcepath=${jpf-symbc}/src/examples

symbolic.dp=z3
symbolic.string_dp=z3str3
symbolic.string=true
symbolic.method=StringBuilderNPE.testStringBuilder(sym)
search.multiple_errors=true
search.depth_limit=30
listener=gov.nasa.jpf.symbc.SymbolicListener
```

- When we run this example and look at the output on the console, we can clearly see that there is a ***NullPointerException*** here.



```
:~/Desktop/spf/jpf-symbc$ java -Xmx1024m -ea -jar ../jpf-core/build/RunJPF.jar ./src/examples/StringBuilde
nin_int=-2147483648
nin_long=-9223372036854775808
nin_short=-32768
nin_byte=-128
nin_char=0
nax_int=2147483647
nax_long=9223372036854775807
nax_short=32767
nax_byte=127
nax_char=65535
nin_double=4.9E-324
nax_double=1.7976931348623157E308
inder core system v8.0 (rev 894e8069468c31985002cc768fd4872eb8f36147) - (C) 2005-2014 United States Govern


============================================= system under test
lderNPE.main()

============================================= search started: 4/8/25 8:54 PM

============================================= error 1
jpf.vm.NoUncaughtExceptionsProperty
.NullPointerException: Calling 'length()I' on null object
t StringBuilderNPE.testStringBuilder(StringBuilderNPE.java:6)
t StringBuilderNPE.main(StringBuilderNPE.java:13)


============================================= snapshot #1
va.lang.Thread:{id:0,name:main,status:RUNNING,priority:5,isDaemon:false,lockCount:0,suspendCount:0}
ack:
t StringBuilderNPE.testStringBuilder(StringBuilderNPE.java:6)
t StringBuilderNPE.main(StringBuilderNPE.java:13)

JPF exception, terminating: exception during instructionExecuted() notification
.NullPointerException
t gov.nasa.jpf.symbc.string.SymbolicStringBuilder.toString(SymbolicStringBuilder.java:99)
t gov.nasa.jpf.symbc.SymbolicListener.instructionExecuted(SymbolicListener.java:324)
t gov.nasa.jpf.vm.VM.notifyInstructionExecuted(VM.java:808)
t gov.nasa.jpf.vm.ThreadInfo.executeInstruction(ThreadInfo.java:1922)
t gov.nasa.jpf.vm.ThreadInfo.executeTransition(ThreadInfo.java:1859)
t gov.nasa.jpf.vm.SystemState.executeNextTransition(SystemState.java:765)
t gov.nasa.jpf.vm.VM.forward(VM.java:1722)
t gov.nasa.jpf.search.Search.forward(Search.java:937)
```

- symbolic.method tells SPF to execute the method **testStringBuilder** symbolically as the sym specifier for object reference in SPF typically includes null in the possible domain by default.
- The values are unconstrained i.e., its value can be anything hence it may be null or not-null which creates a branching point in the execution.
- SPF will explore both execution paths.
- In the str.length() call, before accessing the length property:
- SPF checks if the current path condition allows str to be null
- On the path where str == null, it identifies the NullPointerException.
- This happens through JVM instrumentation, not through string-specific models.

- The JPF file settings enable this detection.
- symbolic.method=StringBuilderNPE.testLength(sym) makes str symbolic
- search.multiple_errors=true allows finding all exception cases
- symbolic.string=true enables string symbolic execution
- But that is not the case with the String methods, the lack because SPF uses special symbolic handlers that replace the actual execution with symbolic constraint generation. These handlers and related classes model string operations but don't properly check for null values.
- Identify the Method: Locate the symbolic model for the string method which is to be modified (e.g., substring, equals, etc.).
- Add Null Checks: Before performing the symbolic operation, check if the string reference is null.
- Generate Path Conditions: If the reference is null, add a path condition to indicate this and throw a symbolic exception.
- Handle Invalid Arguments: For methods like substring, check for invalid indices and generate constraints for these cases.
- Testing is very important. We will test the support added, including cases where the string might be null when using substring or most probably they can also have index out of bound errors as well.
- The pattern can be "Check ⇒ Branch ⇒ Constraints ⇒ Act/Throw", the same pattern can also be applied for corresponding bytecode handlers like iload, idiv etc. Implement exception handling logic in a centralized location, such as a utility class or within the symbolic model itself. This ensures consistency and reduces code duplication.
- Targets will be handleSubstring1, handleSubstring2, handleCharAt etc, inside SymbolicStringHandler.
- Missing Null Reference Checking: The symbolic string methods (like substring, length, etc.) throw runtime exceptions when operands aren't symbolic, but don't check if the string reference itself is null.
- Lack of Path Condition Generation: When exceptions should be thrown (e.g., for null strings), the code doesn't generate appropriate path conditions to represent these scenarios.
- Inconsistent Exception Handling: Different methods handle errors differently - some throw exceptions, others silently create default values.

Phase 2: Java 11 stabilisation and core class handling (Weeks 5 - 8)

- I will implement Java 11 build fixes, address the prioritized list of pending tasks for the Java 11/Gradle 8.4 + build.
- Deliverable can be a version of SPF that passes the test under OpenJDK 11.
- Implement the chosen strategy for handling core Java classes affected by JPMS (Math, Scanner, etc).
- Investigate the work done under PR's #96 and #112.
- I will try to solve some of the TODOS that are currently present in SPF.
- Verify successful build and execution of the gradle-build-java-11 branch.
- Post-JPMS is strongly encapsulated within the java.base module. The standard classloader mechanisms prioritize platform modules, making the simple replacement strategy difficult or impossible.
- The --patch-module allows the replacing or augmenting the content of a compiled module. It might be a way to inject SPF's custom versions of Math and Scanner.
- Investigation needs to determine if this is feasible or not, and how it interacts with JPF's classloading.
- Another possibility is to maintain the native peer classes and intercept the bytecode that is needed to be executed symbolically that are not native.

Phase 3: Extend support and other goals (Weeks 9 - 10)

- Try to solve the open issues related to string and Java 11 support.
- Work on my PR, detailed investigation on the IEEE 754 standards, align with the OpenJDK implementations wherever possible.
- Analyze other methods in ***SymbolicStringHandler*** for potential symbolic exceptions and apply the same patterns.
- Expand the test cases to cover newly handled string methods.
- Implement other goals that can be additional or given by my mentors.
- Refactor my contributions, code and implementations, keep in mind the coding convention that is followed by the JPF Team.
- Document the challenges I faced and how I planned to deal with them during my coding period. One such can be a good understanding of JVM and JPF internals.

Phase 4: Documentation, Finalization and Submission (Weeks 11 - 12)

- Write and finalize all the documentations related to my work on SPF. Detailed explanation of how I supported runtime exceptions in SPF, Java 11 support, migration to a higher Gradle version, test case descriptions.
- If needed I will update the readme/wiki of SPF.
- Code cleanup, refactor for clarity, efficiency. I will also list the coding standards and styles I have followed.
- Execute all test cases one final time on both Java 8 and Java 11.
- Utilize remaining time for unforeseen delays, addressing final review feedback or exploring minor improvements.
- This time will be highly dedicated to testing and documenting my progress on SPF.
- Will discuss backup plans with my mentor if something does not work out as intended.
- As the project is of 350 hours, if needed I can extend or schedule the additional tasks till November.

I think this is a highly stable plan which breaks down how I will work if I get selected for GSoC 2025. Moreover I have watched the Symbolic PathFinder tutorial by Corina Pesareanu and have read some articles/materials by authors like Neha Rungta, Kasper S. Luckow, Runtime Exception Detection in Java programs using Symbolic Execution and a very good reading material on Symbolic String Execution by Gideon Redelinghuys.

There now I am very much familiar with the symbolic execution of programs, and will comply with the goals and implementations mentioned above.

Development Environment

IntelliJ IDEA Community Edition

OpenJDK 8, 11

Gradle, currently 6.9

Ubuntu 24 LTS

Timeline

Pre GSOC Period	
9th April - 7th May	● Solve issues on SPF's Github Repo. ● Work on my PR #111, what is left behind, once the coding period begins I will add more support to my PR.
Community Bonding Period	
8th May - 1st June	● Get to know more about the JPF/SPF community. ● Discussion with my mentor over SPF ● Review PR's and related issues to understand Java 11 support in SPF.
Coding period	
2nd June - 8th June	● Analyze SPF's current handling of NullPointerException for String methods. ● Set up a development environment for symbolic exception handling.
9th June - 15th June	● Implement symbolic handling for NullPointerException in SymbolicStringHandler. ● Begin testing with symbolic inputs.
16th June - 22nd June	● Expand exception handling to methods like substring, charAt, indexOf, replace, etc. ● Create PCChoiceGenerators to handle branching between valid and exception paths.

23rd June - 29th June	● Write unit tests to validate symbolic exception paths. ● Refactor code based on test outcomes and mentor feedback.
30th June - 6th July	● Finalize phase 1 with any remaining tasks and start documenting my progress.
7th July - 13th July	● Prepare for the **midterm evaluation**
14th July - 18th July	● Submit the midterm evaluation with progress updates, challenges, and deliverables. ● Prepare a demo of exception handling features.
19th July - 25 July	● Implement Java 11 build fixes and address pending TODOs related to Java 11 in the SPF codebase. ● Identify compatibility issues related to JPMS (Java Platform Module System) and document them.
26th July - 1st August	● If --patch-module isn't feasible, maintain native peer classes and intercept relevant bytecode for symbolic execution. ● Explore classloading interactions between JPMS and JPF to determine alternative strategies.
2nd August - 8th August	● Test SPF thoroughly with OpenJDK 11, ensuring all core classes behave correctly under symbolic execution

	● Prepare detailed documentation on the strategies used for Java 11 support and finalize the phase 2.
9th August - 15th August	● Address open issues related to symbolic string handling and Java 11 compatibility.Continue working on PR #111, ensuring it aligns with recent changes and improvements.Conduct a detailed investigation of the IEEE 754 standard for floating-point arithmetic, comparing SPF's implementation with OpenJDK.
16th August - 22nd August	● Implement symbolic handling for additional methods. Expand test cases to cover newly supported scenarios, including edge cases. Refactor the codebase to ensure clarity, efficiency, and adherence to JPF's coding standards. Document the challenges faced, solutions implemented, and key insights gained throughout the coding period and finalize the phase 3.
23rd August - 29th August	● Write comprehensive documentation covering the implementation of runtime exception support in SPF. Strategies for Java 11 support and migration to Gradle 8.4+. Detailed descriptions of test cases and their results. Update SPF's README and Wiki pages to reflect the new features

	and improvements.Perform final code cleanup and refactoring for maintainability. Review and apply coding standards and best practices followed by the JPF team. ● **Final Week**
30th August- 8th September	● Execute all test cases across Java 8 and Java 11 to verify stability and correctness. Discuss any backup plans with my mentors for any unseen challenges. ● Finalize the phase 4 ● **Final Evaluation Period starts**
1st September - 17th November	● As the project is of 350 hours, if needed I can extend or schedule the additional tasks till November. ● Then I will help my fellow contributors as well. ● Update the documentations and work on additional tasks given by the mentors.

I have carefully allocated enough time to each task, and have factored in testing and documentation time. Additionally, I will also keep some buffer days between tasks to account for any unforeseen delays or if any previous tasks require additional time. This will ensure that even if any critical bugs arise, I can handle them without impacting the project's overall progress.

Additional Information

Availability

With no other obligations this summer, GSoC takes precedence as my sole focus. I am prepared to dedicate around 48-50 hours per week to this project. I will generally work from 12 AM to 8 PM IST, although it is flexible to adjust my working hours if required. Besides this project, I have no other commitments or vacations planned for the summer. I shall keep my status posted to the JPF community and to the mentor every week and maintain transparency in the project.

Scheduling Issues

I do not have any scheduling issues as such, the only thing left are my End Sems or Final Examinations which always take place in May or June. The exam dates have not been announced by the university yet. This semester, I have 4 subjects along with a major academic project, which I have already completed, along with its presentation/seminar. I will update my mentor as soon as the exam schedule is released. During this period, I will need to adjust my working hours on the project, but I plan to dedicate 4-5 hours a day (28-35 hours a week) to it since subjects are not too tough, and the time-consuming major project is already done. However, I will still need some time to prepare for the exams, and I will inform my mentor and keep her updated about this.

Presence in the community

As a newcomer, I prioritize engaging with my fellow contributors on Discord, where I have both asked and answered questions to help others. Although I am new, I have actively participated in discussions and have in-depth conversations with the mentors, especially with Soha Hussein, regarding my project. Additionally, I have a PR open on SPF's sv-comp branch, where I've had discussions and received valuable feedback, including comments and suggested changes from my mentor on the contributions made.

Post GSoC Period

After my completion of GSoC, I will still contribute to the organization. I will look at the open issues #109 and will try to solve them, because of my familiarity with the tech stack now. I plan to address new issues as well. The primary goal after the GSoC period ends, is to add more and more support to SPF, which will make it a high rank tool. Other than that I will explore JPF as well, and will also try to contribute to it, as I plan again to apply at JPF for GSoC 2026 if it happens. I will help others wherever possible either Discord community, Github discussions, issues, PR's, etc. As I mentioned earlier, contributing to JPF/SPF after the GSoC, will help me in my career ahead as it will serve as a proof that I know Java well and have some development experience.

Conclusion

Thank you for taking the time to read my proposal. I have provided a detailed overview of my project and how I plan to execute it. Among all the organizations in GSoC 2025, JPF attracted me the most, and I ended up choosing Support Runtime Exception in SPF as a project to contribute on, because it interests me the most. I think I have proved that I am ready to participate in this project.

For GSoC 2025 my main goal in person is to enhance my knowledge in open-source software development, increasing my practical experience by building and understanding the project. As for the tools and technologies (tech stack), I am well versed with all the necessities required for this project. I am confident that I can complete this project within the given timeline and take full responsibility for implementing and supporting missing concepts from SPF. Solving problems in SPF will be my primary focus.

Please consider my proposal. Thank you once again.

I am 100% dedicated to JPF and have no plans whatsoever to submit a proposal to any other organization.