Google Summer of Code 2025

Support portfolio of solvers in SPF

Rehan Chalana

1. Abstract

Java PathFinder (JPF) is an extensible software analysis framework for java bytecode which can be used to check for concurrency defects like deadlocks, and unhandled exceptions like *NullPointerExceptions* and *AssertionErrors*. Symbolic PathFinder (SPF) is its extension that enables symbolic execution of java programs.

This proposal outlines my plan to contribute to **The JPF Team** by working on the **Support portfolio of solvers in SPF** Project during Google Summer of Code 2025.

The main goal of this project is to enable the simultaneous invocation of multiple constraint solvers, where execution halts as soon as any solver returns a satisfactory result. This approach is expected to enhance SPF's ability to handle a broader range of constraints.

This Project will extend SPF branch *sv-comp*. The outcome will enhance SPF's scalability and applicability in symbolic execution tasks.

Project Scope: ~350 hours

2. Self Introduction

I am Rehan Chalana, a sophomore at Chitkara University pursuing Bachelor of Engineering in Computer Science. My passion for technology has driven me to explore various technologies, including *Java, Kotlin, Go, JavaScript, Spring Boot, React.js, Node.js, PostgreSQL, Docker* and more.

Open Source Involvement

My open-source journey began when I first used Linux (Ubuntu). I was fascinated by the idea that such a powerful operating system was free and built by a global community. Since then, I have actively contributed to open-source projects.

As a mentee in the Linux Foundation Mentorship Program (CNCF-KCL), I integrated *LSP4IJ* support into the IntelliJ-KCL plugin and migrated the project from the *Gradle IntelliJ Plugin* to the *IntelliJ Platform Gradle Plugin*. Below are some of my significant contributions from the mentorship program.

PR #38 Set up LSP4IJ support
PR #39 Resolve kcl-language-server from system environment variables
PR #44 Migrate to Intellij Platform Plugin

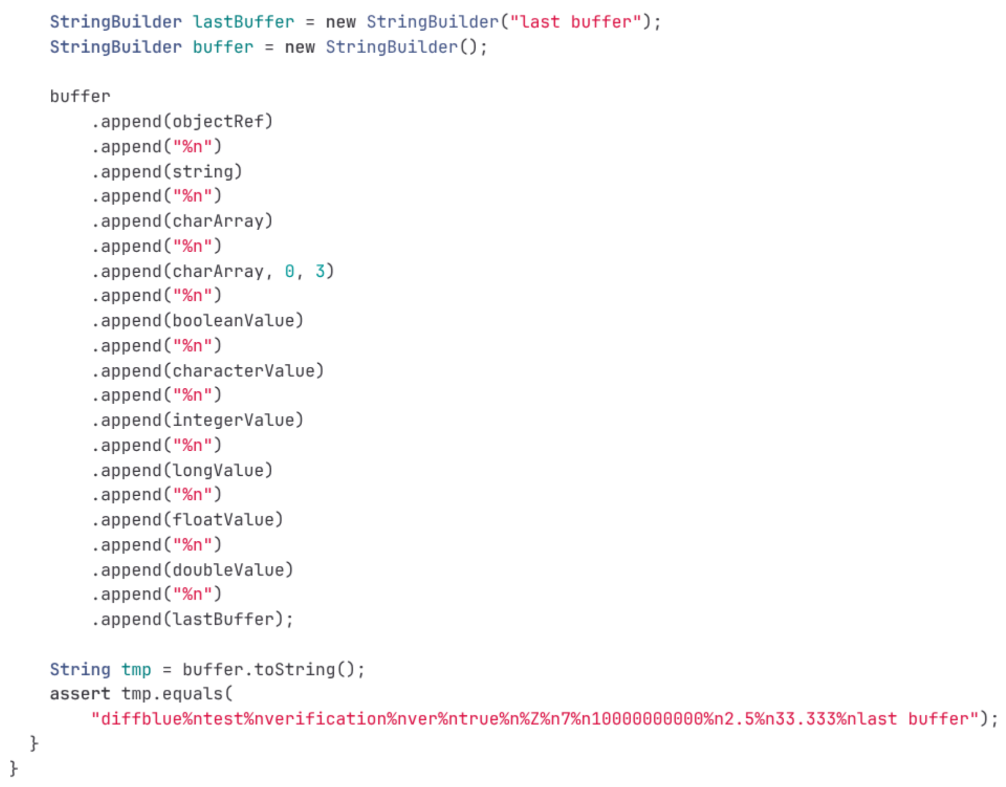
In addition to the mentorship, I have participated in open-source programs like Hacktoberfest and Social Winter of Code, where I contributed to projects such as HTML Sanity Check, JabRef and others.

Beyond individual contributions, I am committed to fostering the open-source movement. As the Programming Lead at Open Source Chandigarh, a student community at my university, I organize events and workshops that introduce students to FOSS and encourage active participation in open-source projects.

Contributions to The JPF Team

While exploring open issues to contribute to, I came across [Issue #108: Additional String Support is Needed](#).

I began investigating the problem and examined the test file *`jbmc-regression/StringBuilderAppend02.yml`* along with its corresponding Java file *`Main.java`*. The tests were designed to verify the functionality of various *`StringBuilder.append()`* methods.



```
    StringBuilder lastBuffer = new StringBuilder("last buffer");
    StringBuilder buffer = new StringBuilder();

    buffer
        .append(objectRef)
        .append("%n")
        .append(string)
        .append("%n")
        .append(charArray)
        .append("%n")
        .append(charArray, 0, 3)
        .append("%n")
        .append(booleanValue)
        .append("%n")
        .append(characterValue)
        .append("%n")
        .append(integerValue)
        .append("%n")
        .append(longValue)
        .append("%n")
        .append(floatValue)
        .append("%n")
        .append(doubleValue)
        .append("%n")
        .append(lastBuffer);

    String tmp = buffer.toString();
    assert tmp.equals(
        "diffblue%ntest%nverification%nver%ntrue%n%Z%n7%n10000000000%n2.5%n33.333%nlast buffer");
  }
}
```

Upon analysis, I discovered that the test was failing due to unsupported variants of *`StringBuilder.append()`*, specifically:

- *StringBuilder.append(char[])*
- *StringBuilder.append(char[], int start, int end)*

To begin addressing this gap, I raised [PR #115: Add support for `StringBuilder.append(char[])`](), which introduces initial support for the *`append(char[])`* variant in Symbolic PathFinder (SPF).

This contribution is part of an ongoing effort to expand and improve symbolic string handling in the JPF ecosystem.

Why The JPF Team Organization ?

Java was the first programming language I learned, and ever since, I have been deeply fascinated by the Java ecosystem. Over the years, I have explored various Java technologies and frameworks, but I have always wanted to understand the language at a deeper, more fundamental level.

Working with The JPF Team offers a unique opportunity to dive into the core internals of java and learn new concepts, such as bytecode, symbolic execution, and model checking. This project aligns perfectly with my goal of strengthening my low-level understanding of Java and making meaningful contributions to the Java ecosystem and open source community.

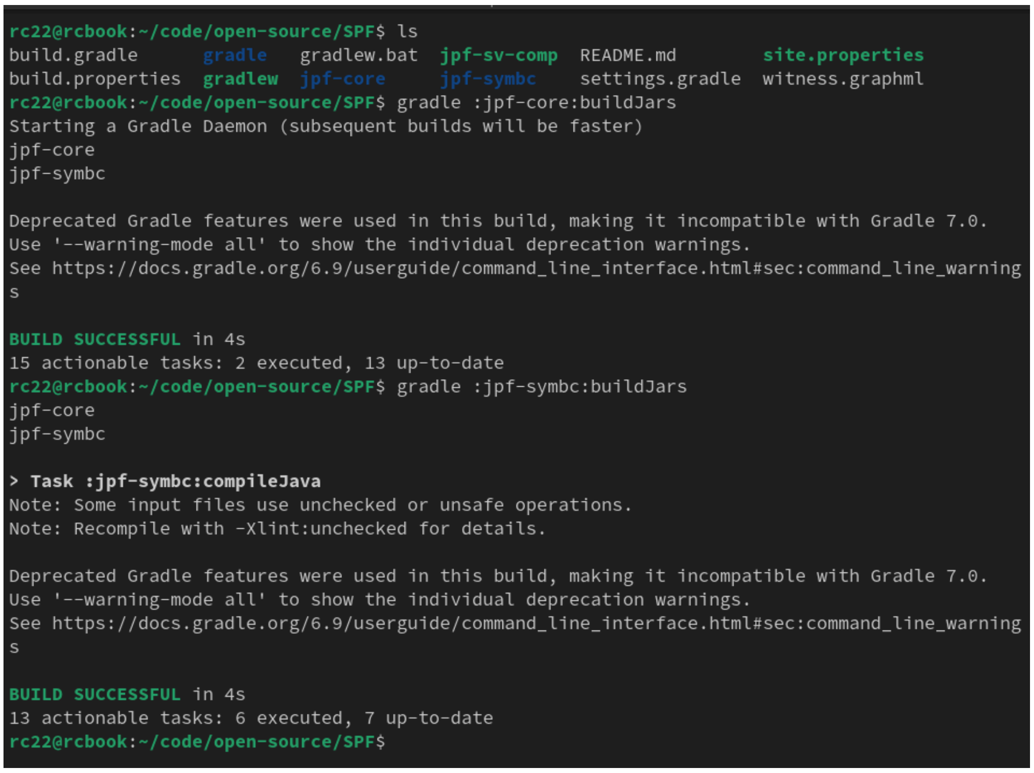
Also I was really fascinated by learning the history of JPF that it was originally developed at NASA Ames Research Center in 1999. As someone with a deep interest in space and astrophysics, it means a lot to me to potentially contribute to a project that has its roots in one of the world’s leading space research institutions. Being part of a project with such great legacy and research-driven fascination is very inspiring, and I’m excited about the possibility of contributing to it through Google Summer of Code.

Contact info and timezone

Primary Email: rehanchalanaprsnl@gmail.com
Contact Number: +91 9855519509
GitHub: RehanChalana
Discord: rc__22
Twitter: rehan_chalana
Time Zone: Kolkata, India (GMT+5:30)
Preferred mode of Communication: Email, Google meet, Discord.

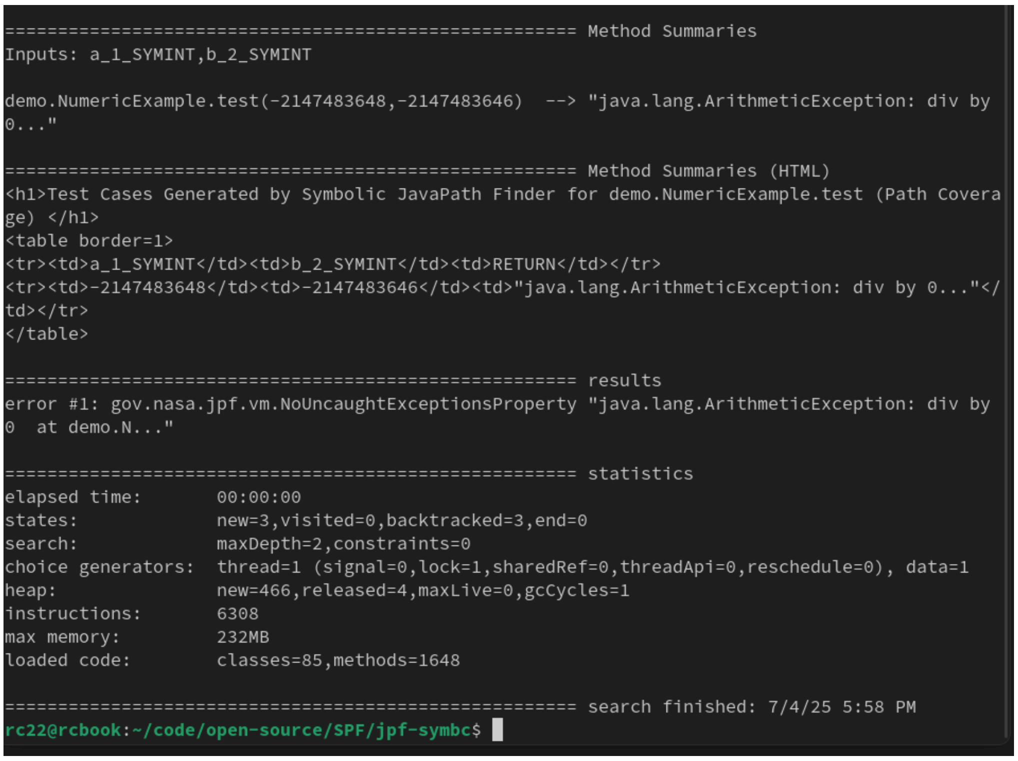
3. Essential Prerequisites

1) I have successfully compiled and run both jpf-core and jpf-symbc on my local machine.



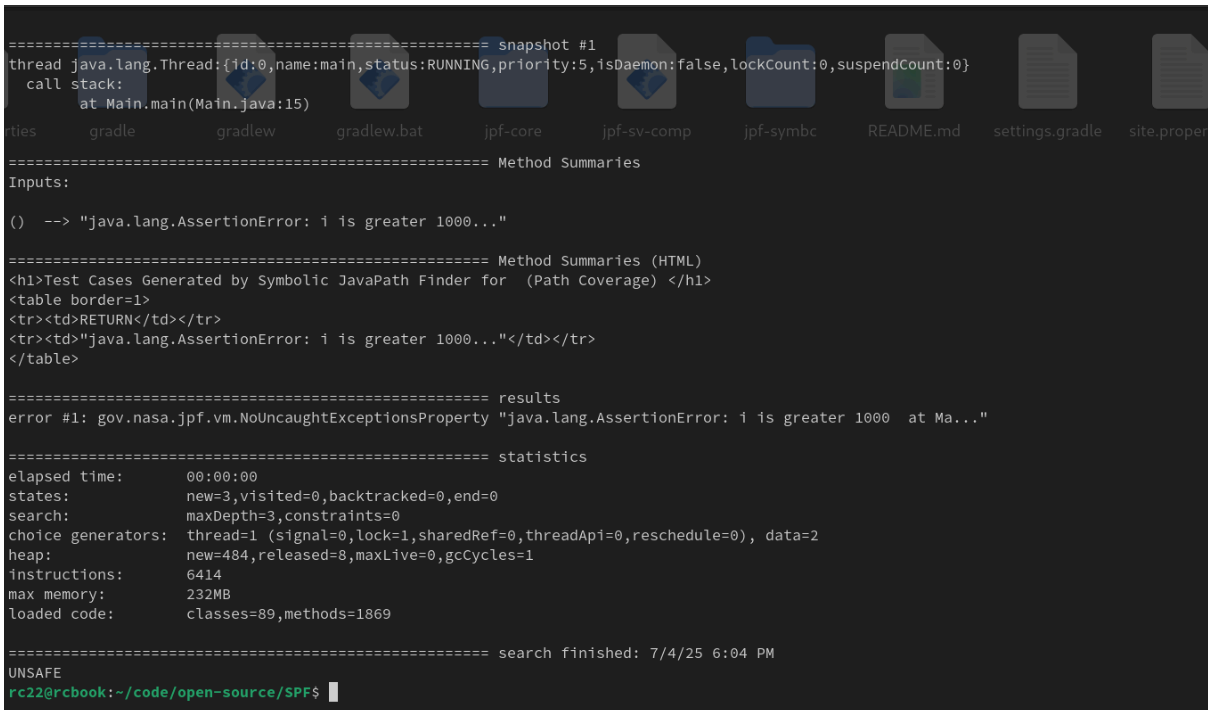
```
rc22@rcbook:~/code/open-source/SPF$ ls
build.gradle      gradle   gradlew.bat  jpf-sv-comp  README.md        site.properties
build.properties  gradlew  jpf-core     jpf-symbc    settings.gradle  witness.graphml
rc22@rcbook:~/code/open-source/SPF$ gradle :jpf-core:buildJars
Starting a Gradle Daemon (subsequent builds will be faster)
jpf-core
jpf-symbc

Deprecated Gradle features were used in this build, making it incompatible with Gradle 7.0.
Use '--warning-mode all' to show the individual deprecation warnings.
See https://docs.gradle.org/6.9/userguide/command_line_interface.html#sec:command_line_warning
s

BUILD SUCCESSFUL in 4s
15 actionable tasks: 2 executed, 13 up-to-date
rc22@rcbook:~/code/open-source/SPF$ gradle :jpf-symbc:buildJars
jpf-core
jpf-symbc

> Task :jpf-symbc:compileJava
Note: Some input files use unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

Deprecated Gradle features were used in this build, making it incompatible with Gradle 7.0.
Use '--warning-mode all' to show the individual deprecation warnings.
See https://docs.gradle.org/6.9/userguide/command_line_interface.html#sec:command_line_warning
s

BUILD SUCCESSFUL in 4s
13 actionable tasks: 6 executed, 7 up-to-date
rc22@rcbook:~/code/open-source/SPF$
```

2) I was also able to run a sample example as described in the docs.



```
====================================================== Method Summaries
Inputs: a_1_SYMINT,b_2_SYMINT

demo.NumericExample.test(-2147483648,-2147483646)  --> "java.lang.ArithmeticException: div by
0..."

====================================================== Method Summaries (HTML)
<h1>Test Cases Generated by Symbolic JavaPath Finder for demo.NumericExample.test (Path Covera
ge) </h1>
<table border=1>
<tr><td>a_1_SYMINT</td><td>b_2_SYMINT</td><td>RETURN</td></tr>
<tr><td>-2147483648</td><td>-2147483646</td><td>"java.lang.ArithmeticException: div by 0..."</
td></tr>
</table>

====================================================== results
error #1: gov.nasa.jpf.vm.NoUncaughtExceptionsProperty "java.lang.ArithmeticException: div by
0  at demo.N..."

====================================================== statistics
elapsed time:       00:00:00
states:             new=3,visited=0,backtracked=3,end=0
search:             maxDepth=2,constraints=0
choice generators:  thread=1 (signal=0,lock=1,sharedRef=0,threadApi=0,reschedule=0), data=1
heap:               new=466,released=4,maxLive=0,gcCycles=1
instructions:       6308
max memory:         232MB
loaded code:        classes=85,methods=1648

====================================================== search finished: 7/4/25 5:58 PM
rc22@rcbook:~/code/open-source/SPF/jpf-symbc$
```

3) I was also able to run a benchmark at SV-COMP on SPF.



```
====================================================== snapshot #1
thread java.lang.Thread:{id:0,name:main,status:RUNNING,priority:5,isDaemon:false,lockCount:0,suspendCount:0}
  call stack:
        at Main.main(Main.java:15)

====================================================== Method Summaries
Inputs:

()  --> "java.lang.AssertionError: i is greater 1000..."

====================================================== Method Summaries (HTML)
<h1>Test Cases Generated by Symbolic JavaPath Finder for  (Path Coverage) </h1>
<table border=1>
<tr><td>RETURN</td></tr>
<tr><td>"java.lang.AssertionError: i is greater 1000..."</td></tr>
</table>

====================================================== results
error #1: gov.nasa.jpf.vm.NoUncaughtExceptionsProperty "java.lang.AssertionError: i is greater 1000  at Ma..."

====================================================== statistics
elapsed time:       00:00:00
states:             new=3,visited=0,backtracked=0,end=0
search:             maxDepth=3,constraints=0
choice generators:  thread=1 (signal=0,lock=1,sharedRef=0,threadApi=0,reschedule=0), data=2
heap:               new=484,released=8,maxLive=0,gcCycles=1
instructions:       6414
max memory:         232MB
loaded code:        classes=89,methods=1869

====================================================== search finished: 7/4/25 6:04 PM
UNSAFE
rc22@rcbook:~/code/open-source/SPF$ 
```

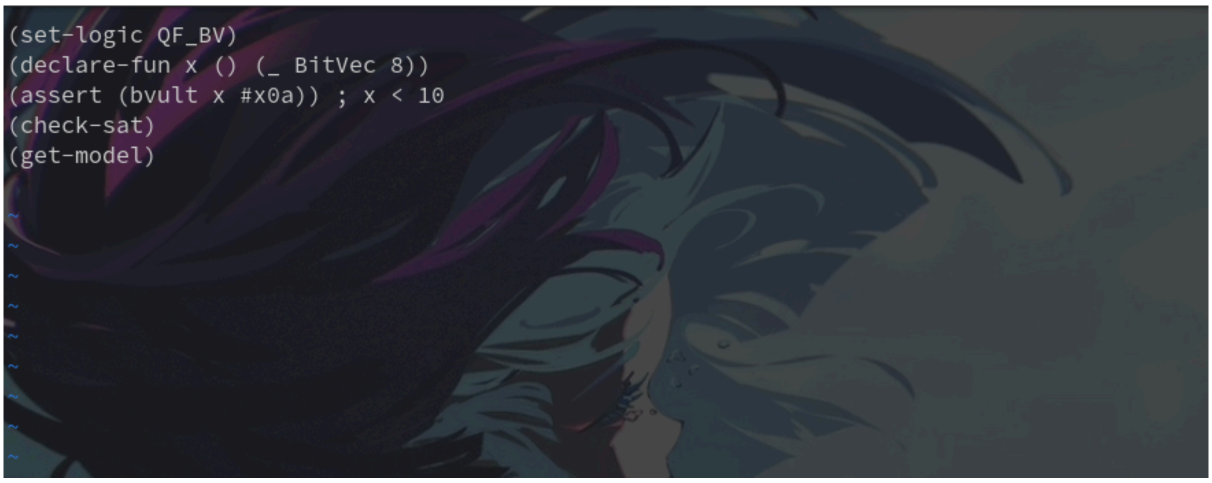
4) I was also able to try some examples on both z3 and cvc5 to understand the differences between their input and output, Here are my findings.

Setup:

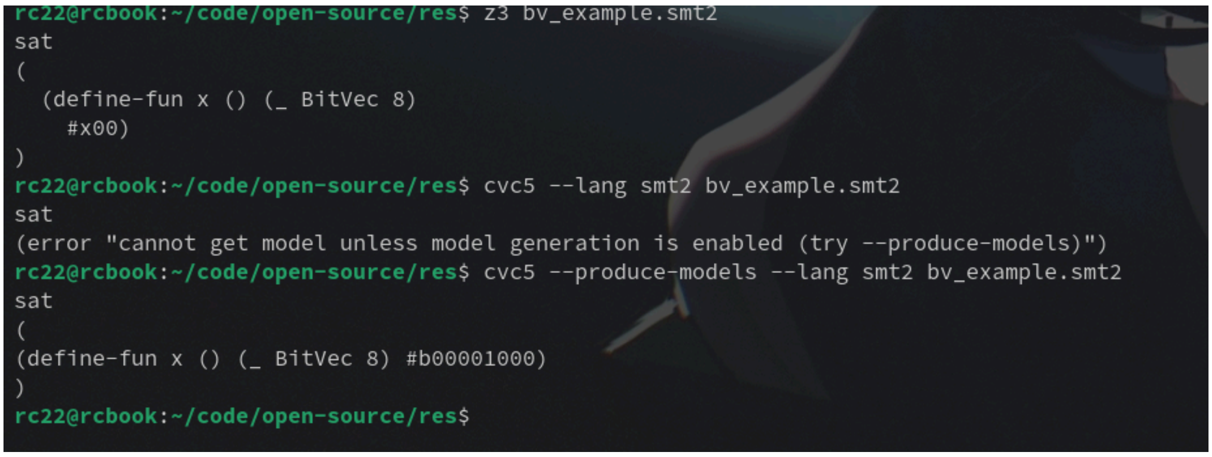
I ran equivalent SMT-LIB 2.0 examples on both solvers, testing them independently to observe:

- Input syntax acceptance
- Constraint solving behavior
- Output formatting and model generation

Example 1: Find an 8-bit value `x` such that `x < 10`



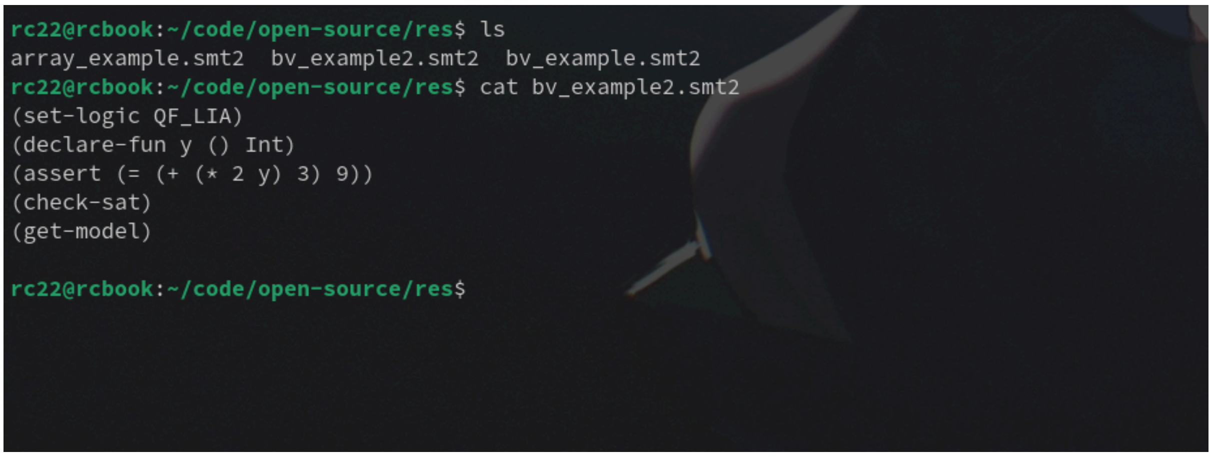
```
(set-logic QF_BV)
(declare-fun x () (_ BitVec 8))
(assert (bvult x #x0a)) ; x < 10
(check-sat)
(get-model)
~
~
~
~
~
~
~
~
~
```



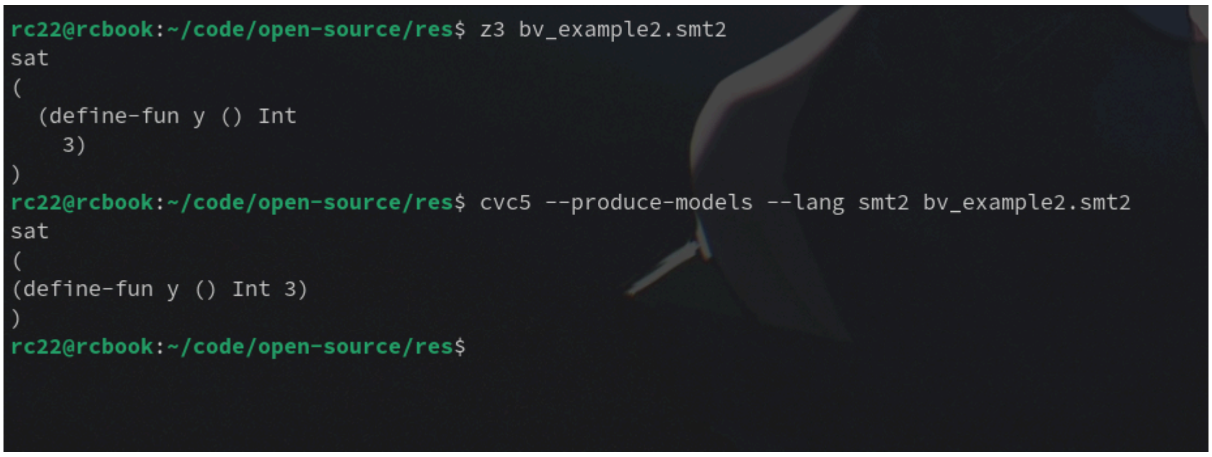
```
rc22@rcbook:~/code/open-source/res$ z3 bv_example.smt2
sat
(
  (define-fun x () (_ BitVec 8)
    #x00)
)
rc22@rcbook:~/code/open-source/res$ cvc5 --lang smt2 bv_example.smt2
sat
(error "cannot get model unless model generation is enabled (try --produce-models)")
rc22@rcbook:~/code/open-source/res$ cvc5 --produce-models --lang smt2 bv_example.smt2
sat
(
(define-fun x () (_ BitVec 8) #b00001000)
)
rc22@rcbook:~/code/open-source/res$
```

- Z3 returns a valid model (x = 0) and supports get-model by default.
- CVC5, on the other hand, requires the *--produce-models* flag to enable model generation. Without it, it returns *sat* but emits an error when *(get-model)* is requested.
- Additionally, for SMT-LIB 2 input, CVC5 requires explicitly passing the *--lang smt2* flag on the command line.
- Both return a valid model.

Example 2: Find an integer y such that 2 * y + 3 = 9.



```
rc22@rcbook:~/code/open-source/res$ ls
array_example.smt2  bv_example2.smt2  bv_example.smt2
rc22@rcbook:~/code/open-source/res$ cat bv_example2.smt2
(set-logic QF_LIA)
(declare-fun y () Int)
(assert (= (+ (* 2 y) 3) 9))
(check-sat)
(get-model)

rc22@rcbook:~/code/open-source/res$
```



```
rc22@rcbook:~/code/open-source/res$ z3 bv_example2.smt2
sat
(
  (define-fun y () Int
    3)
)
rc22@rcbook:~/code/open-source/res$ cvc5 --produce-models --lang smt2 bv_example2.smt2
sat
(
(define-fun y () Int 3)
)
rc22@rcbook:~/code/open-source/res$
```

- Z3 handles the query directly and returns a correct model: *y = 3*.
- CVC5 also produces the correct model when *--produce-models* and *--lang smt2* are passed.
- This reinforces that the input syntax (SMT-LIB) is largely compatible across solvers.

4. Project Goals

1) Study existing solver integration in SPF:
Analyze how different solvers (e.g., Z3, CVC) are currently integrated within the SPF codebase and run comparative experiments with *Z3 and CVC* on a variety of SMT-LIB queries.

2) Design an architecture for parallel solver execution:
Create a design that allows multiple solver instances to be invoked concurrently for the same query, while ensuring minimal disruption to the existing codebase.

3) Implement the Proposed Architecture:
Develop and integrate the designed system into SPF, extending the `sv-comp` branch as a base.

4) Test and evaluate the implementation:
Thoroughly test the parallel solver support with a variety of benchmark constraints to ensure correctness, performance, and reliability.

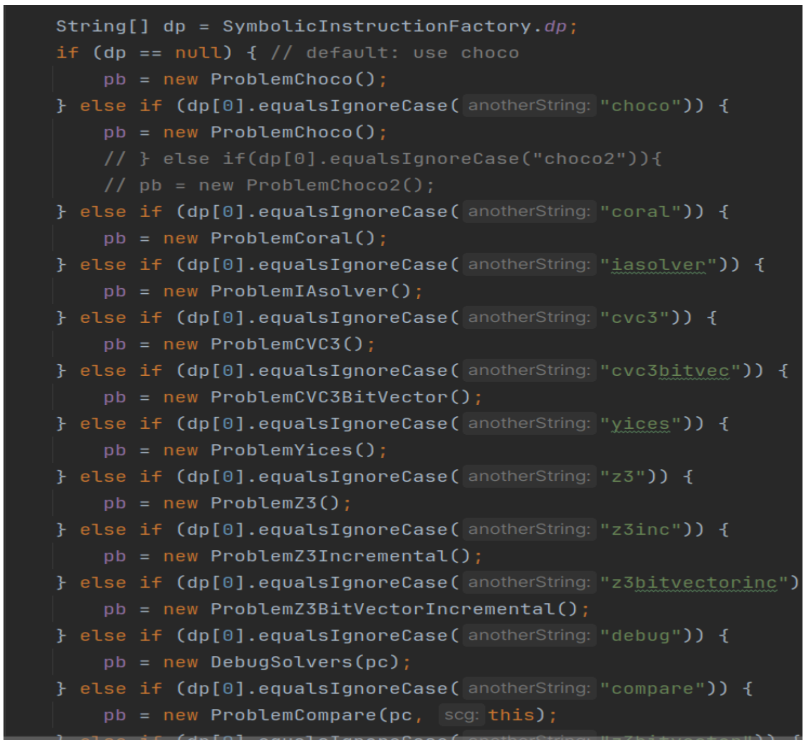
5) Document the Design and Implementation:
Provide comprehensive documentation covering the architecture, implementation details, usage instructions, and configuration options for parallel solver execution. This will ensure future maintainability and ease of onboarding for new contributors.

5. Implementation Plan

Current Implementation Summary

In the current implementation of the SPF, symbolic constraints from the path condition are solved using a single selected constraint solver. The choice of the solver is made based on the *symbolic.dp* configuration (e.g., `z3`, `choco`, `cvc3`, etc.), and the corresponding Problem General subclass (like `ProblemZ3`, `ProblemChoco`, etc.) is instantiated accordingly.

A single solver is selected and used throughout the symbolic execution of a program. This solver handles all path conditions encountered during execution. If it fails to solve or throw an exception, fallback mechanisms are usually used. However, parallel solver strategies are not currently employed, and solving halts as soon as one solver fails or succeeds.



```
String[] dp = SymbolicInstructionFactory.dp;
if (dp == null) { // default: use choco
    pb = new ProblemChoco();
} else if (dp[0].equalsIgnoreCase( anotherString: "choco")) {
    pb = new ProblemChoco();
    // } else if(dp[0].equalsIgnoreCase("choco2")){
    // pb = new ProblemChoco2();
} else if (dp[0].equalsIgnoreCase( anotherString: "coral")) {
    pb = new ProblemCoral();
} else if (dp[0].equalsIgnoreCase( anotherString: "iasolver")) {
    pb = new ProblemIAsolver();
} else if (dp[0].equalsIgnoreCase( anotherString: "cvc3")) {
    pb = new ProblemCVC3();
} else if (dp[0].equalsIgnoreCase( anotherString: "cvc3bitvec")) {
    pb = new ProblemCVC3BitVector();
} else if (dp[0].equalsIgnoreCase( anotherString: "yices")) {
    pb = new ProblemYices();
} else if (dp[0].equalsIgnoreCase( anotherString: "z3")) {
    pb = new ProblemZ3();
} else if (dp[0].equalsIgnoreCase( anotherString: "z3inc")) {
    pb = new ProblemZ3Incremental();
} else if (dp[0].equalsIgnoreCase( anotherString: "z3bitvectorinc")
    pb = new ProblemZ3BitVectorIncremental();
} else if (dp[0].equalsIgnoreCase( anotherString: "debug")) {
    pb = new DebugSolvers(pc);
} else if (dp[0].equalsIgnoreCase( anotherString: "compare")) {
    pb = new ProblemCompare(pc,  scg: this);
```

Step by Step Implementation Plan

Note: This plan is a high level overview and more of a draft, it's expected to change with further understanding of the code base.

Phase 1: Exploration, Solver Study & Abstractions

- Deep dive into the solver abstraction (*ProblemGeneral* and its subclasses).
- Identify exact points in the codebase where constraint solving is triggered.
- Run comparative experiments with *Z3 and CVC5* on a variety of SMT-LIB queries.
- Focus on how constraints are encoded and how models are returned.
- Document syntax differences, output formats, and solver flags required (e.g., *`--produce-models`* in CVC5).
- Design a *SolverPortfolioManager* class to coordinate parallel execution.

Phase 2: Parallel Execution With Homogenous Solvers (Z3 Only)

- Begin with support for multiple parallel instances of Z3.
- Focus exclusively on parallel execution and thread management.
- Validate core functionality like early SAT detection and interrupting other solver threads.
- Implement *`SolverPortfolioManager`* that executes solvers in parallel threads.

- Use `ExecutorService` or similar Java concurrency utilities to manage solver threads.

Phase 3: Heterogeneous Portfolio Support (Z3 + CVC5)

- Generalize the portfolio system to support different solvers.
- Add support for CVC5.

Phase 4: Configuration & Extensibility

- Extend SPF's configuration file to accept a portfolio of solvers (e.g., *symbolic.dp.portfolio=z3,z3 + cvc5*).
- Allow configuration of timeouts, priorities, and solver-specific options.
- Add support for fallback to traditional single-solver mode for backward compatibility.

Phase 5: Integration with ***sv-comp*** Branch.

- Merge with the *sv-comp* branch of SPF.
- Ensure symbolic execution tests and `sv-comp` benchmarks pass with portfolio mode enabled.
- Handle edge cases: timeouts, memory exceptions, contradictory results.

Phase 6: Testing and Documentation

- Add unit and integration tests for the portfolio manager.
- Add developer documentation for the portfolio solver module.
- Contribute to or update existing README or wiki pages to reflect the changes made during the project.

6. Timeline and Work Planning

On average I expect to be able to put 20 - 40 hours per week towards GSoC. The workload is distributed over 11 - 12 weeks for about 350 hours of work.

Milestone	Goal	Phase (s)	Week (s)
1.	Study existing solver integration in SPF	1	1
2.	Design an architecture for parallel solver execution	1	1
3.	Implement the Proposed Architecture	2,3,4	8
4.	Test and evaluate the implementation	5	1
5.	Document the Design and Implementation	6	1

Week 1

Goal: Analyze how different solvers (e.g., Z3, CVC) are currently integrated within the SPF codebase and run comparative experiments with *Z3 and CVC5* on a variety of SMT-LIB queries.

Time Period: (June 2 - June 8 | 40 h)

- This week will be used to analyze and understand the codebase and integration of different solvers thoroughly.
- Additionally, this period will serve as a community bonding phase. I will actively engage with mentors and the SPF community to clarify doubts, discuss design considerations, and ensure alignment with the broader project goals.

Week 2

Goal: Design an architecture for parallel solver execution.

Time Period: (June 9 - June 22 | 40 h)

- This week will be utilized to design the architecture for parallel solver execution with constant feedback from mentors and the community.
- This time period will provide a solid foundation for the actual coding phase.

Week 3 - 10

Goal: Implement the Proposed Architecture.

Time Period: (June 23 - August 11 | 220 h)

- This 8-week period will be dedicated to implementing the architecture design.
- Development will follow an incremental approach, with continuous testing and integration. Progress will be regularly merged into the `sv-comp` branch to maintain version control and allow early feedback from mentors and the community.

Week 11

Goal: Test and evaluate the implementation.

Time Period: (August 12 - August 18 | 25 h)

- This time period will focus on thoroughly testing the parallel solver execution system to ensure its correctness, performance, and reliability.
- This time period will also be used to identify, debug, and resolve any unexpected issues that arise during testing

Week 12

Goal: Document the Design and Implementation.

Time Period: (August 19 - August 25 | 25 h)

- This final phase will be dedicated to creating comprehensive documentation for the parallel solver execution feature. The documentation will serve as a reference for both users and future contributors, ensuring the work is understandable, maintainable, and easy to build upon.

7. Plans for future improvement and involvement

After completing GSoC, I intend to remain actively involved with the Java PathFinder (JPF) team and continue contributing to the project. I will be fully committed to maintaining and improving the work done during GSoC, and I'll remain available to address any bugs or issues that may arise from my implementation. I also look forward to contributing to other areas of the project and supporting new contributors as they onboard.

8. Bibliography

- https://smt-lib.org/papers/smt-lib-reference-v2.6-r2021-05-12.pdf
- https://www.philipzucker.com/z3-rise4fun/guide.html
- https://github.com/cvc5/cvc5
- https://de-engineer.github.io/SMT-Solvers/