**Wikiprint Book**

**Title: JPF and Google Summer of Code 2011**

**Subject: Java Path Finder - events/soc2011**

**Version: 28**

**Date: 03/17/2013 04:26:08 PM**

## Table of Contents


**JPF and Google Summer of Code 2011** **3**
- Interested Students - Contact Us 3
- Timeline 3
- Required Skills 3
- Beginners Projects 3
- Advanced Projects 4
  - Model Checking Android Applications 4
  - JPF Inspector 5
  - Trace Server; project page: summer-projects/2010-trace-server 5
  - Dimensional Analysis 5
  - Verifying Scala Applications 5
  - Swing UI Model Checking 5
  - Search Visualization 6
  - Property Specific Visualization 6
  - Generic Temporal Properties 6
  - State Machine Visualization 6
  - Listener at SUT Level 6
  - Fast Native Collections 6
  - Java Compatibility Testsuite 6
  - State Comparison 7
  - Checking Human Machine Interactions 7
  - Extending String analysis in Symbolic Java Pathfinder 7
  - Test Generation for programs with polymorphism 7
  - Eclipse plugin for automated test generation 7
  - Support for new Java concurrency primitives 7
  - Extending support for Java 1.5 concurrency constructs 7
  - Memory Model for Java 7
  - Real-Time Specification for Java 7
  - Connect JUnit and JPF 8
  - Verifying X10 Applications 8
  - Parameterized Unit Tests 8
  - Java Platform Debugger Architecture 8
  - Checking Java Annotations 8
  - Confirm Warnings 8
  - Coverage Visualization 8
  - Security Policies 8
  - Load testing 9
  - Neko protocol simulator 9

## JPF and Google Summer of Code 2011

Java Pathfinder (JPF) is here again in 2011. We are planning to participate with a number of exciting projects as part of Google Summer of Code (GSoC) 2011. If you are new to the Google Summer of Code program, this is an annual event where Google sponsors students to work on selected open source projects, each student being supported by an experienced mentor. Projects have about 3 month scope, carry a relatively low administrative overhead, can be done remotely, and are generally fun. You also get a cool t-shirt, so what's not to like.

This page will list the project descriptions from last year, which will evolve rapidly until April 2011.

### Interested Students - Contact Us

If you have any questions or suggestions regarding JPF and GSoC, email us at <**jpf.gsoc [at] gmail.com**>. Please be sure to describe your interests and background. The more we know about you, the better we will be able to answer any questions you may have about JPF and/or its potential projects. If you are interested in a project that is not listed here but is relevant to JPF, we would love to hear about it. Join our IRC channel #jpf on freenode to engage in a discussion about all things JPF.

### Timeline

- 03/11 : Mentoring organization application deadline
- 03/14 - 03/17 : Google program administrators review org applications
- 03/18 : List of accepted mentoring organizations announced
- 03/18 - 03/27 : Student discussion period
- 03/28 - 04/08 : application period for students
- 04/25 : announcement of accepted students
- 05/25 - 08/22 : code

### Required Skills

JPF is written in Java, and it analyzes Java bytecode, so obviously you need to know about Java. At least you should know there is more to it than just the language - it's the language, the libraries and the virtual machine/bytecodes. Not all projects require all levels though, please look at the project descriptions to find out which parts are more important.

JPF is a verification tool, so you should be somewhat familiar with formal methods. However, JPF is where research meets development, so for many projects it is not a show stopper if you can't spell "Buechi automaton" backwards.

JPF is a fairly complex system. The first step to start is to get JPF running and configured. This in itself can be a steep learning curve. It also helps if you already know what listeners, bytecode factories and native peers are, but now worries - the mentors will help you there. One thing you have to look at, but what is now surprisingly simple is how to set up JPF projects.

### Beginners Projects

Due to public demand we've added two topics that are suitable for JPF novices. Yes, we know - JPF can be intimidating, and the listed topics were a bit on the heavy side. Nobody was born knowing about JPF, formal methods or VMs. We've got a number of inquiries from obviously bright and motivated students, and we want to give them a chance too. Here goes:

(A) **Create program execution diagrams** - there is the `gov.nasa.jpf.listener.SimpleDot` listener, which creates diagrams of the program state space, but it has been neglected for a long time. It's a simple JPF Listener that produces diagrams with GraphViz. It's coolness could be significantly increased, e.g. by using JUNG, by showing statements as labels instead of source locations, and in a myriad of other ways (e.g. only showing method invocations). Be creative, be an artist.

(B) **Generate JPF option lists from code** - JPF is configured by using Java properties. There is no single place where all options are specified, since JPF is an open system and each extension has its own set of options. Only that nobody remembers all of them. You could get immortal fame if you write a tool that collects all options by parsing for `gov.nasa.jpf.Config` usages and turns them into an XML list that can be translated into a number of target formats (e.g. HTML). There was a tool in the old Sourceforge distribution that was using javadoc plugins, but it would be nice to have a standalone tool with more flexibility on the output side. For example, you could use the BCEL library (which is part of the JPF distribution anyways) to collect the `Config` calls. Generations of JPF users would love you for that and buy you beer.

(C) **Write Property Annotation Doclet** - the jpf-aprop extension implements source annotations like `@Const`, `@NonNull` and even full programming-by-contract support with `@Requires`, `@Ensures` and `@Invariant`. That's not just very useful to let JPF check for corresponding violations, but it is also useful program documentation. Only that it doesn't appear in javadoc generated output, because `javadoc` doesn't know about such annotations. That is unless you write a Doclet which parses classfiles for these annotations and turns them into `javadoc` output. Are you up to the

task? No model checking required.

(D) **Interactive JPF tutorial and help system** - prepare a "beginner's guide" to JPF. Select/write simple input programs for demonstrating different extensions of JPF. Reproducible documentation of each step of the process of applying JPF / extensions. Develop a help function as an extension of JPF shell.

### Advanced Projects

**This is not an exclusive list**! If you have variations, or other project ideas altogether, let us know on <jpf.gsoc [at] gmail.com> or the JPF Google Group. The sooner, the better.

1. Model Checking Android Applications - use JPF to verify Android components
2. Verifying Scala Applications - make JPF Scala aware
3. Verifying X10 Applications - make JPF X10 aware
4. JPF Inspector - interactively control and inspect what JPF is doing
5. Trace Server - store and post-mortem analyze program traces outside JPF
6. Dimensional Analysis - specify and check physical units for integers and floating points
7. Swing UI Model Checking - extend the UI model checking script language
8. Search Visualization - create cool graphics about what JPF is doing
9. Property Specific Visualization - Visualizations related to properties specific to concurrent programs
10. Generic Temporal Properties - specify and check general event sequences
11. State Machine Visualization - port and extend the statemachine animation
12. Runtime Weaver - link and execute listeners that run at SUT level
13. Fast Native Collections - create MJI versions for the java.util collections
14. Java Compatibility Testsuite - adapt a Java testsuite to find out where we lack library support
15. State Comparison - compare differences between two program states
16. Checking Human Machine Interactions - use JPF to check properties of human machine interactions
17. Symbolic Strings - extending String analysis in Symbolic Java Pathfinder
18. Test Generation for programs with polymorphism - Generate tests for object-oriented programs using/extending SJPF
19. Test Case Generation - Eclipse plugin for automated test generation
20. Adding libraries - adding support for Java 1.6 libraries
21. jpf-concurrent - extending support for Java 1.5 concurrency constructs
22. Memory Model - implement current Java memory model
23. Checking RTSJ Programs - implement support for RTSJ into JPF
24. JUnit and JPF - connect JUnit and JPF
25. Parameterized Unit Tests - create a Pex-like user interface for parameterized unit tests
26. Java Platform Debugger Architecture - create a back-end for JPDA based on JPF
27. Checking Java Annotations - extend and develop listeners to check Java annotations in JPF
28. Confirm Warnings - dynamic confirmation of statically detected warnings
29. Coverage Visualization - show coverage achieved by JPF
30. Security Policies - verify Java security policies
31. Parallel Symbolic Execution - Scaling up symbolic execution via parallelization techniques
32. Beam Search Based Load Test Generator - Beam-search guided and incremental symbolic execution to discover test inputs leading to extensive loop iterations
33. Neko Protocol Simulator - Support for a simulation framework of protocols written in Java

---

#### Model Checking Android Applications

Android is an extremely cool programming platform for mobile devices. It is programmed in Java, event driven and component based. What is more obvious than to use JPF to model check its applications? This comes with a treasure chest of possible projects.

One thing is to model the Android framework in a similar way like what jpf-awt does. A lot of the Android classes could probably be used straight from http://source.android.com/, with replacements of the native Android libraries APIs. The goal of such a project would be to verify Android apps that come with sources, by executing them as Java bytecode in JPF, i.e. bypassing Dalvik.

Another angle would be to verify *.dex files, i.e. compiled Android apps without sources. Obviously, this would be aimed at automatically detecting malicious apps, which is a **very** hot topic right now. Since JPF executes on / models a Java VM, this means that *.dex files would either have to be re-translated into bytecode (possibly using IcedRobot), or we have to model the whole Dalvik VM in Java (possibly using http://code.google.com/p/android-dalvik-vm-on-java/).

Those are not beginner projects, and we would have to scope it down in the context of GSoC. Solid understanding of Android is a prerequisite. It will require a lot of coding, and you need to learn quickly about JPF. The reward is that you do something (possibly beyond GSoC) that really matters.

---

**JPF Inspector**

JPF can get very busy at times without you knowing what it does. Logging what is going on is fine, but what would be much better is to be able to set breakpoints, stop/resume, backtrack on demand, "single-step" over instructions or transitions, and inspect object or stackframes. Sort of like a debugger on steroids. The good thing is that we already have the framework for this - jpf-shell. What we have in mind is to add new panes to the JPF shell that allow you to do the things mentioned above, i.e. control and inspect JPF execution.

---

**Trace Server; project page: summer-projects/2010-trace-server**

Traces are memory hogs. If you have some production code SUT, it is quite normal that you end up with traces that contain millions of steps (`Instruction` objects). Not good to store millions of objects while you explore the state space and you don't even know yet if you are ever going to need the trace. The normal mitigation is to first run JPF without the "trace" topic in the reports, store the `ChoiceGenerator` path if you hit a defect or otherwise need a trace, and then replay this path with traces turned on if you need more information. This is a bit complicated. This is why the trace server was created in 2010. With such a database, you can use post mortem analyzers to find out about defects. Post mortem analyzers would not only speed up JPF in the first place, but also avoid having to re-run JPF on a large system under test if you need to try several trace analyzers. Work for 2011 includes creating more analyzers, improvements in the database, and the performance of the entire framework.

---

**Dimensional Analysis**

Ok, everybody loves to make jokes about NASA's trouble with imperial and metric unit conversion (or the lack thereof). Everybody except of the Mars Climate Observer guys - this is a serious problem for verification of technical software. There is JSR-275, but it requires using a fairly involved API, hence is expensive to add after the fact for existing software. Enter the JPF attribute system, which allows you to attach your own objects to local vars, operands, fields and even objects themselves. Such attributes automatically travel with the associated values (e.g. when a local var gets pushed), so it is convenient to use them to keep track of units. Specifying them in the first place can be done by annotations (e.g. for fields), or through an initialization API, like `double velocity = Unit.initialize(42.0, Unit.meter_per_second)`, but subsequently you just use the basic types (like `double`) for your computations and JPF does the rest. This project is about implementing such an annotation/attribute system that is easy to use with legacy applications.

---

**Verifying Scala Applications**

Scala is cool, Scala compiles into the JVM. Used the right way, it could actually avoid problems that otherwise need JPF for detection. If you stick to the paradigm, there are no shared memory races when using Scala Actors. But there still might be other defects (like deadlocks) you can do, for which JPF could be very helpful. Only that Scala's implementation is done on top of all the "nastiness" of low level Java constructs, which would show in traces and properties. The goal of this project is to make JPF Scala aware, so that for instance deadlocks would be reported at the level of `receive` patterns, i.e. the code a user can see - not the (invisible) code of Scala implementation itself. This is not restricted to actors and concurrency, i.e. could also include Scala specific properties to check for. One should look into Basset for an example on connecting Scala and JPF.

---

**Swing UI Model Checking**

The jpf-awt extension allows model checking of Swing applications, using a scripting language to specify user input sequences to explore, which looks like

```
$MyTextField:input.setText("whatever")
ANY { $Option1.doClick(), $Option2.doClick() }
...
```

Apart from only supporting a small number of Swing components, `jpf-awt` lacks in terms of expressiveness of this scripting language (which happens to be the same syntax like the statemachine scripts). This project would add support for more components like `JTree` or `JTable`, and extend the scripting language so that especially choices become more convenient, like `ANY_CHECKBOX_COMBINATION <group-name>` or `ANY_LIST_ITEM <list-name>`. The scripting could also be extended to include properties like `?TEXT_EQUALS <textfield-name> <expected value>`, so that users could completely script verification runs without also having to provide listeners checking such functional properties.

**Search Visualization**

Visual attraction is important, anybody out there arguing with this? While there are first attempts to visualize a JPF search within the jpf-shell project using JUNG to show and navigate 2d search graphs, this is more of an interactive replacement for the `ExecTracker`, i.e. a debugging tool. If you have a cool idea about using something entirely else to show what JPF is doing, like animated 3d graphics, this is the place. Everybody has seen the bubble-and-arrow DAGs, we need something with a "wow" effect.

**Property Specific Visualization**

This project looks to further extend the functionality of JPF shell by adding the ability analyze and debug different concurrent executions of the verified system. In particular, we hope to add new tabs to the shell that show: 1) An abstract slice of the verified program that only includes critical preemption points; 2) A serialization of the verified program that shows how concurrent executions are interleaved in the verification run; and 3) A direct link between the abstract slices, the serialization, and the actual source code that lets the user connect each of these views so selecting a node in the slice automatically highlights the line of code and shows where the code is executed, but which thread, in the serialization of the program. The three views can also be toggled between byte-code and source-code depending on the needs of the user.

**Generic Temporal Properties**

Temporal logic allows for a succinct specification of sequences of events. For example, any security-sensitive action requires that a user logs in beforehand. Most model checkers support either linear temporal logic (LTL) or computational tree logic (CTL).

Hard to believe - JPF has come this far without really providing generic support for LTL or CTL. Ok, we have the LTL-translator extension, but there never was a listener that is generic enough to check the Buechis it produces. We also need a good way to identify events we reason about, and simple method invocations don't cut it. This needs to be instance aware (e.g. we are only interested in the sequence of file operations with respect to the same file object), and could also include field access and others. In addition, languages like LTL are not exactly hip and easy to read for programmers, so the input could use more high level temporal specifications.

A concrete application of this could be to verify conformance with UML sequence diagrams, which could be built on top of the sequence annotations that are already in jpf-aprop (which so far only are used to create sequence diagrams, not to verify existing ones).

**State Machine Visualization**

The JPF statechart extension lets you model check UML state machines, and already contains a visualizer that shows animated state graphs. In fact, it is so useful that we want to get this ported and extended to the new jpf-shell infrastructure.

**Listener at SUT Level**

Call this a "runtime weaver". While JPF listeners that execute at the host VM (Java) level are quite useful, they sometimes can be difficult to use if you have to traverse large data structures of JPF objects, or even call System under Test (SUT) methods. The idea is to use such a "real" JPF listener to weave in code that gets executed by JPF, i.e. within the SUT, at certain times of the execution (like before and after method calls, or when accessing fields). The point is that such "SUT listeners" are compiled with/against the SUT, and therefore can access the SUT constructs much less painful. Of course one could use something like JAspect, but this seems a bit overkill, and we might want to tie execution of such code to JPF events like backtracking, i.e. something you can't pinpoint in the SUT code.

**Fast Native Collections**

Do we really have to analyze the java.util.HashMap? implementation time and again, for every system under test (SUT) that uses such collections? No, we trust Sun. At least we want to be able to, by giving JPF a MJI version for the standard Java collections, i.e. code that is executed at the host VM level and doesn't obfuscate the JPF traces. This should also speed up execution time. The trick with this is that it can require round trips, if the MJI code has to call SUT methods in order to update the collection (e.g. calling overridden `equals()` or `hashCode()` methods), but there are many cases where this isn't required.

**Java Compatibility Testsuite**

The one area where we really need to catch up is to find out which standard libraries we support, and esp. where we are missing native methods. This project could use an existing Java compatibility test suite to determine what we don't cover, preferably on the basis of an open source class library project like Mauve, or the OpenJDK itself.

**State Comparison**

Sometimes you don't need to see the whole execution history, sometimes it is enough to find out what the differences are between two program states. This project would tackle this by looking at the thread and heap snapshots of these program states, and reporting the deltas in a readable format. As an add-on, this could be integrated into jpf-shell, but the emphasis is on the underlying state comparison.

---

**Checking Human Machine Interactions**

This project will experiment with the use of JPF for checking properties of systems involving humans, user interfaces and the machine being controlled. It will focus on expressiveness of relevant properties, as well as analysis capabilities.

---

**Extending String analysis in Symbolic Java Pathfinder**

Extend the symbolic execution to allow efficient string analysis of all Java string operations. For example, concatenation, startsWith, endsWith as well as operations where there is interaction between integer and string operations, such as indexOf. The implementation should support both automata-based decision procedures for strings as well as bit-vector based approaches. For the former the Java String Analyzer (JSA) system can be used and for the latter HAMPI. The project should include a detailed comparison of these two approaches.

---

**Test Generation for programs with polymorphism**

Generating test cases for object-oriented programs with minimal redundancy. There are two primary problems, (1) automatically generating a sequence of methods invocations that lead to a required shape of the heap and (2) Given a particular program generating a minimal set of test cases for obtaining optimal coverage. Intuitively, there may be multiple subtypes of an object that essentially test the same part of the program; the goal of this work is to remove the redundancy during the automated test case generation part. The work involves the use of lazy and uber-lazy initialization in symbolic Java Pathfinder and potentially extending it for additional features.

---

**Eclipse plugin for automated test generation**

Develop an Eclipse plugin that will automatically generate tests to obtain the optimal branch coverage of the methods that are saved (assuming no compilation errors). The system should support pre- and post-condition annotations that can respectively restrict the inputs to a method and be used to calculate the oracle portion of the test-case. In essence this should work in the same manner as the current Eclipse system when a file is saved and compilation errors are shown, here it just needs to immediately produce a JUnit test suite for all methods that have changed since the last time tests were generated. This will require integration with the symbolic execution extension for JPF to generate the tests.

---

**Support for new Java concurrency primitives**

This project is to bring on-board in JPF all of the Java 1.6 concurrency primitives that enable wait-free and lock-free constructs. Such an addition to JPF will enable JPF to verify advanced concurrent containers as well as transactional memory systems.

---

**Extending support for Java 1.5 concurrency constructs**

Java 5 introduced new concurrency constructs. Classes like ReentrantLock, Semaphore or CyclicBarrier became very popular and widely used in sophisticated concurrent applications. Enabling JPF to verify this applications is very important task. Large part of this project has been done already in jpf-concurrent extension. Student assignment will be to implement around 12 classes(out of 35) from java.util.concurrent package and prepare extensive test suite based on TCK tests.

---

**Memory Model for Java**

This project is to implement the current Java memory model standard in JPF so that JPF's exploration includes the behavior where you see stale loads from volatile and non-volatile shared variables. The volatile keyword affects whether memory is sequentially consistent or not. When memory is not sequentially consistent, JPF should show that Dekker-like synchronization on shared variables breaks.

---

**Real-Time Specification for Java**

The domain of real-time applications is where model checking can be very useful, as such applications are often used in critical systems. This project would add support for RTSJ API and semantics into JPF. It will also provide a test suite for the purpose of evaluating the coverage of RTSJ API and correctness of implementation with respect to the RTSJ specification.

**Connect JUnit and JPF**

The project will make JUnit run on top of JPF (or make JPF run as a special JUnit runner, in particular for JUnit version 4). The basic features are to get tests running in JPF such that, for example, a multithreaded test has all its schedules expored by JPF and fails if any of those schedules fails. For this, one needs to handle several advanced features of JUnit such as annotations for methods (as done in JUnit 4) or configuration files (as JUnit 3.8.1 had). An advanced feature is to get a GUI changed such that it support JUnit tests running in JPF, for example change Eclipse's JUnit plugin to properly show failing and passing tests (and also show appropriate traces for failing tests) or change JUnit's standalone GUI to properly show failing and passing tests. This project could extend previous work on unit checking.

**Verifying X10 Applications**

X10 is a new programming language being developed at IBM Research in collaboration with academic partners. X10 can be compiled to Java, and thus X10 programs should be verifiable with JPF. However, for such verification to scale and provide useful output (at the level of X10 source and not compiled Java code), it is necessary to customize JPF for X10. This project will look into such customization, making X10 applications verifiable with JPF. The project is related to verifying Scala applications with JPF.

**Parameterized Unit Tests**

Port the parameterized unit tests from GSoC'08 to the new JPF extension model, and create a Pex-like user interface for it.

**Java Platform Debugger Architecture**

The project will create a back-end for JPDA based on JPF. It will allow to use JPF instead of a common JVM for the purpose of debugging Java applications in a modern Java IDE (e.g., NetBeans or Eclipse). The key task would be to implement the JDWP protocol on top of JPF - the JDWP protocol can be used by debuggers in IDEs to retrieve information about the state of a debugged program and to control its execution. This project is related to JPF Inspector.

**Checking Java Annotations**

Program annotations are a mechanism for specifying information, e.g., properties, about a program. Annotations do not affect the execution of the code, however, they do provide a mechanism for documenting the code and can be used to analyze the program. For example, one can specify the @SandBox annotation to specify that a class is allowed to modify only its own fields but no other fields. This project will build on the existing jpf-aprop project framework by extending and creating JPF listeners to check various annotations useful in developing Java programs following a *Programming by Contract* approach.

**Confirm Warnings**

Existing techniques for software verification can be broadly classified into two major categories: dynamic and static. Dynamic verification techniques suffer from scalability issues when applied to large applications. On the other hand, static verification techniques, despite scalable, suffer from a large number of false warnings due to their conservative nature. Therefore, an ideal approach is to statically detect violations and dynamically confirm those violations. This project borrows ideas from existing approaches such as Check ‘n’ Crash, and involves developing new techniques in JPF to prune states that do not help in dynamically confirming the statically detected violations.

**Coverage Visualization**

This project involves developing new techniques for visualizing the coverage information achieved by JPF (e.g., its symbolic execution based tester). Visualization helps JPF users by assisting users in navigating both code under test and test code, thereby assisting users to understand and infer reasons of why certain code portions are not covered. This project will be implemented as an Eclipse plug-in by using Eclipse Visualization Toolkit including SWT, GEF, and Zest.

**Security Policies**

This project extends JPF for verifying security policies. Access control mechanisms use security policies to control which subjects (such as users or processes) have access to which resources. In general, a policy is explicitly specified in a well-defined policy representation (e.g., XACML) with domain-specific syntax and semantics. Faults in security policies can cause security problems (such as unauthorized access to data). This project will involve using JPF for verifying security policies. In particular, this project will convert policies into Java programs and uses JPF for verifying those security policies.

**Load testing**

Load testing aims to reveal faults that are manifested when a system is heavily exercised. Existing load test generation tools focus on increasing the size or rate of the input but provide no support for choosing the particular inputs, which can dramatically impact how heavily the system is exercised. To address this lack, we introduce an approach for smarter load test case generation based on the observation that often the most effective load tests iterate extensively through loops as they create or traverse data structures, or access or consume resources. We leverage this observation by performing a beam-search guided and incremental symbolic execution to discover path conditions leading to extensive loop iterations. The resulting path conditions are then compared and solved to generate a suite of diverse and loaded tests. The approach assessment reveals that it can generate effective and diverse load tests, and can scale to large input sizes.

---

**Neko protocol simulator**

Neko [ http://ddsg.jaist.ac.jp/neko/ ] is a Java framework for constructing distributed algorithms. Processes interact using message passing. It is highly extensible, yet simple and easy to use. Algorithms can be either simulated or executed on a real network; the same Java code is used in both cases. A variety of simulated and real networks are supported, and a library of fault tolerance algorithms has been implemented. It would be great if we could not just simulate these protocols in the JVM, but actually verify them using JPF. Work to do here includes missing support for native libraries, such as for loading XML configuration files. This would also benefit other applications, and greatly reduce the amount of manual abstraction needed to analyze complex applications in JPF. Once the basics are supported, further work includes optimizations, possibly using abstractions over the protocol state space.