POLITECNICO
MILANO 1863

Java **Path finder**

30 April 2025

Domenico Popolizio, Nicola Zarbo, Enrico Vento, Federico Valentino

POLITECNICO
MILANO 1863

What's **JPF**

A VM Supporting Model Checking

Operates on Java bytecode

It handles instructions generated by a standard Java compiler. It can test an entire system or small portions of code, through usage of Java Annotations

Explores all possible Execution Paths

JPF identifies execution choices, or points in which the execution of the system under test could diverge

A tool for Mission Critical Applications

In some scenarios, we cannot afford to learn about defects after deployment. As of now, there are defects that can be only identified through JPF.

Timeline

1999

NEED
Model checking
real java systems

2000

SPIN FRONTEND
Java to Promela
NASA

2003

JPF VM
NASA

2005

EXTENSIBILITY

2008 and on

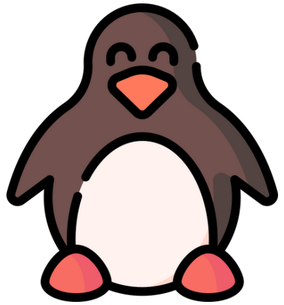
OPEN SOURCED

GOOGLE SUMMER CODE
And Further
development

Key Features

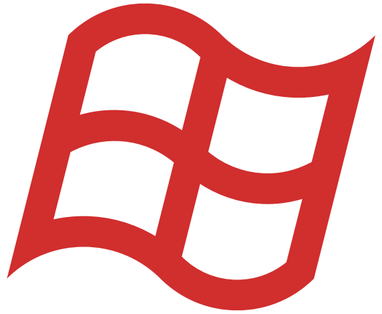
- Explicit State Model Checking
- Symbolic Execution
- State Matching
- Backtracking
- Partial Order Reduction
- Extensibility

Install & **Config**



Linux

Fairly quick and easy process, the better alternative



Windows

We recommend using WSL

Preliminary Steps

Download JDK

Make sure to download the correct version of the Java Development Kit. As of now, JDK 11 is the right version, but since it is subject to change, refer to the official JPF Github page

Download Git

Verify that Git is installed on your machine before proceeding to the installation steps

Set up Environment Variables

Ensure that JAVA_HOME has been created as a new environment variable, and that Git has been added to the Path environment variable

Installation Steps

1. Clone the JPF GitHub Repository

Navigate to the desired folder, and then input the following command:

git clone https://github.com/javapathfinder/jpf-core.git

2. Run the Gradle script

Navigate to the jpf-core subdirectory and run the gradlew script in a shell:

./gradlew

3. Set up the Environment Variables

You must now add a new environment variable, JPF_HOME, pointing to the jpf-core directory, and adding jpf as a system wide command:

- Open .bashrc in nano to edit it
- Add export JPF_HOME=/<path_to_jpf>/jpf
- Add export PATH="$PATH:/<path_to_jpf>/jpf-core/bin"
- Apply changes with source ~/.bashrc

Installation Steps

4. Create the site.properties file

- Navigate to user/home and create the hidden directory .jpf
- Within this directory, create a file names site.properties
- Assuming the jpf-core folder is a subdirectory of user/home/jpf, edit the file to include the following content:

```
# JPF site configuration
jpf-core=${user.home}/jpf/jpf-core
extensions=${jpf-core}
```

Notice - RAM Limitations

JPF is not a lightweight system. Some machines, especially embedded systems, may struggle to run it. Ensure that your machine has more RAM than that required to host a Java Virtual Machine

POLITECNICO
MILANO 1863

Using **JPF**

Program run by JPF

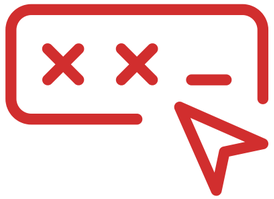
```
24    static class Philosopher extends Thread {
25
26      Fork left;
27      Fork right;
28
29      public Philosopher(Fork left, Fork right) {
30        this.left = left;
31        this.right = right;
32        //start();
33      }
34
35      @Override
36    public void run() {
37        // think!
38        synchronized (left) {
39          synchronized (right) {
```

Being JPF itself a VM it runs java code.
This is crazy because it brings the power of model checking to a real programming language and real software!

Explored domains

Scheduling Sequences

Concurrent application are testable because we can control the scheduler.

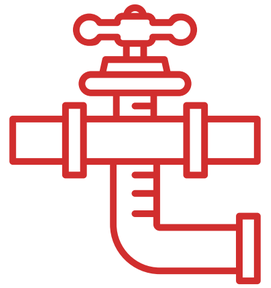


Input Variation

Similar to systematic testing, it allows, through heuristics to test all the significant input spectrum.

Environment Changes

JPF can simulate environment events, enabling model checking for application like IoT or Web

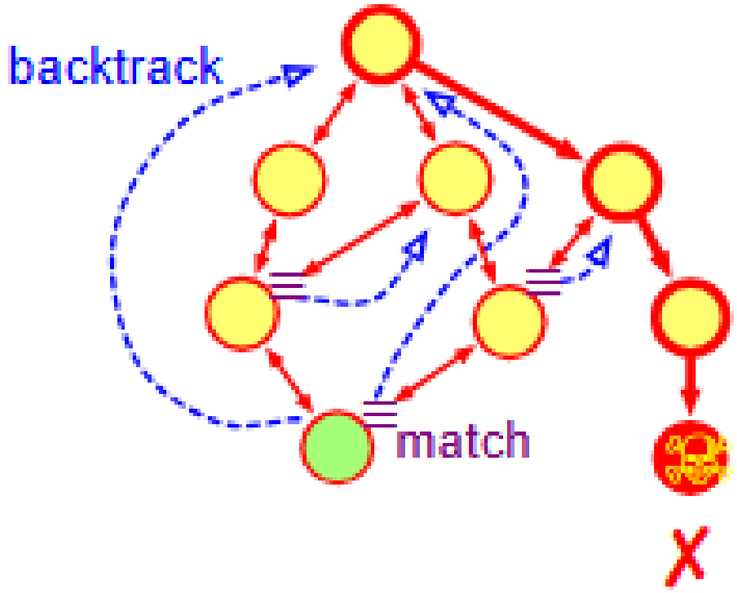


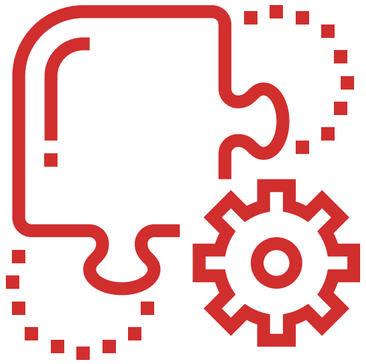
Control Flow

It automatically check all the branches of your code for running hidden parts of it.

How JPF does the trick

JPF Performs model checking on the code execution, reaching all possible states.





Basic JPF Structure

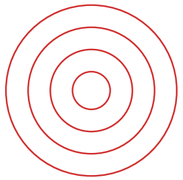
System under testing

Model Java Interface

Java Path finder VM

Java virtual Machine

Java Native Interface



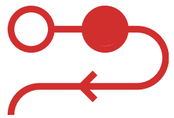
Operating System

Program types

JPF is a very flexible tool and can be used with different types of program

JPF Dependent program

The basic use is testing specific models and algorithm implemented in Java.

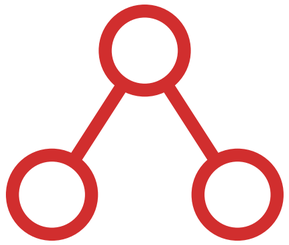


JPF Enabled programs

Programs that have only some parts that must be model checked

JPF Unaware programs

Finally JPF can model check entire java programs that have been separately designed!



JPF Dependent

What

Testing specific models and algorithm implemented in Java, written for being tested with JPF.
Also, it takes less resources than checking real programs

Why

JPF Allow you to have control over scheduling, finding race conditions

How

By being a VM, JPF can check any algorithm.
Moreover there are specific JPF Annotations and methods for implementing specific checks.

JPF Unaware

What

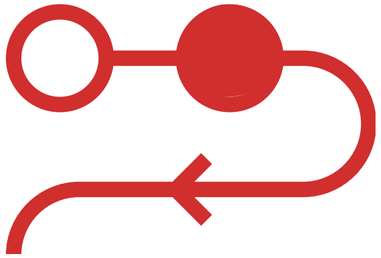
Normal java code, of any type.

Why

To test in a deeper way than any other testing system.

How

JPF Being a VM can run any program, and model check them.
Notice that it will be slower than running on JVM, and that big program may lead to state explosion.



JPF Enabled

What

Problem where only a part of the code is going to be model checked

Why

Model checking is resource expensive, testing entire programs may be not ideal.
You can test only part of a program.

How

Through annotation (e.g. @NonNull and many others) which JPF Recognizes and checks.

Running JPF

JPF run every program similarly to how you run any java program

Invoking JPF

JPF is invoked through the jpf command

```
~$ jpf <...>
```

The alternative syntax, without storing jpf into the path is

```
~$ java -jar <jpf-core-dir>/build/RunJPF.jar <...>
```

Running JPF

JPF run every program similarly to how you run any java program

Command line

The most basic way of running jpf

```
~$ jpf <main class>
```

If the class is outside the configured project class path it can be specified

```
~$ jpf +classpath=<class-path> <main class>
```

Notice that other properties are specifiable, but this is not the correct place to do it

Running JPF

JPF run every program similarly to how you run any java program

Example

```
~$ ls
 HelloWorld.java      HelloWorld.class
```

```
~$ jpf  +classpath=.  HelloWorld
```

.jpf
Files to run it

Although you can run JPF like this, the correct way is different.
A . jpf file must be created

Create a .JPF File

```
~$ nano <name>.jpf

target = <main class>
classpath = <class-path>
```

The classpath can be absolute or relative to the .jpf file location

Execution

```
~$ jpf <JPF file>
```

.jpf
Files to run it

Although you can run JPF like this, the correct way is different.
A . jpf file must be created

Example

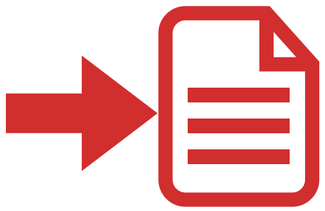
Only the first time

```
~$ nano TestingHelloWorld.jpf

 target = HelloWorld
 classpath = ~/projects/difficulty/hard/
```

Every time

```
~$ jpf TestingHelloWorld.jpf
```



Outputs

System Under Testing

This output come from the system under testing, and depends on what that systems prints out.

Reports

This is the core of JPF output, and practically essential to make the model checking significative.

Logging

Won't go into details: it is possible to log with standard java loggers, the internal mechanism of JPF.
There are various level of details.

S.U.T. Output

This output come from the system under testing, and depends on what that systems prints out.
Notice that it print everything even when backtracking!

Code

```
public class Rand {
    public static void main (String[] args) {
        Random random = new Random(42);

        int a = random.nextInt(2);
        System.out.println("a=" + a);

        int b = random.nextInt(3);
        System.out.println("  b=" + b);

        int c = a/(b+a -2);
        System.out.println("    c=" + c);
    }
}
```

Java Output

```
$ java Rand.java
a = 1
  b = 0
    c = -1

$ java Rand.java
a = 0
  b = 1
    c = 0

$ java Rand.java
 ....
```

S.U.T. Output

This output come from the system under testing, and depends on what that systems prints out.
Notice that it print everything even when backtracking!

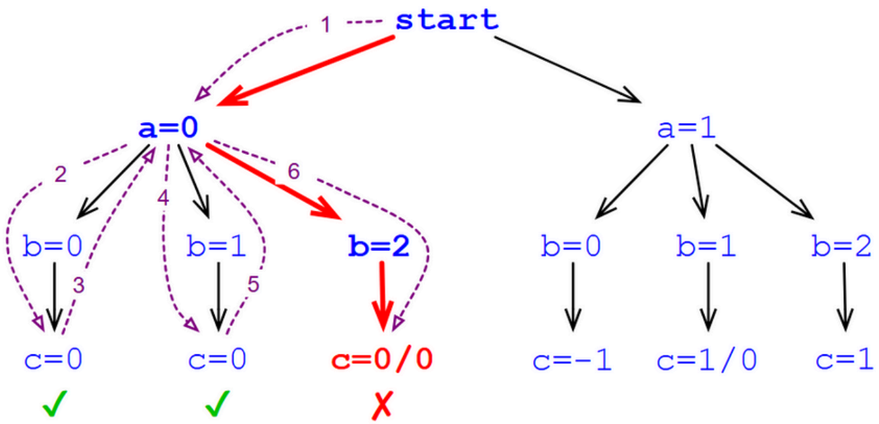
Code

```
public class Rand {
    public static void main (String[] args) {
        Random random = new Random(42);

        int a = random.nextInt(2);
        System.out.println("a=" + a);

        int b = random.nextInt(3);
        System.out.println("  b=" + b);

        int c = a/(b+a -2);
        System.out.println("    c=" + c);
    }
}
```



JPF "Tree" Output

```
$ jpf Rand.jpf
a=0
  b=0
    c=0
  b=1
    c=0
  b=2
```

Error

S.U.T. Output

This output come from the system under testing, and depends on what that systems prints out.
Notice that it print everything even when backtracking!

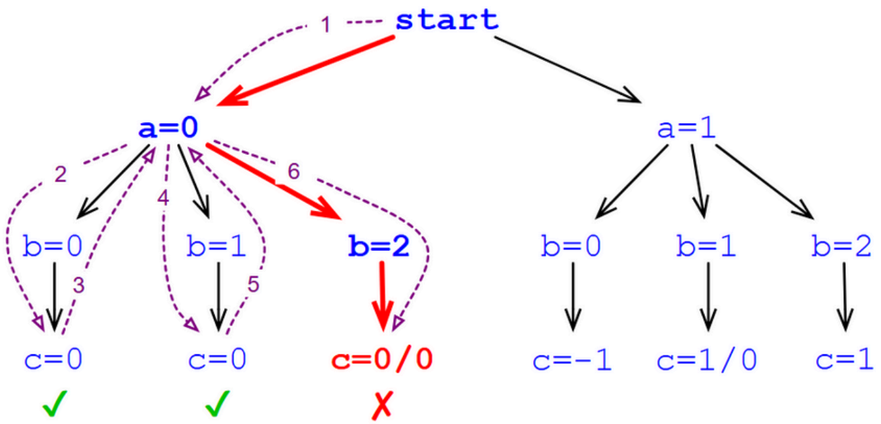
Code

```
public class Rand {
    public static void main (String[] args) {
        Random random = new Random(42);

        int a = random.nextInt(2);
        System.out.println("a=" + a);

        int b = random.nextInt(3);
        System.out.println("  b=" + b);

        int c = a/(b+a -2);
        System.out.println("    c=" + c);
    }
}
```



JPF "Path" Output

```
$ jpf Rand.jpf
a=0
  b=2

Error
```

The path output stores the output and print only the output of the path that create the error!

Report Output

This is the core of JPF output, and practically essential to make the model checking significative.

Supports: Text, XML, API Calls

Target: IDE

Output phases

Start

Configurations Date-Time Versioning
Platform SUT Class User

Transition

Statistics

Probe

Statistics Probe interval

Property Violation

Error Snapshot
Output Statistics
Trace (config, code, location, method, source steps)

Constraints

Constraint Snapshot
Statistics Output Trace

Finished

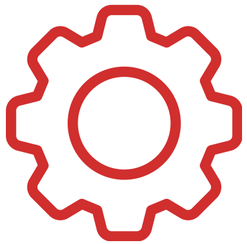
Statistics

Example report

Start (versioning, SUT), Transition (Statistics), Property Violation (error, trace)

```
JavaPathFinder core system v8.0 ....
================ system under test
HelloWorld.main()
==================== search started: 30/02/2025
Hello World!
==================== statistics
elapsed time: 00:00:00
states:           new=1, visited=1, backtracked=0, end=0
search:           maxDepth=3, constraints=0
choice generators, heap, instructions,
max memory, loaded code

Now I am going to perform 3/0 =
==================== error 1
gov.nasa.jpf.vm.NoUncaughtExceptionProperty
java.lang.ArithmeticException: division by zero
  at UncaughtException.main (UncaughtEception.java:3)
====================trace #1
------------------------ transition #0 thread: 0
HelloWorld.java:4
System.out.println("Hello World!");
HelloWorld.java:5
if(true) {
------------------------ transition #1 thread: 0
HelloWorld.java:5
if(true) {
HelloWorld.java:6
   System.out.print(3/0);
```



JPF and JUnit Tests

We can run JPF verification directly inside a standard JUnit test class

The Test Class

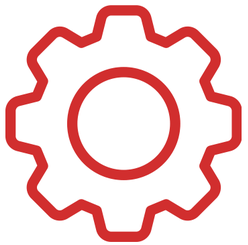
JPF already uses JUnit as internal testing infrastructure. We can exploit this to write our own custom test classes

```
import gov.nasa.jpf.util.test.JPFTestSuite;
import org.junit.Test;

public class MyTest extends TestJPF {

  @Test
  public void testSomeFunction() {
    if (verifyNoPropertyViolation(jpfOptions))
    {
      someFuntction();
    }
  }
```

The function within the class will be tested by JPF



JPF and JUnit Tests

We can run these test classes simply by command line

```
bin/test <test-class> [<test-method>]
```

Which in turn call the executable java -jar tools/RunTest.jar

We can also specify temporary JPF properties, for example, using a specific listener

```
bin/test+test.listener=.listener.ExecTracker gov.nasa.jpf.test.mc.basic.AttrsTest
```

jpf.run()

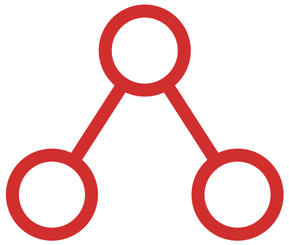
Launch From Within a Program

JPF can be launched directly inside a JAVA program, and it's output can be used in the program flow itself

```
public class MyJPFLauncher {
 ...
 public static void main(String[] args){
  ..
  try {
   Config conf = JPF.createConfig(args);
   conf.setProperty("my.property", "whatever");
   MyListener myListener = ...
   JPF jpf = new JPF(conf);
   jpf.addListener(myListener);
   jpf.run();
   if (jpf.foundErrors()){
      }
  } catch (JPFConfigException cx){

  } catch (JPFException jx){

  }
...
```



Coding JPF Dependent

Annotations

@JPFConfig(...) allow to add per-method configurations
@FilteredField allow to specify variable as non relevant for states
@Nonnull @NonShared @Immutable @Requires ....

Verify API

Enable to verify explicit behavior

Data Choice Generators
boolean cond = Verify.getBoolean()
int d = Verify.getInt(0, 100); ....

Search Pruning
Verify.ignoreIf(condition);

State Annotation
Verify.interesting(condition)
if(condition) { Verify.setAnnotation("label"); }

Atomicity Control
Verify.beingAtomic() ---- 1 transition only -----> Verify.endAtomic();

JPF Properties

JPF is not a monolithic solution, and should be configured to better fit our needs. Its properties are the main configuration mechanism

What are JPF Properties

Each property is a key - value statement, presented int he following format:

key = value

For example, consider the file presented in slide 23, reported here:

```
~$ nano <name>.jpf

 target = <main class>
 classpath = <class-path>
```

These are both JPF properties

Property Types

There are three different property files, each containing properties of the corresponding type. These files are referenced into a single dictionary object, the Config file (jpf-core/gov/nasa/jpf)

Site Properties

Comprised of properties in the form of <name> = <directory>.
A peculiar key is extensions, which can have as values a comma separated series of jpf-core modules and user made extensions.

It is mandatory to have ${jpf-core} listed in the extension key

Project Properties

Each JPF module, either core or extension, contains a jpf.properties file. The project property file defines the JPF specific paths that need to be set for the module to work properly. The relevant keys are:

1. native_classpath
2. classpath
3. test_classpath
4. sourcepath

Application Properties

IIdentify the path of the system to be tested and specific properties on how to run the test

Application Properties

First Level of properties. Used to identify the system under test and to define how to run the test itself

Location

User Defined

Uses

Identification of the SUT path

Personalizing testing options

Mandatory Properties

```
target = MyClass
classpath = <MyClass_path>
```

Other Properties

For examples, randomizing the order with which JPF chooses which branch to take

```
cg.enumerate_random = true
cg.randomize_choices = VAR_SEED
```

Application Properties

Application properties can also be set by command line directly as properties of the jpf executable

Uses

Overriding properties in the application properties file

Convenient for quickly switching between useful property values

Syntax

```
cg.enumerate_random += true
```

Application Properties

Note

To gain access to JPF application properties that regard randomization, JPF requires access to the java.utils.Random library.

However, this seems to be a problem if using the version of JPF currently featured on the official GitHub page.

Solution

Go to the jpf-core/bin directory, and open the jpf script with nano. Paste this command at the end of the script

```
java \
  --add-exports java.base/jdk.internal.misc=ALL-UNNAMED \
  --add-opens java.base/jdk.internal.misc=ALL-UNNAMED \
   -jar $JPF_HOME/build/RunJPF.jar "$@"
```

Project Properties

Third level of properties.
Used to configure but also build any project

Location

jpf-core root directory

Uses

Configuration of a project run by a specific jpf version

Building a specific jpf version

Properties

jpf-core.native_classpath = [jpf path]

jpf-core.classpath = [bytecode to be model checked]

jpf-core.sourcepath = [sources to be model checked]

Others

Other default setting for lower levels

Site Properties

Fourth level of properties, basically at O.S. Level

Location

User home/.jpf/site.properties

```
System.out.println(System.getProperty("user.home"));
```

Uses

Configuration valid for the user executing jpf in the os

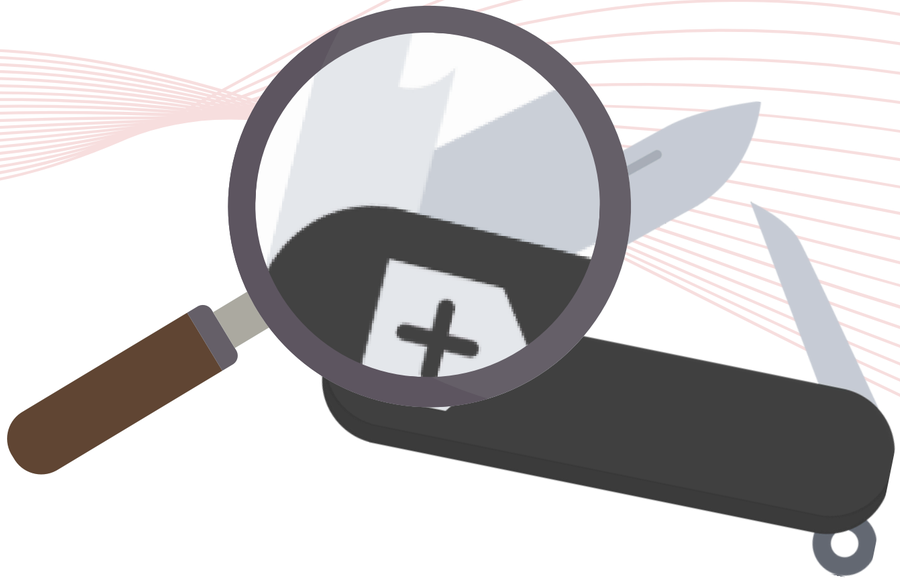
Extensions

Properties

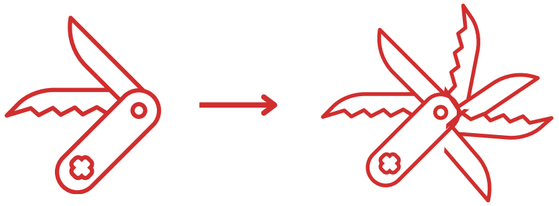
```
jpf-core = [path to jpf-core]
jpf-extension-name = [path to another extension]
...
extensions = ${jpf-core}, ${jpf-extension-name}, ...
```

Others

```
Other default setting for lower levels
```



JPF **Exposed**



Unlocking Full Power

JPF is Open Souirce and highly **extensible**.
To extend it we need to have a look under the hood.

Off-the-shell extensions

There are lot of extensions that fits wide range of needs

JPF Under the hood

Checking what's under the hood enable us to unlock the full potential

(Custom) Properties verification

JPF become **not only for java errors!**
We might check specific (custom?) properties!

Some extensions

Virtually infinite, here there are a few that might be interesting, even if they are out of the scope of understanding jpf itself.

jpf-core

Standard jpf

jpf-probabilistic

Jpf that consider the markov process underneath randomized algorithms.

jpf-symbc

Jpf for symbolic execution

jpf-aprop

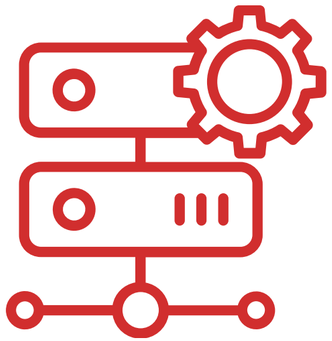
Jpf with annotation based program properties

jpf-nhandler

Support running part of the code on the native JVM

BenchExec

Execute benchmark and resource management



Understanding JPF Design

JPF is designed with two principal components in mind

The Virtual Machine

Holds classes and methods responsible for generating the state representations, which can be:

- **Checked for Equality**
- **Queried**
- **Stored**
- **Restored**

The Search Component

Decides which state to visit next, based on different properties to be checked and/or heuristics

The VM Package

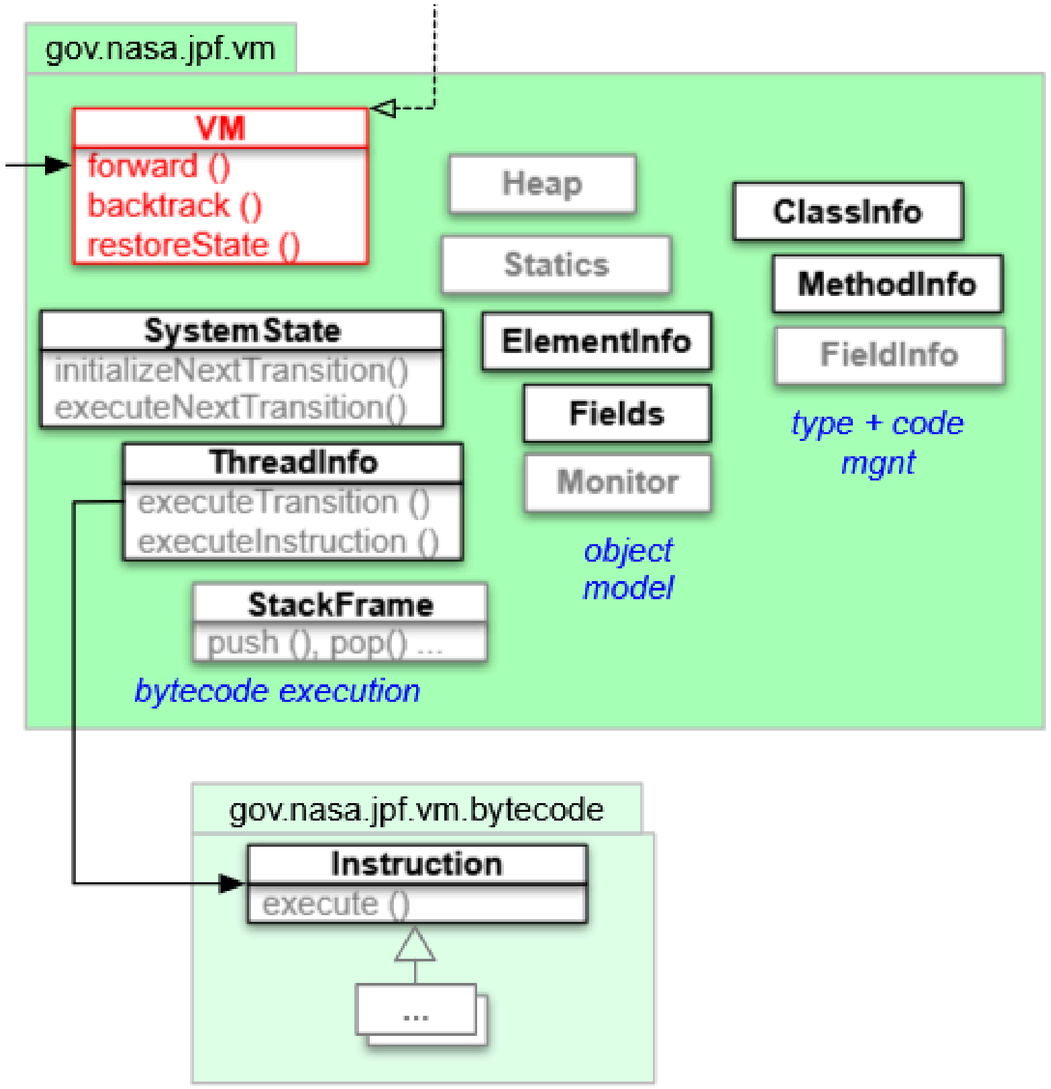
Functionalities

ClassInfo encapsulates and manages classes under test

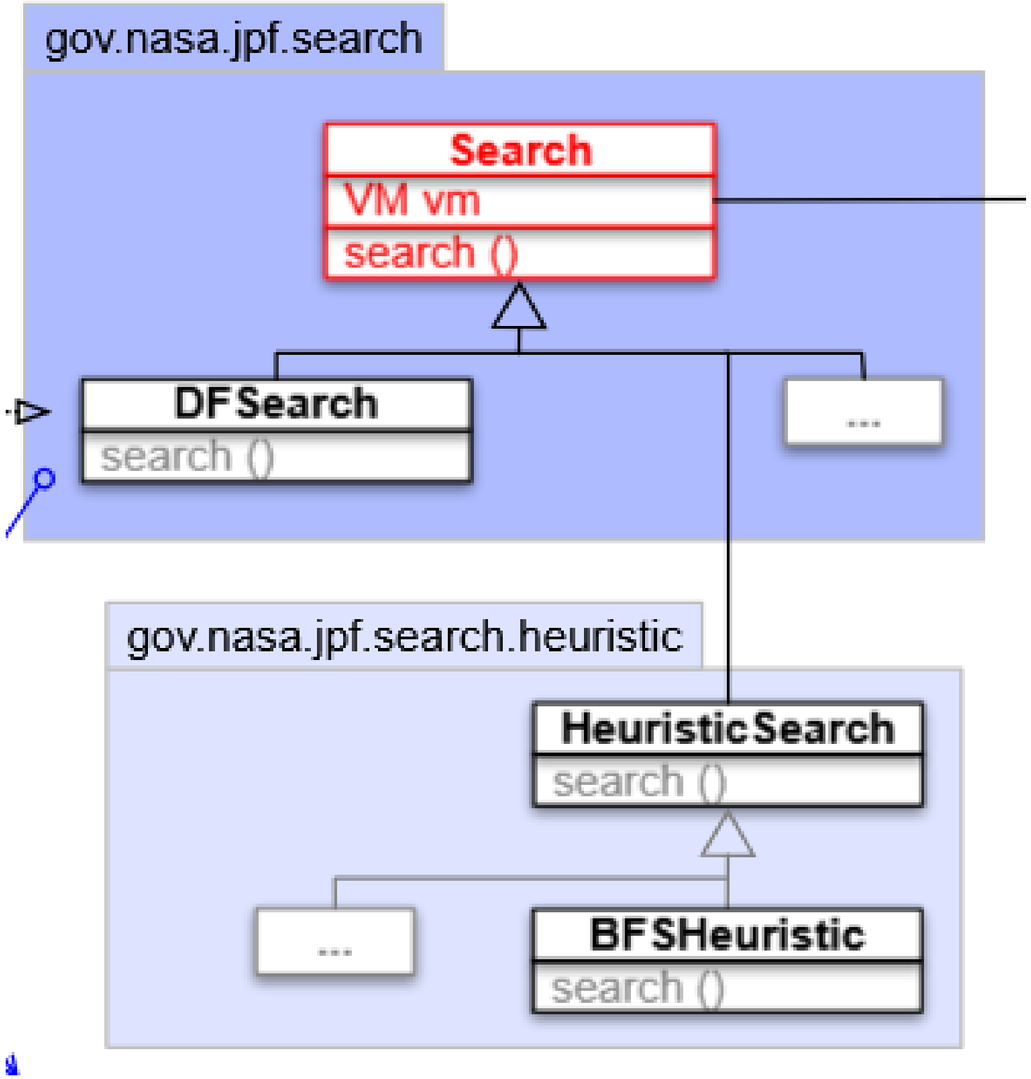
Fields objects store into integer arrays all of the object data

SystemState and ThreadInfo handle the bytecode execution

forward(), **backward()** and **restoreState()** are methods invoked by the Search Component to either, repectively, progress to the next state, restore the state on top of the backtrack stack, or restore an arbitrary state of choice



The Search Package



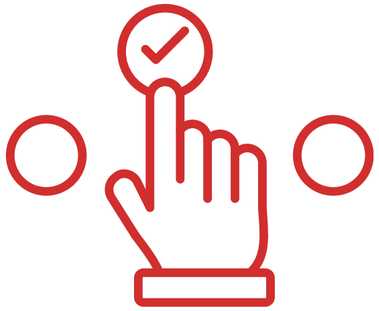
Functionalities

search() is the VM driver, selecting either forward(), backtrack() or restoreState()

DFSearch is the default depth first search streategy for the state space

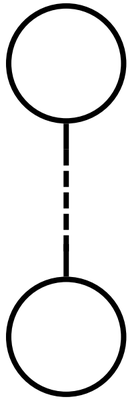
HeuristicSearch and BFSHeuristicSearch configurable heuristics to drive the search towards state that show certain desired characteristics

This component can be extended with new search strategies and heuristics



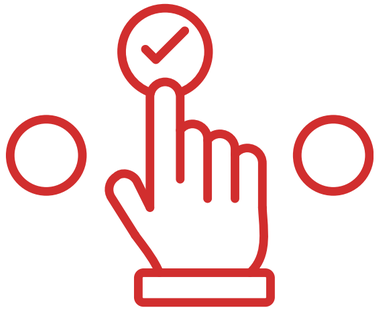
Choice Generator

Program faces non-determinism problems.
Choice Generators Handle them.



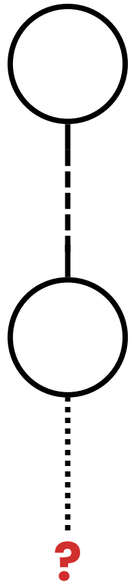
State

Memory status.
Path that lead to it.



Choice Generator

Program faces non-determinism problems.
Choice Generators Handle them.

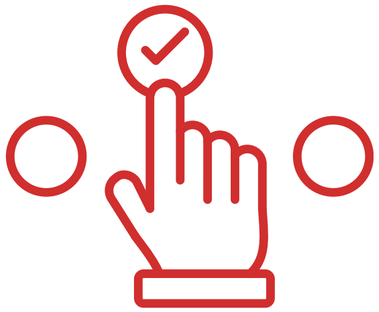


Non-Determinism

Scheduling choices
Thread and shared objects

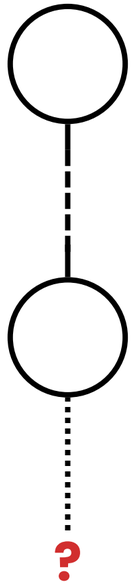
Data choices
Testing different variable values

Control choices
Checking multiple branches of conditions



Choice Generator

Program faces non-determinism problems.
Choice Generators Handle them.

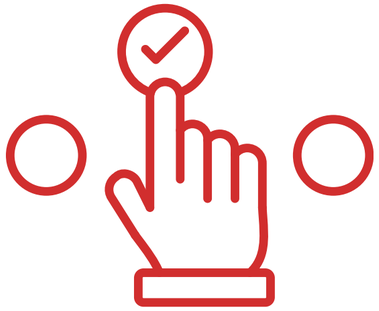


Solutions

Decouple the situation.
Different cases of non-determinism solved by different mechanisms.

Pre-Configure solution for basic cases
Scheduling, Boolean, naive control choices...

Non basic cases
Domain specific cases
Parametrization
External interface working at runtime (JPF Properties)

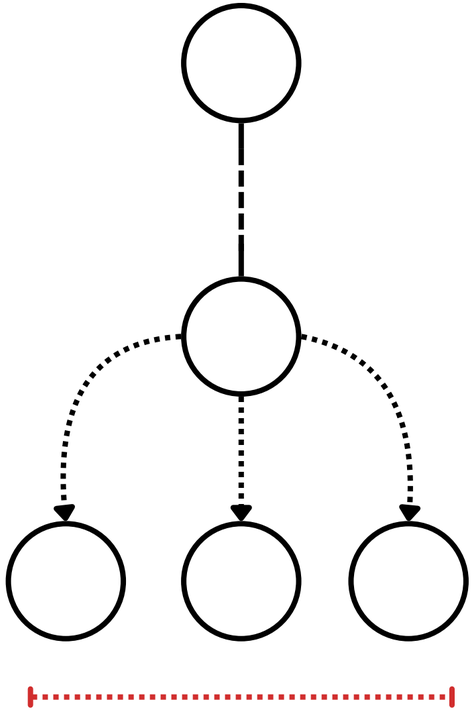


Choice Generator

Program faces non-determinism problems.
Choice Generators Handle them.

Possible Choices

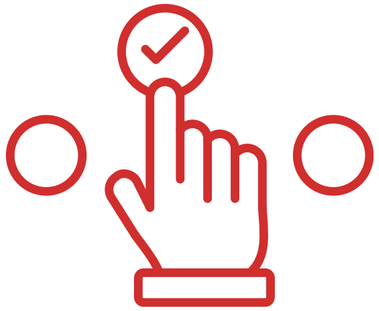
Driven by non-det.
Are possible new states based no how the non determinism is solved.



ChoiceGenerator

Generates possible choices.

This is an extensible component (through heuristics).



Choice Generator

Program faces non-determinism problems.
Choice Generators Handle them.

JPF Unaware

Managed automatically by JPF

JPF Dependant

We can detail choice generation.

Basic types

E.g. Use the JPF Verify API Basic Types

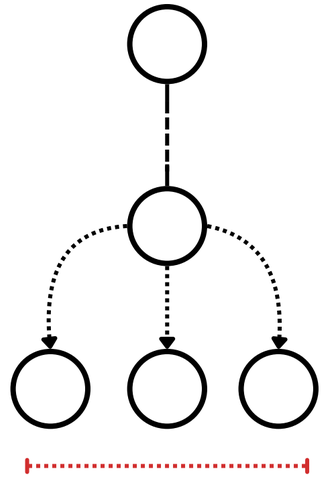
```
boolean managed = Verify.getBoolean();
```

Threshold values

E.g. Use the JPF Verify API Parametrization

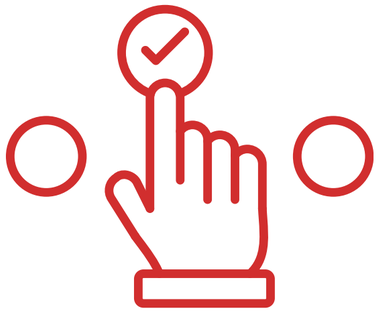
```
double speed = Verify.getDouble("myheur");
```

```
myheur.class = gov.nasa.jpf.jvm.choice.DoubleThresholdGenerator
myheur.threshold = 13500
myheur.delta = 500
```



ChoiceGenerator

Generates possible choices.



Choice Generator

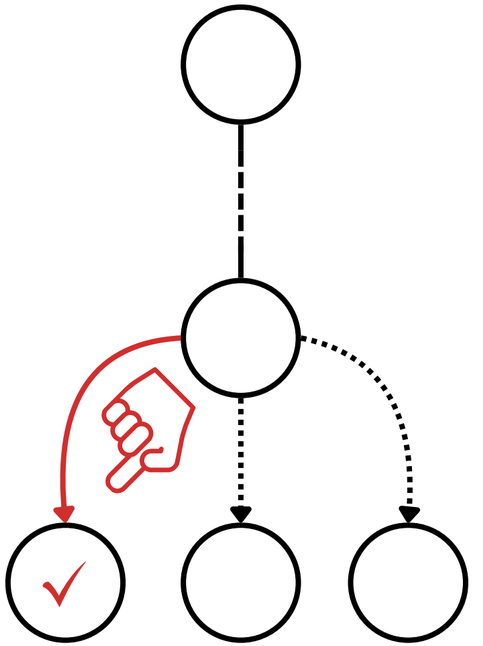
Program faces non-determinism problems.
Choice Generators Handle them.

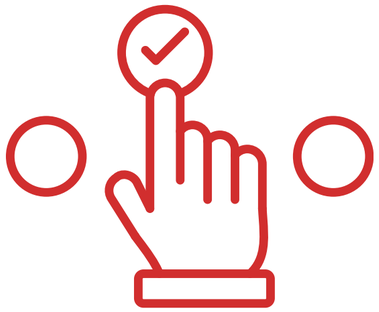
Choice

Driven by non-det.
Starts transition to new state.

Transition

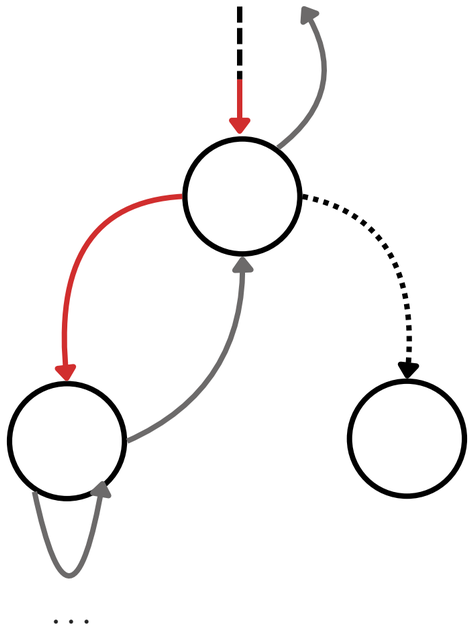
Moving from one state to another.





Choice Generator

Program faces non-determinism problems.
Choice Generators Handle them.



Internally

Each state store its own choice generator.

System state set with a new choice generator when find non determinism.

Execute the code from the new state.

When finishing all the lower level choices, the systems rollback and try a different choice.

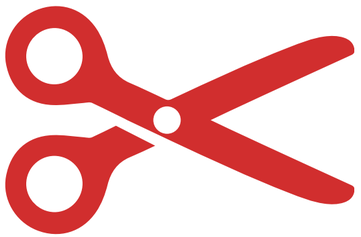
Objects

SystemState
initNextTransition()
setNextChoiceGenerator()

ChoiceGenerator
hasMoreChoices()
advace()

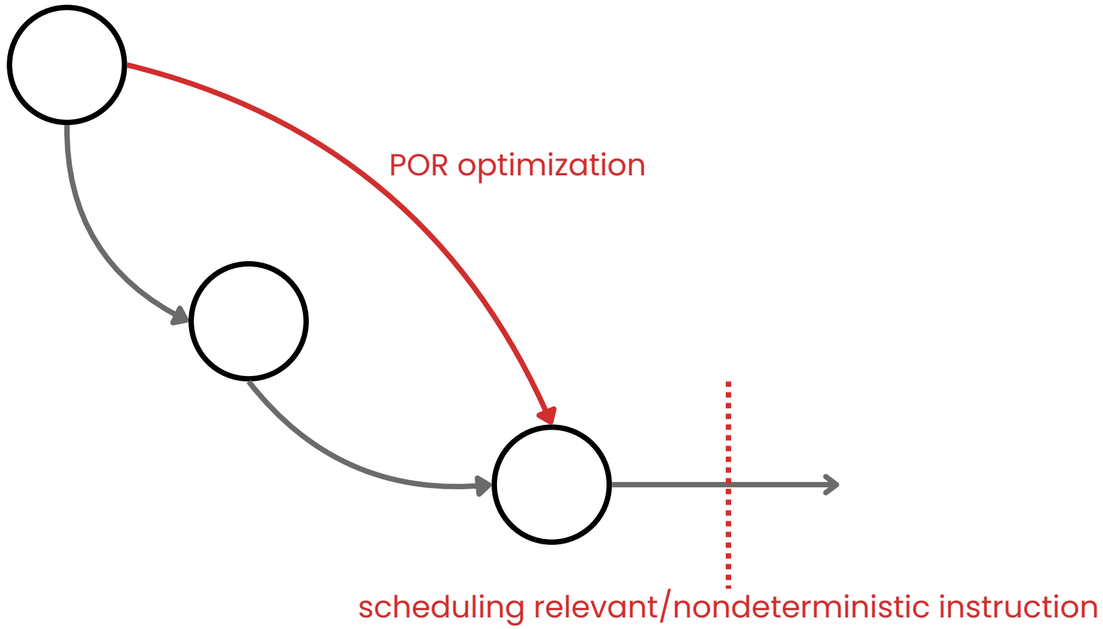
ThreadInfo
ExecuteNextTransition()
ExecuteInstruction()

Instruction
execute()



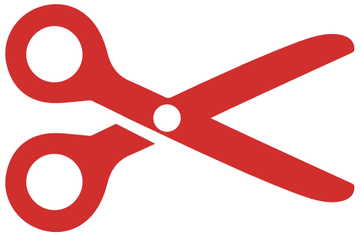
Partial Order Reduction

JPF implements on the fly partial order reduction to achieve a reduction of 70% or more of state spaces



Sequence of instructions within a single thread that are not judged as scheduling relevant or nondeterministic are treated as a single transition

Partial Order Reduction can be enabled by setting vm.por = true in the application property file



Partial Order Reduction

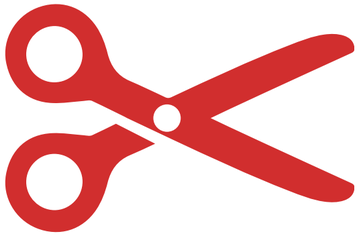
JPF implements on the fly partial order reduction to achieve a reduction of 70% or more of state spaces

Deciding scheduling relevance

An instruction is said to be scheduling relevant if it involves access to shared resources or thread synchronization. These can be instructions like get or set for shared resources, or monitorEnter/Exit for direct synchronization, or certain thread calls, like wait(), sleep(), start()

Deciding non determinism

An instruction is said to be nondeterministic if it involves unpredictable behaviors, such as reading a value from input, or generating a random number

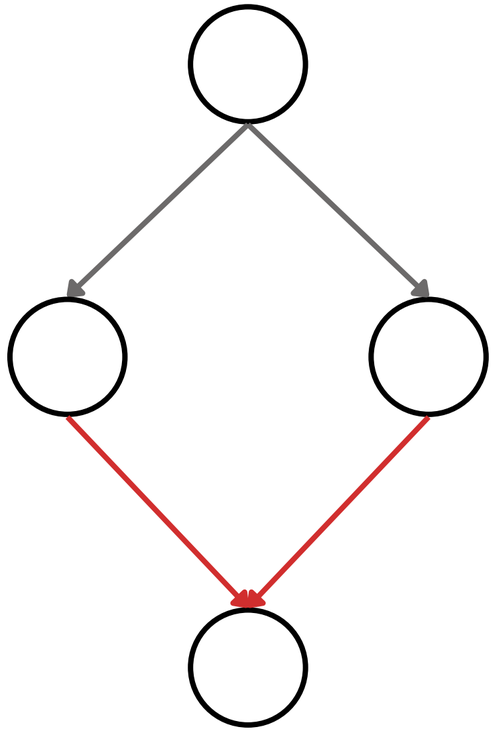


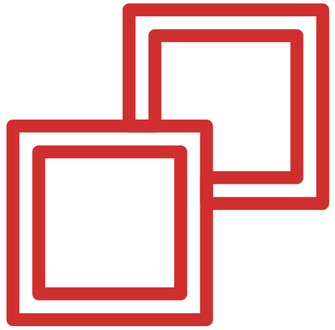
Partial Order Reduction

JPF implements on the fly partial order reduction to achieve a reduction of 70% or more of state spaces

Limitations of Partial Order Reduction

JPF cannot handle diamond cases with partial order reduction, that is, when two thread join on a single state, as it doesn't perform lookahead analysis





Attributes

JPF extends the JVM with a storage mechanism that links variables, fields and even entire objects to configurable meta-data objects, called attributes

Association through API

Program elements can be associated to attributes with the APIs given by:

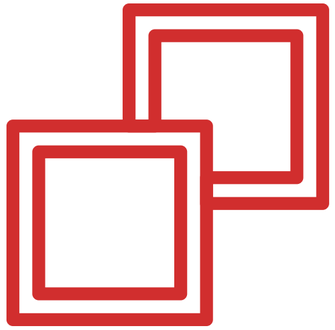
- gov.nasa.jpf.v.Fields for instance and static fields
- gov.nasa.jpf.vm.StackFrame for stack slots (local operands and variables)

Propagation

Each time a program element gets updated, JPF propagates the change to the corresponding attribute, to keet everything in sync

Backtracking Behavior

When backtracking, JPF only restores the attribute references, but not the values



Attributes

JPF extends the JVM with a storage mechanism that links variables, fields and even entire objects to configurable meta-data objects, called attributes

Usage

Attributes can be used to assign symbolic values to certain variables or objects, enabling symbolic and concolyc execution.

Limitations

Attribute generation and upkeep leads to a not indifferent slowdown in execution

Moreover, it is not possible to assign more than one attribute type to a single field or slot

Listeners

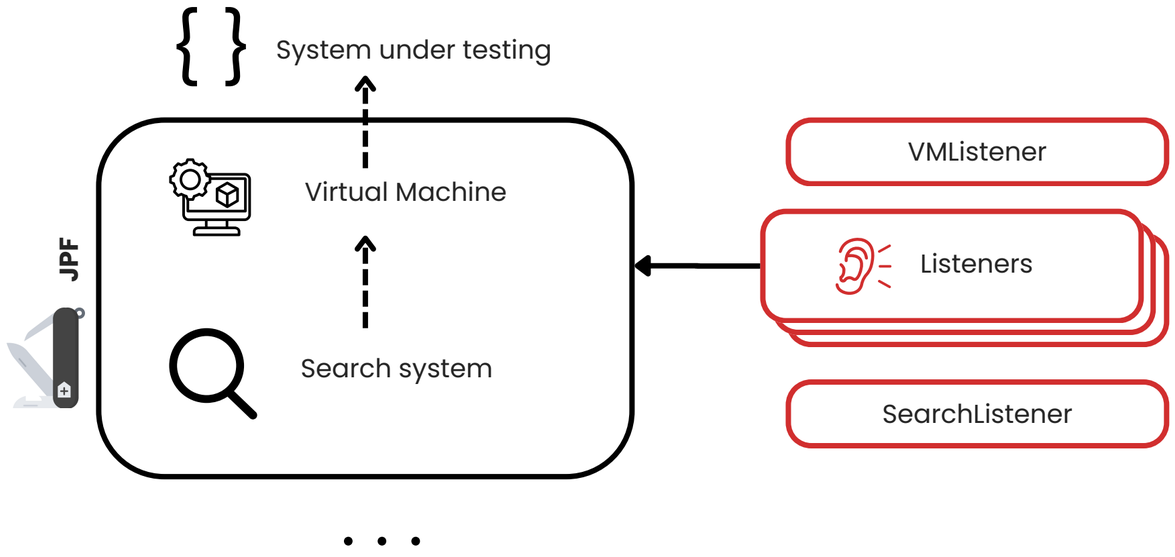
Observer of the running VM and JPF that enable to check the execution.

Main extension point of jpf.

Usable through **JPF Properties**

Diagram

External observers.
No need to rebuild JPF every time.



Using listeners

Inside the property

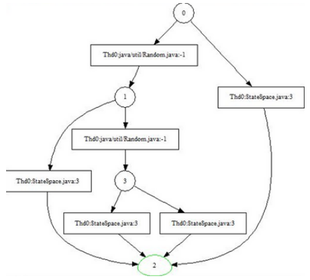
```
listener = [list of listener classes]
```

Some Listeners

Standard listeners exists

gov.nasa.jpf.listener.StateSpaceDot

Draw the state space explored by JPF.
Uses .dot (dotty format)
can be opened with GraphViz



gov.nasa.jpf.listener.BudgetChecker

Control resource contraints are not violated.

```
budeget.max_time  = ...
budeget.max_heap = ...
budeget.max_instn  = ...
budeget.max_state = ...
```

gov.nasa.jpf.listener.EndlessLoopDetector

Control heuristically that there are no infinite loops

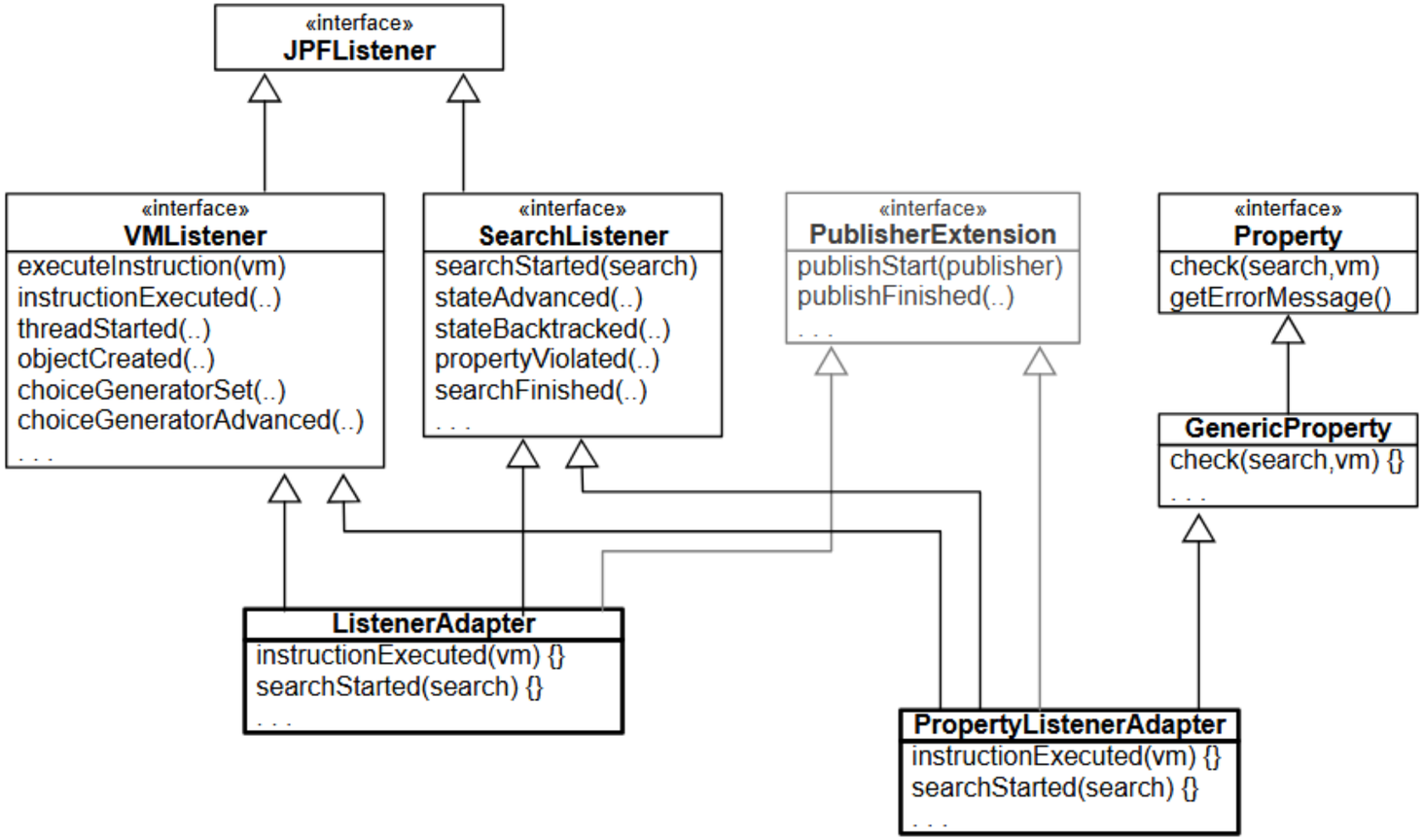
```
idle.max_backjumps = ...
```

Extending Listeners

It is complex but feasible extending JPF Classes.
You have to implement the interface or the abstract adapter for what you want to listen (JPF VM or JPF Search)

Every listener can also have **properties**.
Properties are set run time

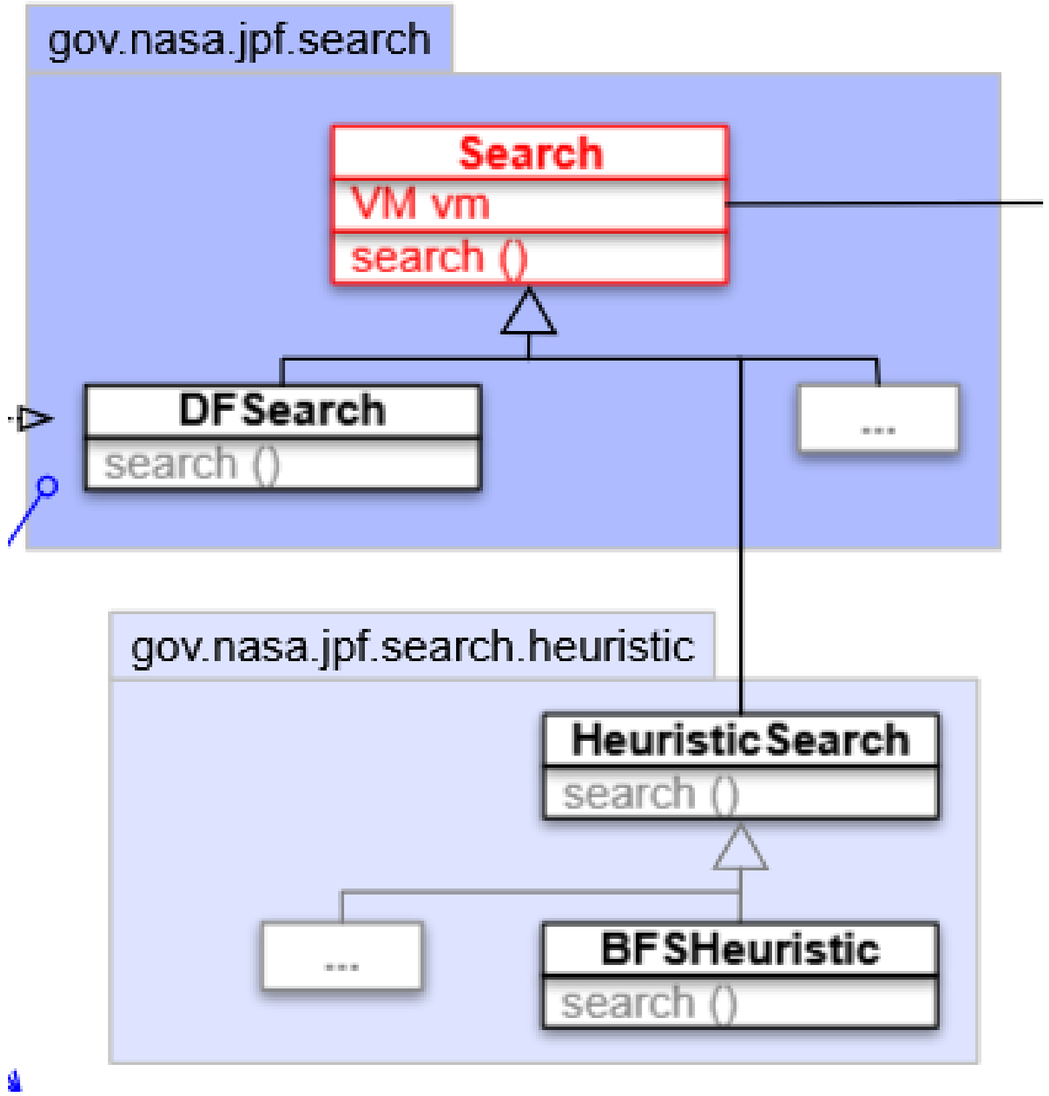
Every listener can check property violations.



Extending Search

As we've already mentioned, we can extend the Search package with our own search strategies.

This is done by extending the Search abstract class as we would normally do in java



Extending Search

Let's now see what characteristics our custom class should have, and which existing methods and attributes we can leverage from the Search class

Constructor

Our class should have a public constructor, receiving the JPF Config file and the JPF Virtual Machine reference as input

Going Forwards and Backwards

As we have already seen, we can use the methods forward() and backtrack() to either explore a new state or go back to the previously explored one. We can also leverage these methods of the Search class to understand where we are exactly in our search:

- isNewState tells us is we've never seen the current state
- isEndState tells us whether the current state is final
- isIgnoredState tests whether the current state should be ignored for search purposes

Extending Search

Let's now see what characteristics our custom class should have, and which existing methods and attributes we can leverage from the Search class

Interaction with Other Components

Listeners could end a search by manipulating the Search class attribute done: this should be considered accordingly in our code

Other components can request a backtrack through the requestBacktrack() method of Search class. We can check whether this has happened with the Search method checkAndResetBacktrackRequest()

Adapting to Search Related Properties

Let's see how to take into account some of these properties:

- search.depth_limit → use getDepthLimit()
- search.min_free → use checkStateSpaceLimit()
- search.multiple_errors → use hasPropertyTermination(), in combination with the done attribute

Extending Search

Let’s now see what characteristics our custom class should have, and which existing methods and attributes we can leverage from the Search class

Notifications

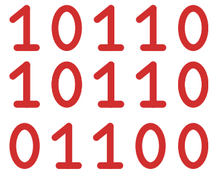
A proper search should notify different listeners of certain events. The interface SearchListener offers us a wide array of notifications:

- notifyStateAdvanced()
- notifystateBacktracked()
- notifyStateProcessed()
- notifySearchStarted()
- notifySearchFinished()
- notifyPropertyViolated()
- notifySearchConstraintHit(String)

Extending the VM

The VM implemented by JPF is itself expandable.

This is out of scope of our explanation but it is interesting!



Bytecode Factory

Enable to implement new and existing bytecodes

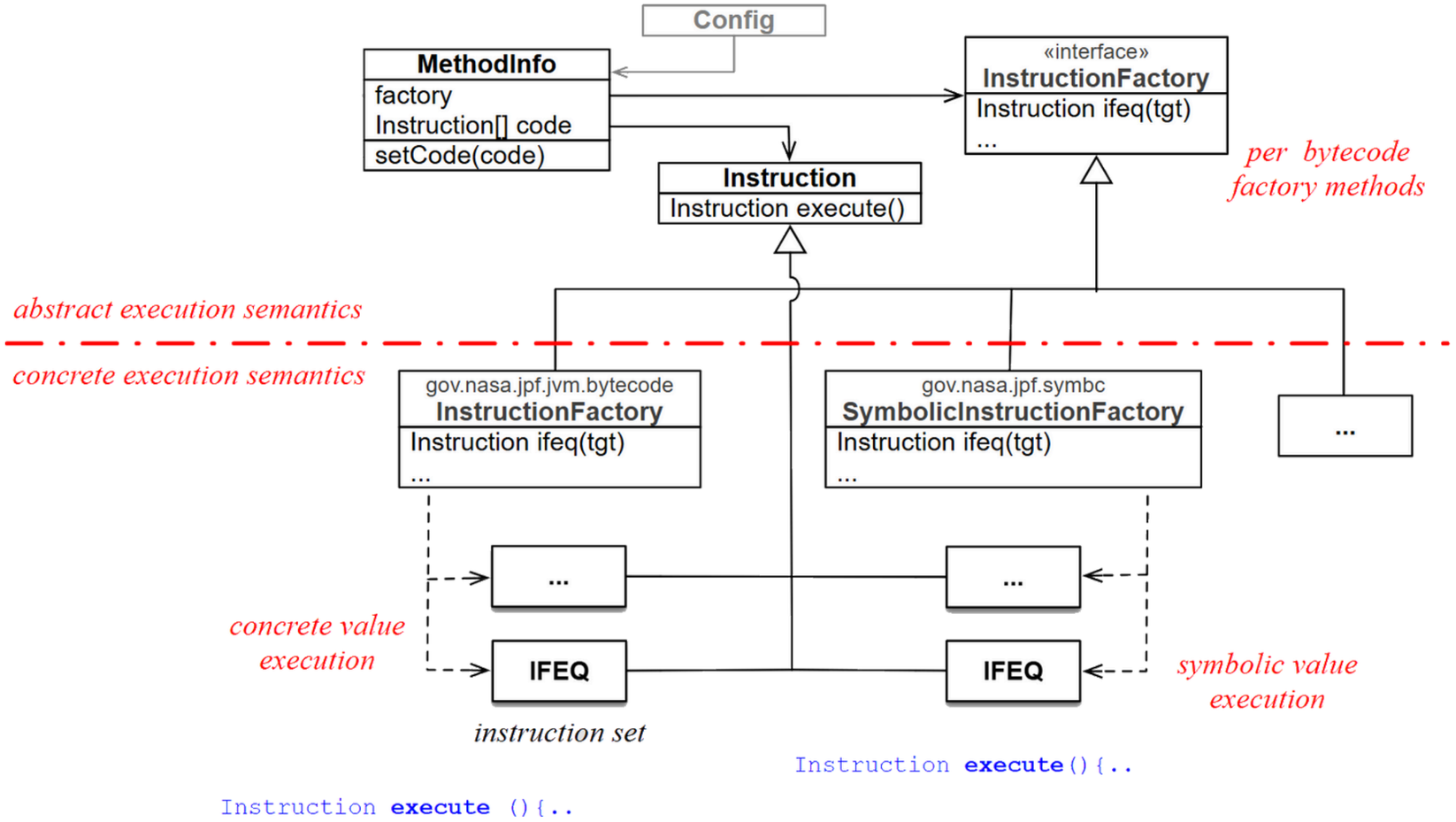
Model Java Interface

Interface SUT with Native JVM

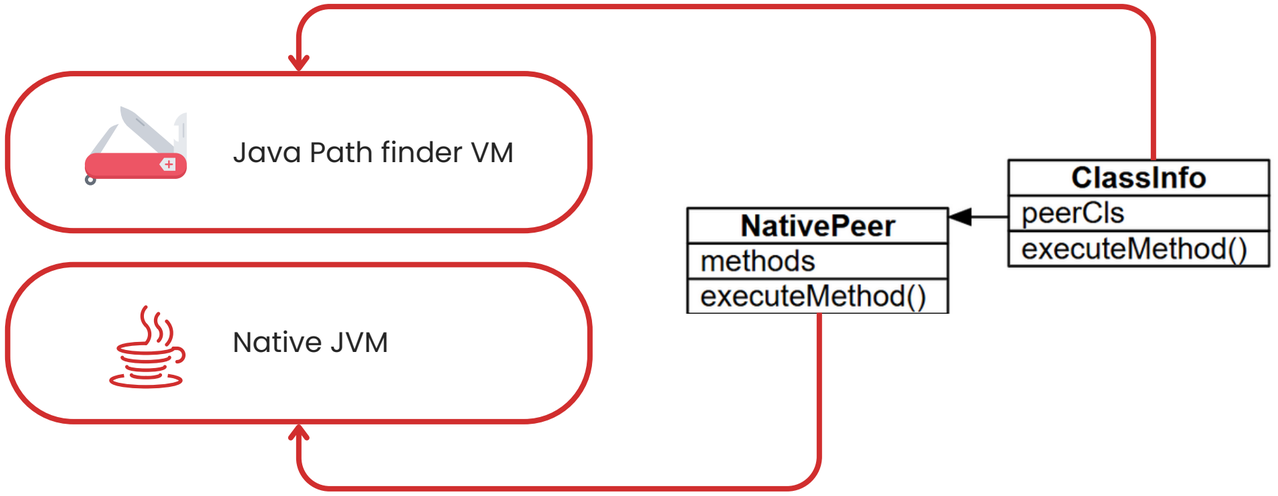
10110
10110
01100

Bytecode Factory

We can specify new instructions, and give a concrete or symbolic execution method.



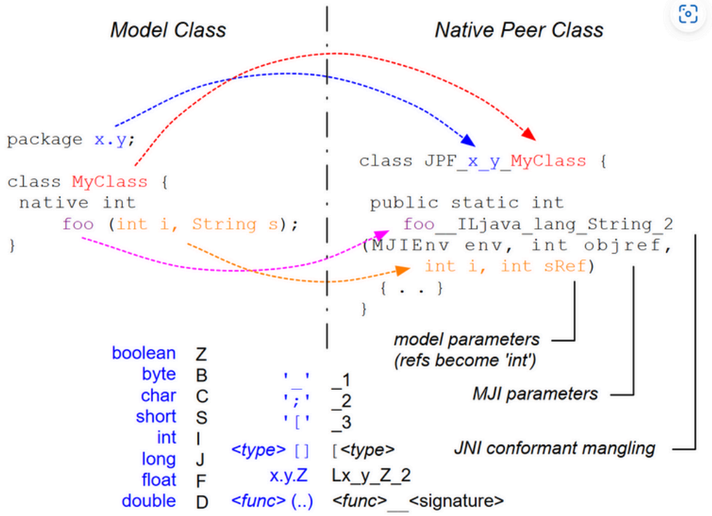
Model Java Interface



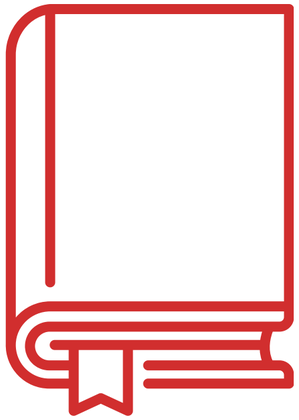
Interface SUT with Native JVM.

A Native Peer is associated to a model class to execute native method (**when loading the model**).

For system specific needs
Enabling state reduction (state are not tracked)
Still permitting some checks (ie. threads management)



Our **bibliography**



Bibliography

JPF Documentation

https://github.com/javapathfinder/jpf-core/wiki/

As of 04/2025, by JavaPathFinder

Java PathFinder: a tool to detect bugs in Java code

F. van Breugel • 2020 • York University

Model checking programs

Visser, Willem & Havelund, Klaus & Brat, Guillaume & Park, Seungjoon. (2000). Model Checking Programs. Automated Software Engineering. 10. 3-11. 10.1109/ASE.2000.873645.

Java PathFinder

A. Mannem, D. White, M. Yousaf • University of Texas at Arlington

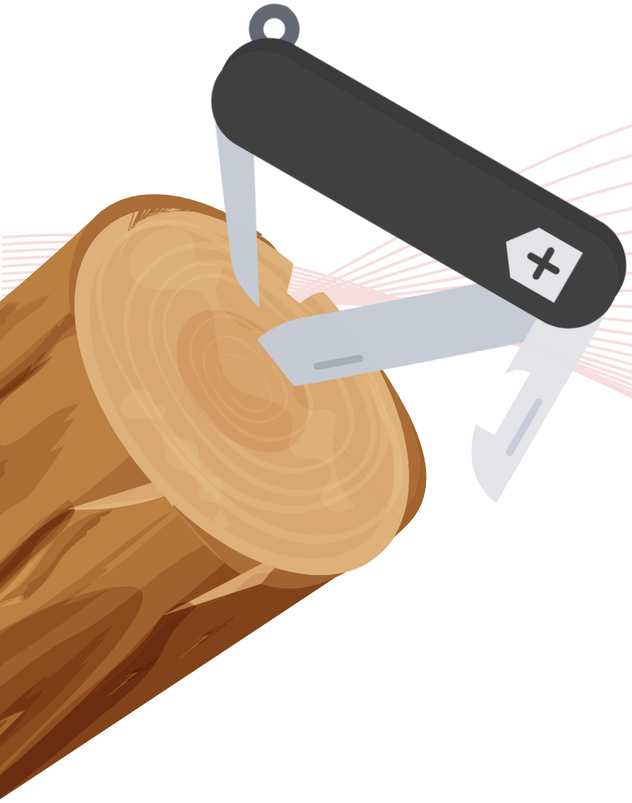
POLITECNICO
MILANO 1863

Thank **You**

30 April 2025

Domenico Popolizio, Nicola Zarbo, Enrico Vento, Federico Valentino

POLITECNICO
MILANO 1863



Time For The **Demos**