The JPF Runtime Verification System

Peter C. Mehlitz
CSC/NASA Ames Research Center
pcmehlitz@email.arc.nasa.gov

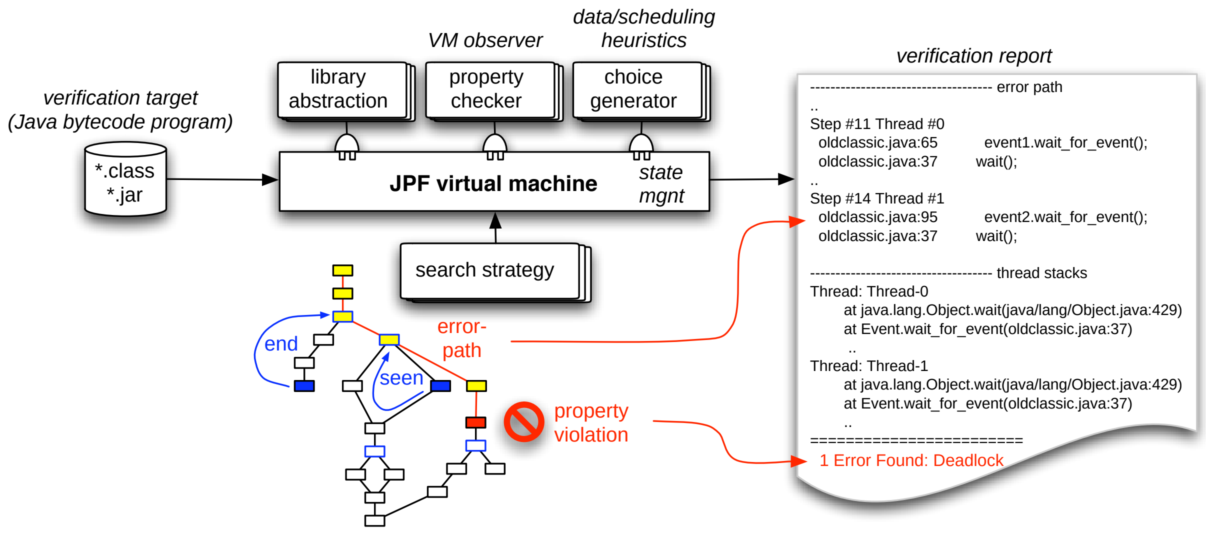
Willem Visser
RIACS/NASA Ames Research Center
wvisser@email.arc.nasa.gov

John Penix
NASA Ames Research Center
Jphn.Penix@nasa.gov

What is Java PathFinder

The answer used to be simple: "JPF is an explicit state software model checker for Java bytecode". Today, JPF is a swiss army knife for all sort of runtime based verification purposes.

If you are not familiar with formal methods, this basically means JPF is a Java virtual machine that executes your program not just once (like a normal VM), but theoretically in all possible ways, checking for property violations like deadlocks or unhandled exceptions along all potential execution paths. If it finds an error, JPF reports the whole execution that leads to it. Unlike a normal debugger, JPF keeps track of every step how it got to the defect.



```
----------------------------------- error path
..
Step #11 Thread #0
  oldclassic.java:65         event1.wait_for_event();
  oldclassic.java:37       wait();
..
Step #14 Thread #1
  oldclassic.java:95         event2.wait_for_event();
  oldclassic.java:37       wait();

----------------------------------- thread stacks
Thread: Thread-0
        at java.lang.Object.wait(java/lang/Object.java:429)
        at Event.wait_for_event(oldclassic.java:37)
         ..
Thread: Thread-1
        at java.lang.Object.wait(java/lang/Object.java:429)
        at Event.wait_for_event(oldclassic.java:37)
        ..
========================
  1 Error Found: Deadlock
```

What can be checked by JPF

Which defects can be found by JPF? Out of the box, JPF can search for deadlocks and unhandled exceptions (e.g. NullPointerExceptions and AssertionErrors), but the user can provide own property classes, or write listener-extensions to implement other property checks (like race conditions).

What programs can be checked by JPF? In general, JPF is capable of checking every Java program that does not depend on unsupported native methods. The JPF VM cannot execute platform specific, native code. This especially imposes a restriction as to what standard libraries can be used from within the application under test. While it is possible to write these library versions, especially by using the Model Java Interface (MJI) mechanism of JPF, there is currently no support for java.awt, java.net, and only limited support for java.io. Another restriction is given by JPF's state storage requirements, which effectively limits the size of checkable applications to ~10kloc (depending on their internal structure) if no application and property specific abstractions are used. Because of these library and size limitations, JPF so far has been mainly used for applications that are models, but require a full procedural programming language. JPF is especially useful to verify concurrent Java programs, due to its systematic exploration of scheduling sequences.

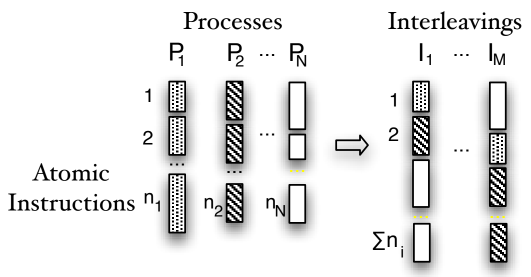
Model Checking vs Testing

What can JPF do that cannot be achieved with normal testing? JPF can simulate non-determinism. Certain aspects like scheduling sequences cannot be controlled by a test driver, and require help from the execution environment (VM). Other sources of non-determinism like random input data are supported with special APIs which can significantly ease the creation of test drivers. Simulating non-determinism requires more than just the systematic generation of all non-deterministic choices. Two capabilities come into play to make this work: backtracking and state matching.

(1) Backtracking means that JPF can restore previous execution states, to see if there are unexplored choices left. For instance, if JPF reaches a program end state, it can walk backwards to find different possible scheduling sequences that have not been executed yet. While this theoretically can be achieved by re-executing the program from the beginning, backtracking is a much more efficient mechanism if state storage is optimized.

(2) State Matching is another key mechanism to avoid unnecessary work. The execution state of a program mainly consists of heap and thread-stack snapshots. While JPF executes, it checks every new state if it already has seen an equal one, in which case there is no use to continue along the current execution path, and JPF can backtrack to the nearest non-explored non-deterministic choice.

In theory, explicit state model checking is a rigorous method - all choices are explored, if there is any defect, it will be found. Unfortunately, software model checking can only provide this rigor for reasonably small programs (usually <10,000 loc), since the number of states rapidly exceeds computational limits for complex programs. This problem is known as state space explosion, and can be easily illustrated by the number of possible scheduling sequences for a given number of processes consisting of atomic sections.



$$M = \frac{(\Sigma_{i=1}^{N} n_i)!}{\Pi_{i=1}^{N}(n_i!)}$$

JPF addresses this scalability problem in three ways: (1) configurable search strategies, (2) reducing the number of states, and (3) reducing state storage costs.

(1) Configurable search strategies try to solve the problem that the whole state space cannot be searched by directing the search so that defects are found quicker, i.e. with less computational resources. This basically means to use the model checker not as a 'proof-', but as a 'debugging-' tool, which is mostly achieved by using heuristics to order and filter the set of potential follow-on states according to some property related relevance. Computation of heuristic values is delegated to a user configured class, i.e. is not hard-coded in the JPF core.

(2) Reducing the number of states that have to be stored is the preferred way to improve scalability, and is supported by a number of mechanisms:

- **Heuristic Choice Generators** means the set of choices in a certain state does not have to be complete. Consider a non-deterministic input float value with a threshold behavior. The float type makes it impossible to generate all possible values anyways, but in terms of checking the system behavior it might be sufficient to try only three choices: less than, equal, and greater than the threshold. The important capability is to make these heuristics configurable so that they can be easily extended or adapted to specific application needs.

- **Partial Order Reduction** is the most important mechanism to reduce the state space in concurrent programs. The goal is to only consider context switches at operations that can have effects across thread boundaries, like PUTFIELD instructions on objects that are accessible from different threads. The challenge is to do this on-the-fly, without requiring error-prone user instrumentation. JPFs partial order reduction makes use of the Java bytecodes, and reachability information obtained from the garbage collector, to achieve this.
- **Host VM Execution** - JPF is a JVM that is written in Java, i.e. it runs on top of a host VM. For components that are not property-relevant, it makes sense to delegate the execution from the state-tracked JPF into the non-state tracked host VM. The corresponding Model Java Interface (MJI) mechanism is especially suitable to handle IO simulaion and other standard library functionality.
- **State Abstraction** - per default, JPF stores all heap, stack and thread changes, which is sometimes a huge overhead if it comes to deciding whether two execution states differ from the perspective of a certain application. For example, state matching based on shape analysis of data structures can yield significant state reduction, and has been successfully used in recent JPF applications

(3) Reducing state storage costs refers mainly to implementation features of the JPF core. While not being the primary measure to deal with state space explosion, efficient state storage is mandatory for a software model checker. Since state transitions usually result in a small amount of changes (e.g. a single stack frame), JPF uses a technique called state collapsing to bring down the per-state memory requirements by storing indexes into state-component specific pools (hash tables) instead of directly storing changed values.

To compare states, JPF extends the state collapsing mechanism by hashing the resulting pool-index vectors, using a single, consecutive number as a unique state-id, thus reducing state equality checks to single integer comparisons. The hash mechanism (state set implementation instead of hash table) is configurable, using MD5 as default. The 128 bit hash values make it much more likely to run out of state memory before ever encountering a hash collision.

Extensibility

From the list of JPF features mentioned above, it should be clear that the system is not a classical model checker anymore. One can think of JPF as an execution system framework for all kinds of dynamic, runtime oriented verification purposes. JPF tries to overcome the systematic scalability problem of software model checking by application- and property- specific adaptation. As a consequence, the major design force driving it's further development is extensibility. This document includes descriptions of two major extension mechanisms: (1) Search-/VMListeners and (2) Model Java Interface (MJI).

(1) Search-/VMListeners provide a convenient way to to extend JPFs internal state model, add more complex property checks, direct searches, or simply gather execution statistics. This is achieved by an Observer pattern that lets concrete observers (listeners) subscribe to certain events inside JPF, like bytecode instruction execution or forward/backtrack steps.

(2) The Model Java Interface (MJI) is a mechanism to separate and communicate between state-tracked execution inside the JPF JVM, and non-state tracked execution inside the underlying host VM (executing JPF itself). This can be used to build standard library abstractions that significantly reduce the application state space.

The State of Affairs

JPF is now in its fourth year of active development, but is still a moving target. There are a number of ongoing and planned areas of work:

(1) Structural cleanup - JPF started as a research tool, and has seen many contributors with different goals over time. As a result, its internal structure still needs to be re-factored, especially with respect to encapsulation of core classes that are related to state management. There are still too many remaining direct field accesses.

(2) Enhanced extensibility - while two major extension mechanisms (Listeners and MJI) are already in place, the configurable choice generators still need to be implemented. This might even include turning Scheduler instances into ordinary choice generators - a step that might help to adapt JPF to specific scheduling needs like Realtime Java.

A second branch of extension mechanisms are application- and property- specific state abstractions. In many cases, state hashing based on complete heap and thread information is too expensive or not even suitable (over-approximation) to identify property relevant execution paths. The complete state is only required for backtracking purposes, but not to prune "visited" parts of the state graph. Implementing a suitable interface to generate state abstractions (e.g. based on data structure shapes) still waits to be done.

(3) Library abstractions - JPF executes bytecode, i.e. analyses not only the application under test, but also all library code used by it, which often significantly exceeds the application size. For many properties, library code is not of interest, and should not be state tracked. The Model Java Interface (MJI) provides a suitable mechanism to replace real library code with abstractions that can be executed outside of the JPF VM, e.g. to model IO operations. Using MJI to abstract standard Java libraries is a major step towards applying JPF to real Java production code.

(4) Execution cost and time model - in its current state, JPF does not model time, which is a prerequisite to adapt JPF to Realtime Java. In order to introduce time, execution costs need to be approximated. This can be done in various degrees of fidelity (interpreter, JIT, AOT), and needs to be adaptable to different target platforms (architectures, OSes), and hence should be kept outside of the JPF kernel, using VM listeners that monitor bytecode execution to compute and store time as a state-extension. First prototypes of corresponding listeners have already been implemented.

History and Credits

JPF has come a long way from it's beginnings in 1999. Four major phases stand out

- 1999 Java-to-Promela translator (using Spin as the model checker)
- 2000 JVM / standalone checker
- 2003 design and implementation of extension structure
- 2005 opensourcing of JPF

During this time, many people and institutions have worked on and with JPF. The majority of work is still done by the Robust Software Engineering (RSE) group at the NASA Ames Research Center. The incomplete list of key contributors (in random order) includes:

Klaus Havelund who had the orignal idea to create a software model checker based on Java.

Willem Visser is (together with Corina) the driving force behind the research part of JPF. He continues to drive it into exciting new places like symbolic model checking and test case generation.

Flavio Lerda did most of the work for phase 2, turning JPF from a translator into a VM

Corina Pasareanu continues to tweak JPF as a testbed for symbiosis of symbolic and explicit state model checking

John Penix fought hard through the NASA ranks to make the JPF opensourcing happen, not at least succceeding based on his reputation as one of the few people who actually have been able to find real defects with software model checking

Masoud Mansouri-Samani shielded the developers from configuration and distribution management in the dark ages of closed JPF sources

Owen O'Malley showed his programming skills by implementing the MD5 based state hashing (with an interesting red-black tree), a huge win in terms of state storage. Owen also added some java.io library support based on MJI.

Dimitra Giannakopoulou happily uses JPF to distribute her famous LTL-to-Buechi-automaton translator (e.g. used in the LTSA system), which is so good that it remains in the JPF distribution even while JPF's LTL search is currently defunct. The translator can be built and used independently from the rest of JPF.

Peter Mehlitz the "refactorator", trying to fill the Java holes, coming up with extension mechanisms, and getting all this into a manageable package

Prerequisites

JPF is a pure Java application that requires at least a Java 1.4.1 runtime. The following third party libraries are used to run JPF:

- BCEL (the Bytecode Engineering Library from <http://jakarta.apache.org/bcel>, usually in bcel.jar), to load classfiles
- Xerces (the XML parsing library from <http://xml.apache.org/xerces2-j>, usually in xercesImpl.jar), to parse eexecution paths stored as XML
- the MD5 libary from Timothy W. Macinta <http://www.twmacinta.com/myjava/fast_md5.php>, usually in fast-md5-version.zip), to efficiently build MD5 state hash codes

All these libraries have to be in the CLASSPATH, but can reside outside the JPF directory tree. To build JPF, the following tools have to be installed:

- Ant <http://ant.apache.org>, to resolve dependencies and compile JPF sources
- JUnit <http://www.junit.org>, to run unit tests

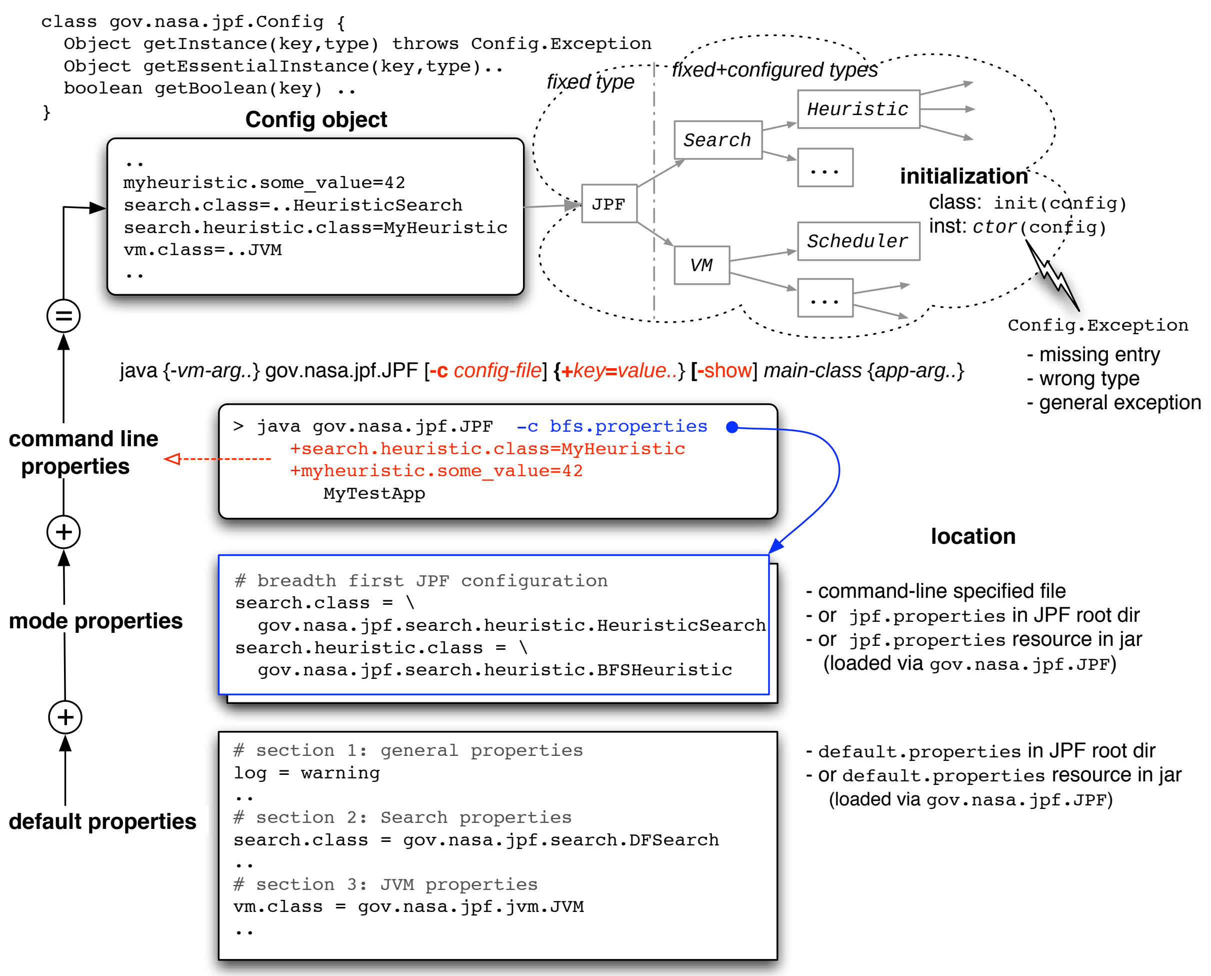
For convenience reasons, there is a build-tools module in JPF that contains unmodified, tested, but not necessarily up-to-date versions of these tools.

Installing JPF

fetch from CVS or distrib, get libs, set CLASSPATH, run

Configuring JPF Runtime Options

Since JPF is an open system that can be parameterized and extended in a variety of ways, there is a strong need for a general configuration mechanism. The challenge for this mechanism is that many of the parts which are subject to parameterization are configured themselves (i.e. classes instantiated via classname parameters). This effectively prohibits the use of a configuration object that contains concrete fields to hold configuration data, since this class would be a central "design bottleneck" for a potentially open number of concrete JPF components like Search, Heuristic and Scheduler implementations. The goal is to have a configuration object that (1) is based on symbolic values, (2) can be extended at will, and (3) is passed down in a hierarchical initialization process so that every component extracts only its own parameters.


```
class gov.nasa.jpf.Config {
  Object getInstance(key,type) throws Config.Exception
  Object getEssentialInstance(key,type)..
  boolean getBoolean(key) ..
}
```

```
..
myheuristic.some_value=42
search.class=..HeuristicSearch
search.heuristic.class=MyHeuristic
vm.class=..JVM
..
```

java {-vm-arg..} gov.nasa.jpf.JPF [-c config-file] {+key=value..} [-show] main-class {app-arg..}

```
> java gov.nasa.jpf.JPF  -c bfs.properties
     +search.heuristic.class=MyHeuristic
     +myheuristic.some_value=42
        MyTestApp
```

```
# breadth first JPF configuration
search.class = \
  gov.nasa.jpf.search.heuristic.HeuristicSearch
search.heuristic.class = \
  gov.nasa.jpf.search.heuristic.BFSHeuristic
```

```
# section 1: general properties
log = warning
..
# section 2: Search properties
search.class = gov.nasa.jpf.search.DFSearch
..
# section 3: JVM properties
vm.class = gov.nasa.jpf.jvm.JVM
..
```

The JPF configuration process uses a java.util.Property subclass to achieve this. This property instance is initialized in three steps:

1. Default Properties are taken from a default.properties file residing in the JPF root directory, or - if there is no such file - from a resource loaded via the gov.nasa.jpf.JPF class itself. This is usually the biggest file/resource, and contains many settings that are seldom changed for normal usage

2. Mode Properties are taken from a jpf.properties file in the JPF root directory, or a correspoding resource loaded via gov.nsas.jpf.JPF if the file is non-existent. A mode property file can also be explic-

itly specified via the -c <config-file> command line option. Mode properties are usually small, mainly containing the classnames of specialized Search and Heuristics classes and their respective parameters

3. Command Line Properties are overlayed on top of the mode properties, to conveniently modify single parameters without the need to change default or mode property files. Command line properties are specified using a +<key>=<value> notation, and are mostly used during development and testing of new components

The resulting property object only holds key/value pairs with String values. It is an instance of gov.nasa.jpf.Config, a utility class that especially contains methods to conveniently instantiate objects from String values, and has separate accessors for optional and mandatory entries. In case of instantiation errors or missing mandatory entries, a gov.nasa.jpf.Config.Exception is thrown.

The Config object instantiates configured classes with the following constructor lookup scheme

- using parameter types and values that were explicitly specified in the instance request
- if no such constructor is found, or no parameters were specified, it looks up a <classname>(Config) constructor, and passes itself as parameter
- if no such constructor is found, it uses a default constructor
- if no default constructor is found, a Config.Exception is raised

The method to request a configured instance can also use an optional type parameter to guarantee type conformance of the created instance, and raise a Config.Exception in case the instance does not satisfy this type constraint.

The Config object itself is created by gov.nasa.jpf.JPF (the main application class), and - by using its API to instantiate configured classes - is passed down in a hierarchical initialization process so that every class that is instantiated has access to it. Instantiated objects can then retrieve their corresponding parameters, and optionally transform and store them in more specialized representations (e.g. int or boolean fields).

For a detailed description of standard properties, see the comments in jpf.properties. The following keys stand out as being application and program property specific:

vm.classpath - colon separated list of directories that are used by JPF to load classes required by the application under test. If a class is not found there, the standard CLASSPATH is searched too.

vm.sourcepath - corresponding list of directories that are searched for sources (in case JPF reports an error)

search.class - fully qualified class name of the Search class to use

listener - colon separated list of classnames that are used to instantiate Search- and VMListeners

Running JPF

Executing JPF from the command line is easy - the jpf script (residing in the jpf/bin directory) acts as a drop in replacement for the normal 'java' executable. In case you don't have specific initialization needs (see [sec:Lots-of-Options]), all you need to do is to call the bin/jpf script with the class name of the main application class to check, and append any arguments the application is expecting. The formal syntax is:

```
> bin/jpf [-c config-file] [-show] {+key=value ..} app-class {app-args ..}
```

-c config-file optionally specifies the java.util.Properties file that should be used for JPF configuration (default is 'jpf.properties')

-show directs JPF to print out the configuration key/value pairs prior to running the application

+key=value is a convenient way to override configuration properties via the commandline

In case you don't want to use the bin/jpf script, you have to setup the classpath and specify gov.nasa.jpf.JPF as the main class to be executed by java

```
> java {vm-args..} gov.nasa.jpf.JPF jpf-args
```

When executing java directly, it is a wise idea to increase the maximum heap space with the -Xmx VM argument (e.g. -Xmx1024m)

To setup the classpath, make sure the following code is reachable, either by setting the CLASSPATH environment variable, or by using the -classpath VM command line argument

- JPF classes (either explicitly from the jpf/build/jpf/ directory, or implicitly via jpf.jar)
- library abstractions to use by JPF (per default in the jpf/build/env/jvm/ directory, or in jpf.jar)
- BCEL (the Bytecode Engineering Library from <http://jakarta.apache.org/bcel/>, usually in bcel.jar)
- Xerces (the XML parsing library from <http://xml.apache.org/xerces2-j/>, usually in xercesImpl.jar)
- the MD5 libary from <http://www.twmacinta.com/myjava/fast_md5.php>, usually in fast-md5-version.zip)
- optionally - constraint resolver library if you use JPF with its symbolic execution extension
- optionally - your additional JPF extension classes (listeners, properties etc.)

JPF can also be used embedded (e.g. an IDE), i.e. called from another Java application. The jpf/src/gov/nasa/jpf/tools/ directory contains various examples, e.g. the ExecTracker application that logs various aspects of JPF execution. A basic code sequence to start JPF looks like this:

```
import gov.nasa.jpf.JPF;
import gov.nasa.jpf.Config;
import gov.nasa.jpf.SearchListener;
import gov.nasa.jpf.VMListener;
..
void runJPF (String[] args) {
  ..
  MyListener listener = new MyListener(..);
  listener.filterArgs( args);  // 'null' any consumed args not to be  JPF-processed
  ..
  Config config = JPF.createConfig( args);
  // set special config key/value pairs here..
  JPF jpf = new JPF( config);
  jpf.addVMListener( listener); // or addSearchListener
  jpf.run();
  ..
}
```

How to Implement Properties

There are two general types of property checks that can be performed with JPF: (1) gov.nasa.jpf.Property based, and (2) gov.nasa.jpf.SearchListener or gov.nasa.jpf.VMListener based.

(1) gov.nasa.jpf.Property instances are used to encapsulate property checks. These instances can be configured statically (via the search.properties setting) or dynamically (via jpf.getSearch().addProperty()), and are checked by the Search object after each transition. In case a Property.check(..) method implementation returns false, and termination has been requested, the search process is ended, and all violated properties are printed (which potentially includes error traces)

JPF comes with the following generic Property classes:

- gov.nasa.jpf.jvm.NotDeadlockedProperty - for every non-end state, test if there is any runnable threadleft
- gov.nasa.jpf.jvm.NoAssertionViolatedProperty - test if any assertion expression has been violated
- gov.nasa.jpf.jvm.NoUncaughtExceptionsProperty - test if any exception was not handled inside the application

New properties can be added by providing additional implementors of the gov.nasa.jpf.Property interface

```
public interface Property extends Printable {
  boolean check (VM vm, Object arg);
  String getErrorMessage();
}
```

or, to save some efforts mostly associated with printing out error traces, by deriving classes from gov.nasa.jpf.GenericProperty, which requires only the check(..) method to be overriden. To configure these new checks, add them to the colon separated list of classnames specified under search.properties in a JPF configuration file (either default or mode specific):

```
search.properties=\
  gov.nasa.jpf.jvm.NotDeadlockedProperty:\
  gov.nasa.jpf.jvm.NoAssertionViolatedProperty:\
  gov.nasa.jpf.jvm.NoUncaughtExceptionsProperty:\
  x.y.z.MyNewProperty
```

(2) gov.nasa.jpf.SearchListener and gov.nasa.jpf.VMListener instances can be used to implement more complex checks that do require more information than what is available after a transition got executed. The rich set of callbacks enables listeners to monitor almost all JPF operations and translate them into internal state. JPF execution control can be achieved in two ways:

(a) by implementing both the appropriate listener interface and the gov.nasa.jpf.Property interface, then registering with Search.addProperty(..), to let JPF automatically check for violated property termination between states.

(b) by calling Search.terminate() to stop searching for new states. This can be done from anywhere within the listener, but does not automatically create error reports, which have to be done explicitly by the the listener.

Instrumenting Applications with Verify

Ideally, JPF can be used to verify arbitrary Java applications, but often, these applications are Java models of other systems. In this case, it can be helpful to call JPF APIs from within the application, to obtain information from JPF or direct its further execution. The JPF API is centralized in the gov.nasa.jpf.jvm.Verify class, which includes methods from the following major categories:

(1) random data generators - this is about to become the major API category, which is suitable for writing test drivers that are model checker aware. The idea is to obtain non-deterministic input data values from JPF in a way that it can systematically analyze all relevant values. Currently, this is restricted to complete enumerations, and hence is only avaliable for boolean and int values via the following methods:

```
public static boolean randomBool ();
public static int (random (int max);
```

but this will be extended towards heuristics that can be chosen application specific, i.e. generators that do only produce certain values based on the associated heuristic (e.g. threshold values for floats and doubles). These APIs are used to initialize test driver data like

```
import gov.nasa.jpf.jvm.Verify;
..
void test (..) {
  ..
  int data = Verify.random(3); // JPF will execute for values [0,1,2,3]
  ..
}
```

(2) search constraints - this category can be used to control the JPF search process. While this is problematic in terms of missing potential defects, it is often the only way to constrain the state space so that JPF can verify a given application. There are currently two instances in this category: atomicity control and search pruning

```
Verify.beginAtomic();
... // all code in here is executed by JPF in one transition
Verify.endAtomic();
```

Direct atomicity control was mainly used before the automatic, on-the-fly partial order reduction (POR) was implemented, and only remains relevant for applications that are (still) problematic with respect to POR. This especially includes frequent access to reachable, but not visible fields in concurrent programs (i.e. there is a reference chain that makes the object reachable from different threads, but the corresponding fields are private or protected, hence not visible for all threads). In general, the role of explicit atomicity control will be further reduced by future POR extensions.

Search pruning is useful for highly application specific properties, where it is obvious that certain values are not of interest with respect to the property.

```
// ..compute some data..
Verify.ignoreIf(data > someValue); // if true, JPF will not further analyze, but
backtrack
// ..do some stuff with data..
```

If the provided expression evaluates to true, JPF does not continue to execute the current path, and backtracks to the previous non-deterministic choice point.

(3) state annotation - based on certain value combinations, an application might give JPF hints about the relevance of an program state that can be subsequently used by Search and/or Heuristic implementations.

```
// ..compute some data
Verify.interesting( data < someValue );
// ..do some stuff with data
```

This does not stop execution by JPF, but stores an 'interesting' attribute for the current state. It's more general version is used to attach arbitrary strings to states:

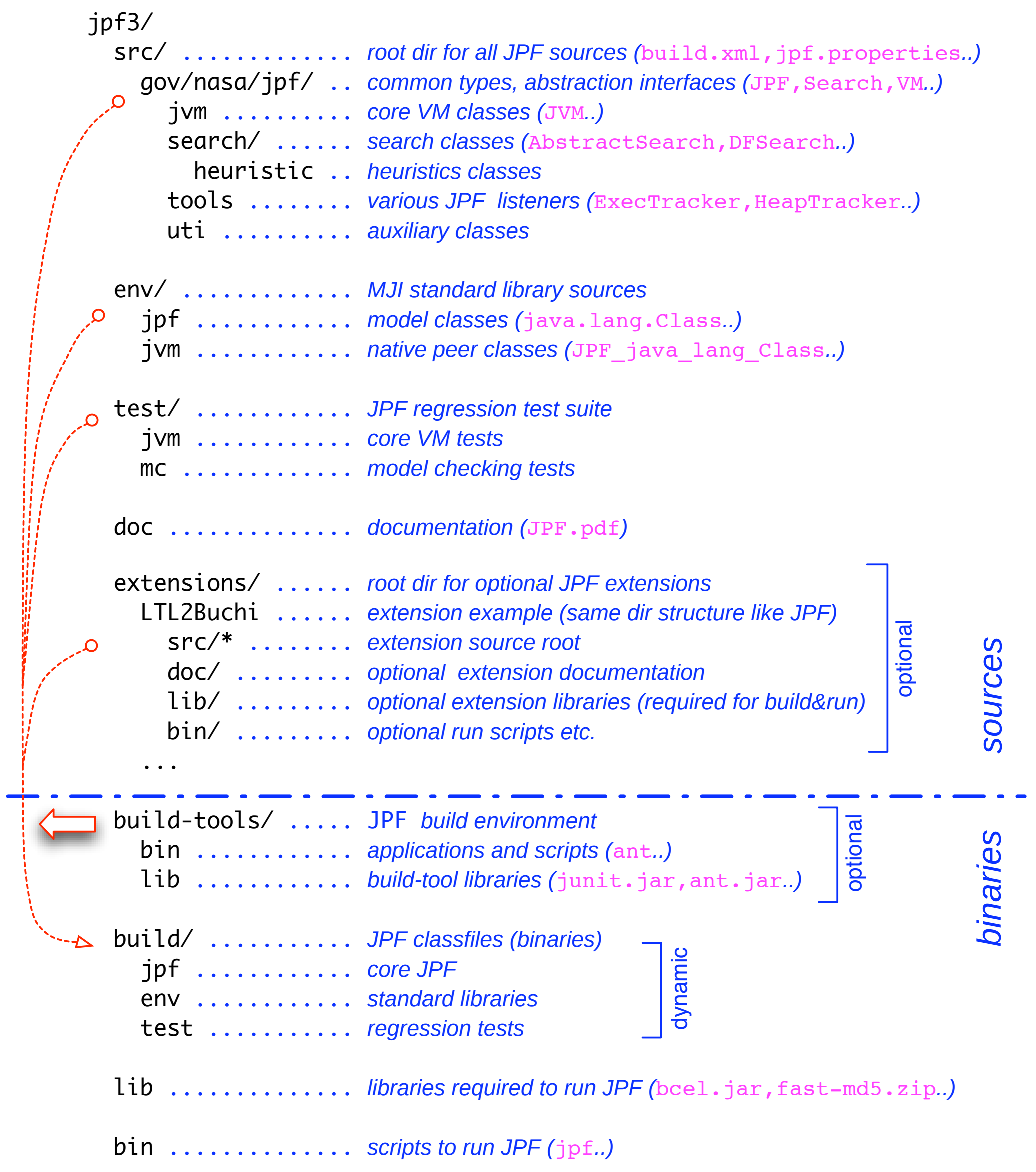
```
// ..compute some data
if (data < someValue) {
  Verify.setAnnotation("critical data value");
  // ..do some stuff with dat
```

Again, this category is about to become less important since Search- and VMListeners are superior mechanisms to store not just strings, but arbitrary objects that refer to specific states.

(4) verification log output - this is the most simple category, which is used to differentiate between normal program output (that is executed and analyzed by JPF), and output that is strictly verification relevant, i.e. should not appear when executing a program outside JPF. Not very surprising, it contains a number of print(..) methods.

Other, more exotic Verify methods support collecting information during JPF execution, which is persistent and can be later-on queried by JPF embedding code (programs that execute JPF). This uses an MJI trick where the native peer class (JPF_gov_nasa_jpf_jvm_Verify) is used to set some data during JPF execution, which can be later-on retrieved by model class (gov.nasa.jpf.jvm.Verify) code that is executed outside of JPF. This is currently used to implement counters, which in turn are used to verify JPF itself.

It should be noted that while most of the Verify APIs have alternative implementations that enable execution outside of JPF, applications using imp2 /Gs2 u3. 8a(o421 0 0259.081 411 Tm (erify)Tj 11 0 0 -11 281.9880 -12ie80la



```
jpf3/
  src/ ............. root dir for all JPF sources (build.xml,jpf.properties..)
    gov/nasa/jpf/ .. common types, abstraction interfaces (JPF,Search,VM..)
      jvm .......... core VM classes (JVM..)
      search/ ...... search classes (AbstractSearch,DFSearch..)
        heuristic .. heuristic classes
      tools ........ various JPF  listeners (ExecTracker,HeapTracker..)
      uti .......... auxiliary classes

  env/ ............. MJI standard library sources
    jpf ............ model classes (java.lang.Class..)
    jvm ............ native peer classes (JPF_java_lang_Class..)

  test/ ............ JPF regression test suite
    jvm ............ core VM tests
    mc ............. model checking tests

  doc .............. documentation (JPF.pdf)

  extensions/ ...... root dir for optional JPF extensions                      ┐
    LTL2Buchi ...... extension example (same dir structure like JPF)          │
      src/* ........ extension source root                                    │ optional
      doc/ ......... optional  extension documentation                        │          sources
      lib/ ......... optional extension libraries (required for build&run)    │
      bin/ ......... optional run scripts etc.                                ┘
    ...
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  build-tools/ ..... JPF  build environment                     ┐
    bin ............ applications and scripts (ant..)           │ optional
    lib ............ build-tool libraries (junit.jar,ant.jar..) ┘
                                                                          binaries
  build/ ........... JPF classfiles (binaries)  ┐
    jpf ............ core JPF                   │
    env ............ standard libraries         │ dynamic
    test ........... regression tests           ┘

  lib .............. libraries required to run JPF (bcel.jar,fast-md5.zip..)

  bin .............. scripts to run JPF (jpf..)
```

To enable builds of JPF outside an integrated development environment, the JPF distribution contains an optional directory tree that contains everything that is required to compile JPF sources from a command line. Using these tools, a build directory is created that holds the class files for the three major source directories.

The lib directory contents are required to run JPF, and need to be in the CLASSPATH if JPF is started directly, i.e. without the provided scripts.

To ease JPF execution, the bin directory contains scripts to automatically set the CLASSPATH. The script bin/jpf can be used to start JPF from the command line like a normal Java VM (i.e. is a java drop-in replacement).

Building JPF from the Command Line

The apache.org Ant system is used to manually build JPF. The toplevel JPF directory contains a build.xml file with all required configuration. Either the ant provided in the optional build-tools module or any recent external version obtained from ant.apache.org (1.6.2 as of this writing) can be used. To list supported targets, type

```
$ build-tools/bin/ant -projecthelp
Buildfile: build.xml
Main targets:
 compile            compile JPF and its specific (modeled) environment libraries
 compile-env-jpf    compile MJI model classes
 compile-env-jvm    compile MJI native peer classes
 compile-examples   compile examples
 compile-ext        compile optional extension classes
 compile-jpf        compile JPF core classes
 compile-tests      compile all the tests for JPF
 dist               generate the compressed distribution tar files
 docs-javadoc       create javadoc documentation
 init               common task/target initialization
 jar                create jar archives for JPF, its JVM and their environment models
 run-tests          run all JPF tests
Default target: compile
```

This target set might change in the future. To erase the jpf/build directory, and start from a clean directory structure, type

```
$ build-tools/bin/ant clean
```

If junit is installed, the preferred way to build from scratch is

```
$ build-tools/bin/ant run-tests
Buildfile: build.xml
init:
    [echo] ****************** JPF build system ********************
    [echo] current dir:      /Users/pcmehlitz/projects/jpf3
    [echo] user home dir:    /Users/pcmehlitz
    [echo] classpath: :build-tools/bin/../lib/ant.jar: ...
    [echo]
    [echo] java version:     1.4.2_05
    [echo] OS:               Mac OS X-ppc-10.3.8
    ...
compile-jpf: ...
compile-env-jvm: ...
compile-env-jpf: ...
compile: ...
compile-tests: ...
   [javac] Compiling 34 source files to /Users/pcmehlitz/projects/jpf3/build/..
run-tests:
    [echo] --- running Junit tests from build/test
   [junit] Running gov.nasa.jpf.jvm.TestArrayJPF
   [junit]   running jpf with args: gov.nasa.jpf.jvm.TestArray test2DArray
   ...
   [junit] Tests run: 6, Failures: 0, Errors: 0, Time elapsed: 6.126 sec
   ...
BUILD SUCCESSFUL
Total time: 1 minute 25 seconds
```

This should compile all sources (except of examples) and then run the regression test suite.

JPF and Eclipse

JPF compiles and runs inside of the Eclipse IDE. To import, create a new Eclipse Java project from CVS or an external location, and make sure all libraries from jpf/lib are in the Eclipse build path of the project Properties dialog (they should appear under the "Libraries" tab). The default output folder should be set to build/jpf

Please note that the above example includes libraries of optional JPF extensions (LTL and symbolic execution).

The Compiler settings in either the project or the workspace Properties dialog should have the "JDK Compliance" (tab "Compliance and Classfiles", settings "compiler compliance level" "generated class files compatibility" and "source compatibility") set to "1.4".

Beyond this, the only caveat for building JPF is the inclusion of the env/jpf source directory. Classes compiled from this location are MJI model classes of standard Java library components, that are only meant to be seen by JPF, not the host VM. If this directory appears in the Eclipse source path, Eclipse will automatically compile and use it for the rest its build process, which might create problems if the relevant classes (e.g. java.lang.Class, java.lang.Thread) do not support certain features found in the real library class. There are two solutions to this problem:

(a) don't include env/jpf in the source path (but this makes the model class sources unavailable when editing/compiling their native peer counterparts in env/jvm).

(b) add missing model class methods and fields, so that Eclipse can build the JPF classes using these features. Keep in mind that all JPF classes other than env/jpf will be executed by the host VM, and not JPF.

To run JPF from inside Eclipse, specify gov.nasa.jpf.JPF as the Main class inside the Run dialog (tab "Main"), use the default working directory (tab "Arguments"), specify the target application main class as the program argument, and make sure to include the jpf default classpath under user entries (tab "Classpath"). If the application under test resides outside the jpf directory tree, its class files of course have to be added to the user entries of the Classpath dialog tab.

The JPF Toplevel Structure

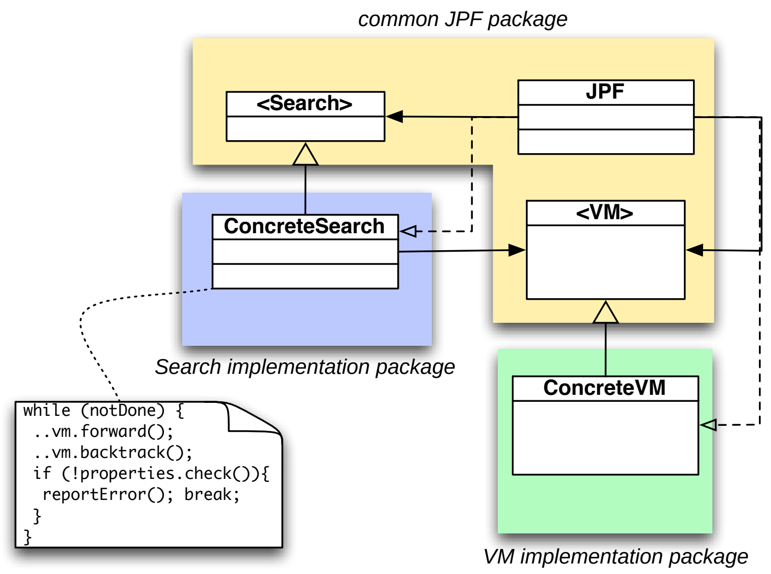
JPF was designed around two major abstractions: (1) the VM, and (2) the Search object.

(1) The **VM** is the execution environment specific state generator. By executing Java bytecode instructions, the VM generates state representations that can be

- checked for equality (has a state been visited before)
- queried (thread states, data values etc.)
- stored
- restored

Since Java is a inherently multithreaded execution environment, the main VM parameterization is the Scheduler type, which constitutes a strategy object to select and systematically explore valid thread scheduling sequences. There are three major VM methods in the context of the VM-Search collaboration

- forward - generate the next state, report if the generated state has a successor. If yes, store on a backtrack stack for efficient restoration.
- backtrack - restore the last state on the backtrack stack
- restoreState - restore a arbitrary state (not necessarily on the backtrack stack)



```
while (notDone) {
 ..vm.forward();
 ..vm.backtrack();
 if (!properties.check()){
  reportError(); break;
 }
}
```

(2) The **Search** object is responsible for selecting the state from which the VM should proceed, either by directing the VM to generate the next state (forward), or by telling it to backtrack to a previously generated one. Search objects can be thought of as drivers for VM objects.

Search objects also configure and evaluate property objects (e.g. NotDeadlockedProperty, NoAssertionsViolatedProperty). The main Search implementations include a simple depth-first search (DFSearch), and a priority-queue based search that can be parameterized to do various search types based on selecting the most interesting state out of the collection of all successors of a given state (HeuristicSearch). A Search implementation mainly provides a single search method, which includes the main loop that iterates through the relevant state space until it has been completely explored, or the search found a property violation.

On-the-fly Partial Order Reduction

The number of different scheduling combinations is the prevalent factor for the state space size of concurrent programs. Fortunately, for most practical purposes it is not necessary to explore all possible instruction interleavings for all threads. The number of scheduling induced states can be significantly reduced by grouping all instruction sequences in a thread that cannot have effects outside this thread itself, collapsing them into a single transition. This technique is called Partial Order Reduction (POR), and typically results in more than 70% reduction of state spaces.

JPF employs an on-the-fly POR that does not rely on user instrumentation or static analysis. JPF automatically determines at runtime which instructions have to be treated as state transition boundaries. If POR is enabled (configured via vm.por property), a forward request to the VM executes all instructions in the current thread until one of the following conditions is met:

1. the next instruction is scheduling relevant

2. the current thread is not runnable anymore (e.g. waiting for a signal)
3. the next instruction is nondeterministic

Detection of scheduling relevance is delegated to the instruction object itself, passing down information about the current VM execution state and threading context.

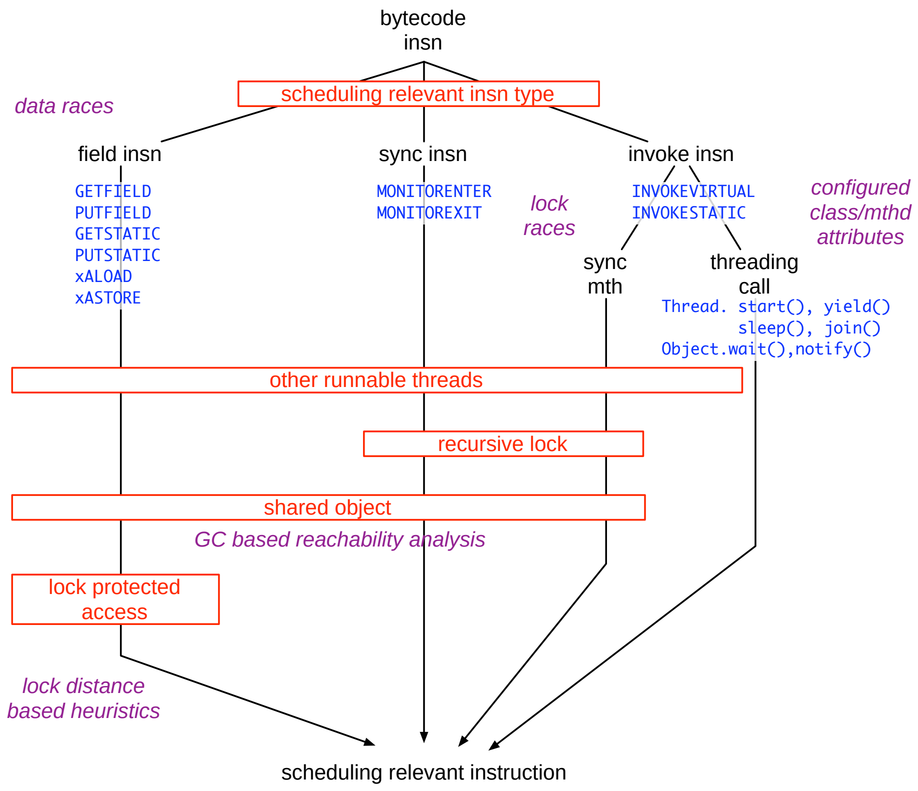
```
executeStep () {
  while (true) {
    Instruction nextInsn = executeInstruction();
    ..
    if (nextInsn.isSchedulingRelevant(<threading-context>)  // (1)
         || !currentThread.isRunnable()                      // (2)
         || nextInsn.isNonDeterministic())                   // (3)
      break;
    ..
  }
}
```

Each bytecode instruction type corresponds to a concrete gov.nasa.jpf.Instruction subclass, that returns scheduling relevance based on the following factors:

Instruction Type - due to the stack based nature of the JVM, only about 10% of the Java bytecode instructions are scheduling relevant, i.e. can have effects across thread boundaries. The interesting instructions include direct synchronization (monitorEnter, monitorexit, invokeX on synchronized methods), field access (putX, getX), array element access (Xaload, Xastore), and invoke calls of certain Thread (start(), sleep(), yield(), join()) and Object methods (wait(), notify()).

Object Reachability - besides direct synchronization instructions, field access is the major type of interaction between threads. However, not all putX / getX instructions have to be considered, only the ones referring to objects that are reachable by at least two threads can cause data races. While reachability analysis is an expensive operation, the VM already performs a similiar task during garbage collection, which is extended to support POR.

Thread and Lock Information - even if the instruction type and the object reachability suggest scheduling relevance, there is no need to break the current transition in case there is no other runnable thread. In addition, lock acquisition and release (monitorenter, monitorexit) do not have to be considered as transition boundaries if there they happen recursively - only the first and the last lock operation can lead to rescheduling.



While JPF uses these informations to automatically deduce scheduling relevance, there exist three mechanisms to explicitly control transition boundaries (i.e. potential thread interleavings)

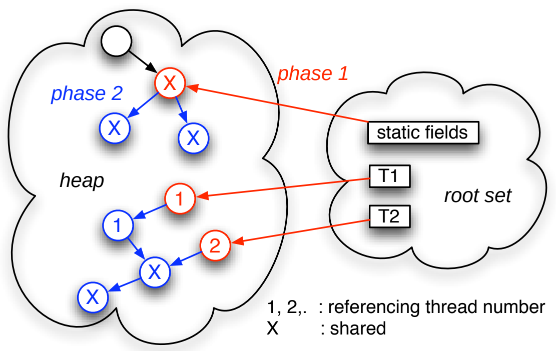
Attributor - a configurable concrete class of this type is used by JPF during class loading to determine object, method and field attributes of selected classes and class sets. The most important attributes with respect to POR are method atomicity and scheduling relevance levels: (a) never relevant, (b) always scheduling relevant, (c) only relevant in the context of other runnables. (d) only relevant of toplevel lock.

VMListener - a listener can explicitly request a reschedule by calling ThreadInfo.yield() in response of a instruction execution notification

Verify - the Verify class serves as an API to communicate between the test application and JPF, and contains beginAtomic(), endAtomic() functions to control thread interleaving

The main effort of JPFs POR support relates to extending its precise mark and sweep collector. POR reachability is a subset of collector reachability, hence the mechanism piggybacks on the mark phase object traversal. It is complicated by the fact that certain reference chains exist only in the (hidden) VM implementatiion layer. For instance, every thread has a reference to its ThreadGroup, and the ThreadGroup objects in turn have references to all included threads, hence - from a garbage collection perspective - all threads within a group are mutually reachable. If the application under test does not use Java reflection and runtime queries like thread enumeration, POR reachability should follow accessibility rules as closely as possible. While JPF's POR does not yet support protected and private access modifiers, it includes a mechanism to specify that certain fields should not be used to promote POR reachability. This attribute is set via the configured Attributor at class load time.

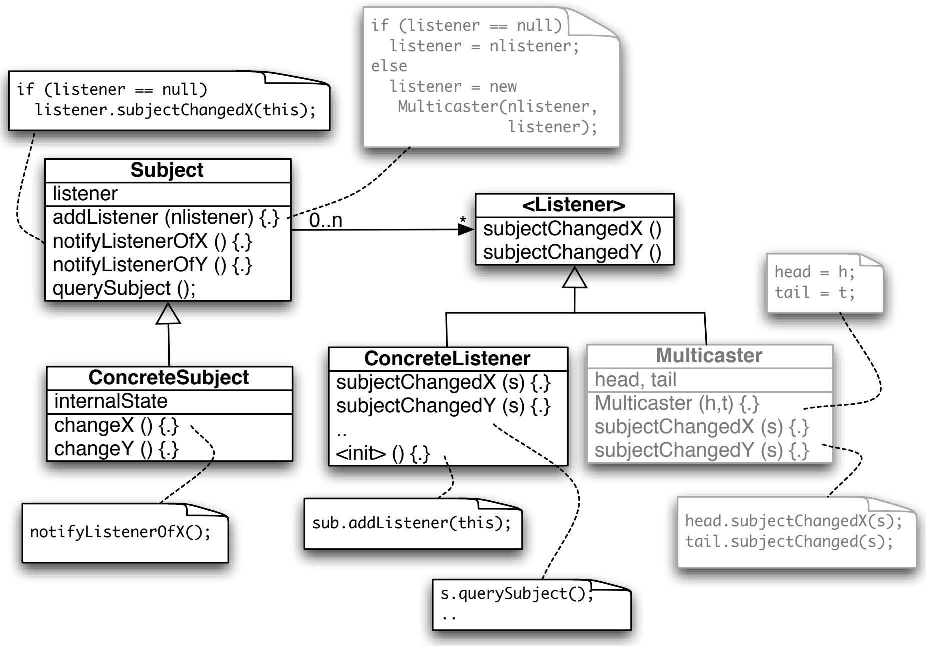
With this mechanism, calculating POR reachability becomes a straight forward approach that is divided into two phases. Phase 1 non-recursively marks all objects of the root set (mostly static fields and thread stacks), recording the id of the referencing thread. In case an object is reachable from a static field, or from two threads, it's status is set to shared. Phase 2 recursively traverses all heap objects, propagating either a set shared status or the referencing thread id through all reference fields that are not marked as reachability firewalls. Again, if the traversal hits an object that is already marked as referenced by another thread, it promotes the object status to shared, and from there propagates the shared status instead of the thread id.



Search- and VMListeners

Purpose

Beyond this basic Search-VM collaboration, there are numerous potential variations, e.g. to gather statistics, to monitor the state exploration progress, or to query details of states like field values. These are typical tasks for programs that use JPF, and add certain functionality on top of it (e.g. a graphical user interface). The goal is to provide an extension mechanism in JPF that enables adding such functionality without modifying Search or VM implementations



```
if (listener == null)
  listener.subjectChangedX(this);
```

```
if (listener == null)
  listener = nlistener;
else
  listener = new
   Multicaster(nlistener,
               listener);
```

```
head = h;
tail = t;
```

```
notifyListenerOfX();
```

```
sub.addListener(this);
```

```
s.querySubject();
..
```

```
head.subjectChangedX(s);
tail.subjectChanged(s);
```

The required extensibility is achieved by means of a Listener pattern (a Observer variant with a wide, change-topic specific notification interface), i.e. Listener instances register themselves either with the Search and/or the VM object (Subject), get notified when their corresponding Subjects perform certain operations, and can then interact with the Subject to query additional information, or even control the successive Subject behavior.

Changed facets of the Subjects are mapped into separate Observer methods, passing in the corresponding Subject instance as a parameter. As a implementation detail, Subjects keep track of registered listeners via so called MultiCasters (linked lists consisting of nodes implementing the listener interface), to avoid runtime costs for container traversal, which is suitable for high frequent notifications with small numbers of listeners.

Both interfaces reside in the general gov.nasa.jpf directory, hence we avoid using parameters which expose underlying Search or VM implementation constructs (like ThreadInfo etc.), and rely on Listeners residing in the right package to access detailed information by casting the Subject to its implementation class. In general, we avoid interface methods with varying degrees of specialization, i.e. don't provide several notifications based on the same event if

- the specialization can be detected / queried by the Listener (e.g. method call and instruction execution)
- there is no non-observable symmetric notification, e.g. visible object creation (NEW instruction executed), but invisible object destruction (garbage collection)

There are three different levels of Subject information retrieval by listener implementations:

1. Generic - listener resides outside any JPF package and just uses the information that is publicly available via gov.nasa.jpf.Search / VM (potentially using other gov.nasa.jpf classes and interfaces)

2. Search-specific - listener resides outside JPF packages but casts Subject notification parameter (Search or VM) to concrete implementation (e.g. gov.nasa.jpf.search.heuristic.BFSHeuristic), using its public API to retrieve implementation-specific information

3. Internal - listener resides in concrete Subject implementation package (e.g. gov.nasa.jpf.jvm), accessing package private information

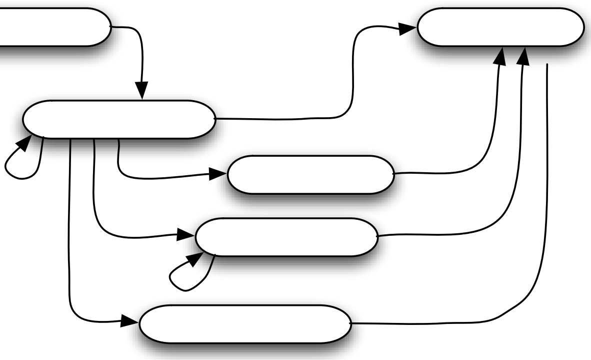
SearchListener

SearchListener instances are used to monitor the state space search process, e.g. to create graphical representations of the state-graph. They provide notification methods for all major Search actions.

```
public interface SearchListener {

 /* got the next state */
 void stateAdvanced (Search search);

 /* state was backtracked one step */
 void stateBacktracked (Search search);

 /* a previously generated state was restored
    (can be on a completely different path) */
 void stateRestored (Search search);

 /* JPF encountered a property violation */
 void propertyViolated (Search search);

 /* we get this after we enter the search loop, but BEFORE the
    first forward */
 void searchStarted (Search search);

 /* there was some contraint hit in the search, we back out
```

```
    could have been turned into a property, but usually is an attribute of
    the search, not the application */
 void searchConstraintHit (Search search);

 /* we're done, either with or without a preceeding error */
 void searchFinished (Search search);
}
```

For the standard depth first search (gov.nasa.jpf.search.DFSearch), listener implementations can assume the following notification model:



E xample

Using this interface, a sample application to log a generic search progress can be programmed like this

```
import gov.nasa.jpf.JPF;
import gov.nasa.jpf.Transition;
import gov.nasa.jpf.SearchListener;
import gov.nasa.jpf.Search;

public class TestClient implements SearchListener {

  /***************************************** main entry: driver ***************/

  public static void main (String[] args) {

    TestClient listener = new TestClient();

    Config conf = JPF.createConfig(args);
    // add your own args
    conf.setProperty(“jpf.print_exception_stack”, “true”); //..

    JPF jpf = new JPF(conf);
    jpf.addSearchListener(listener);

    System.out.println("---------------- JPF started”);
    jpf.run();
    System.out.println("---------------- JPF terminated");
  }


  /****************************************** SearchListener notifications *****/
  public void stateRestored(Search search) {
    log( "restore ", search);
  }

  public void stateBacktracked (Search search) {
    log( "back ", search);
  }

  public void searchStarted (Search search) {
    System.out.println(? search started");
  }

  public void searchFinished (Search search) {
    System.out.println(" search finished");
  }

  public void propertyViolated (Search search) {
    ErrorList errors = search.getErrors();
    for (int i=0; i <errors.size(); i++) {
      System.out.println(?property violated: ? + errors.getError(i));
    }
  }

  public void searchConstraintHit(Search search) {
    System.out.println(?constraint hit ? + search.getConstraint();
  }

  public void stateAdvanced (Search search) {
    log( search.hasNextState() ? "> " : "* ", search);
  }

  /************************************************* helper methods *************/

  private void log (String prefix, Search search) {
    Transition trans = search.getTransition();     // acquire last transition
                                                   // (list of executed insns)
    if (trans != null) {
      System.out.print(prefix);
      System.out.print( trans.getThread());        // get current thread number
      System.out.print(" ");
      System.out.print( search.getStateNumber());  // get unique state id
      System.out.print(" ");
      System.out.print( search.getSearchDepth());  // get current search depth
      System.out.print("  : ");
      System.out.println( trans.getLabel());
    }
  }
}
```

More elaborate examples of SearchListeners can be found in test/gov/nasa/jpf/tools, which also includes StateSpaceDot, a generator for GraphViz specific graph descriptions (Dot files) to generate state graph images.

VMListener

VMListeners are used to follow the detailed VM processing, e.g. to monitor certain execution environment specific instructions (like Java IF instructions for coverage analysis, or PUTFIELD, GETFIELD instructions for potential race detections).

```
 public interface VMListener {
 /* VM has executed next instruction
     (can be used to analyze branches, monitor PUTFIELD / GETFIELD and
     INVOKExx / RETURN instructions) */
  void instructionExecuted (VM vm);

  /* new Thread entered run() method */
  void threadStarted (VM vm);

  /* Thread exited run() method */
  void threadTerminated (VM vm);

  /* new class was loaded */
  void classLoaded (VM vm);

  /* new object was created */
  void objectCreated (VM vm);

  /* object was garbage collected (after potential finalization) */
  void objectReleased (VM vm);

  /* garbage collection mark phase started */
  void gcBegin (VM vm);

  /* garbage collection sweep phase terminated */
  void gcEnd (VM vm);

  /* exception was thrown */
  void exceptionThrown (VM vm);
}
```

VMListeners usually do reside in JPF implementation packages, and are mainly intended to be a internal, non-intrusive JPF extension mechanism, e.g. to provide additional information for specific Search implementations. The reason for this restriction is that VM is a very coarse abstraction of its potential implementors, and we do not want to ?bubble up? abstration types of execution environment specific classes into the common gov.nasa.jpf package.

Configuration

Listener configuration can be done in two ways: (a) per configuration file, and (b) dynamic. In both cases, we have to distinguish between separate and combined listener instances.

(a) configuration file - there are three property entries that can be used to set listeners

- listener - for instances that are both VM and SearchListeners
- search.listener - SearchListener instances only
- vm.listener - VMListener instances only

All entries contain optional lists of colon separated,fully qualified listener class names, e.g.

```
listener=x.y.MyFirstListener:x.z.MySecondListener
```

(b) dynamic configuration - is usually done by applications that run JPF embedded

```
MyListener listener= new MyListener(..);
..
Config config = JPF.createConfig( args);

JPF jpf = new JPF( config);
jpf.addSearchListener(listener);
jpf.addVMListener( listener);
jpf.run();
..
```

Most listeners tend to fall into three major categories: (a) system class (e.g. for logging), (b) complex properties, and (c) JPF debugging. The first category (a) is usually configured via the default.properties, (b) is configured with an application specific mode property file, (c) is specified via the command line ('+key=value' overrides).

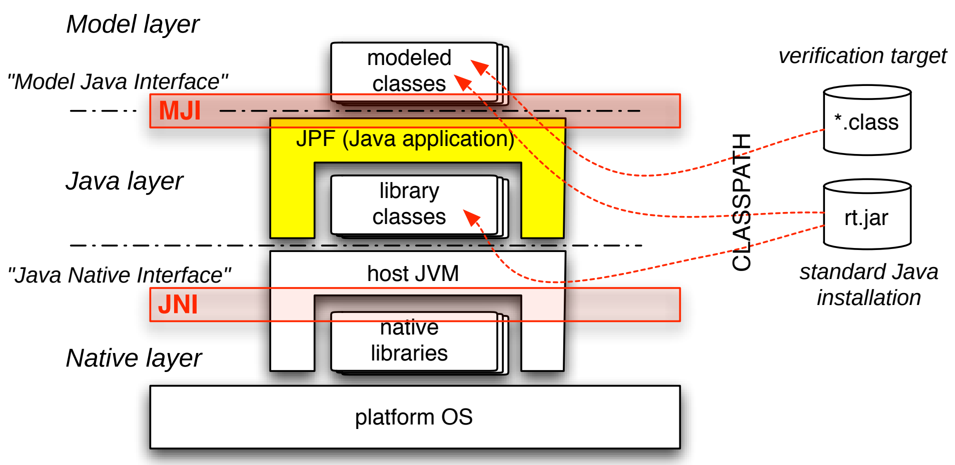
The Model Java Interface

Purpose

Even if it is just a Java application (i.e. solely consists of Java classes), JPF can be viewed as a Java Virtual Machine (JVM) in itself. The consequence is that (*.class) classfiles, and even the same files at times, are processed in two different ways in a JVM running JPF

- as ordinary Java classes managed and executed by the host JVM (standard Java library classes, JPF implementation classes)
- as "modeled" classes managed and processed (verified) by JPF

Class lookup in both layers is based on the CLASSPATH environment variable / command line parameter, but this should not obfuscate the fact that we have to clearly distinguish between these two modes. In particular, JPF (i.e. the "Model" layer) has its own class and object model, which is completely different and incompatible to the (hidden) class and object models of the underlying host JVM executing JPF



Each standard JVM supports a so called "Java Native Interface" (JNI), that is used to delegate execution from the Java level (i.e. JVM controlled bytecode) down into the (platform dependent) native layer (machine code). This is normally used to interface certain functionalities to the platform OS / architecture (e.g. I/O or graphics).

Interestingly enough, there exists a analogous need to lower the "execution" level in JPF, from JPF controlled bytecode into JVM controlled bytecode. According to this analogy, the JPF specific interface is called "Model Java interface" (MJI).

Even though MJI offers a wide range of applications, there are three major usages for delegating bytecode execution into the host JVM

(1) Interception of native methods - without a abstraction lowering mechanism, JPF would be forced to completely ignore native methods, i.e. would fail on applications relying on the side effects of such methods, which is not acceptable (even if many native methods indeed can be ignored if we restrict the set of verification targets)

(2) Interfacing of JPF system level functionality - some system level functions of standard library classes (esp. java.lang.Class, java.lang.Thread) have to be intercepted even if they are not native because they have to affect the JPF internal class, object and thread model (etc. loading classes, creating / starting threads). It should be noted that MJI can also be used to extend the functionality of JPF without changing its implementation.

(3) State space reduction - by delegating bytecode execution into the non-state-tracked host JVM, we can cut off large parts of the state space, provided that we know the corresponding method side effects are not relevant for property verification (e.g. System.out.println(..))

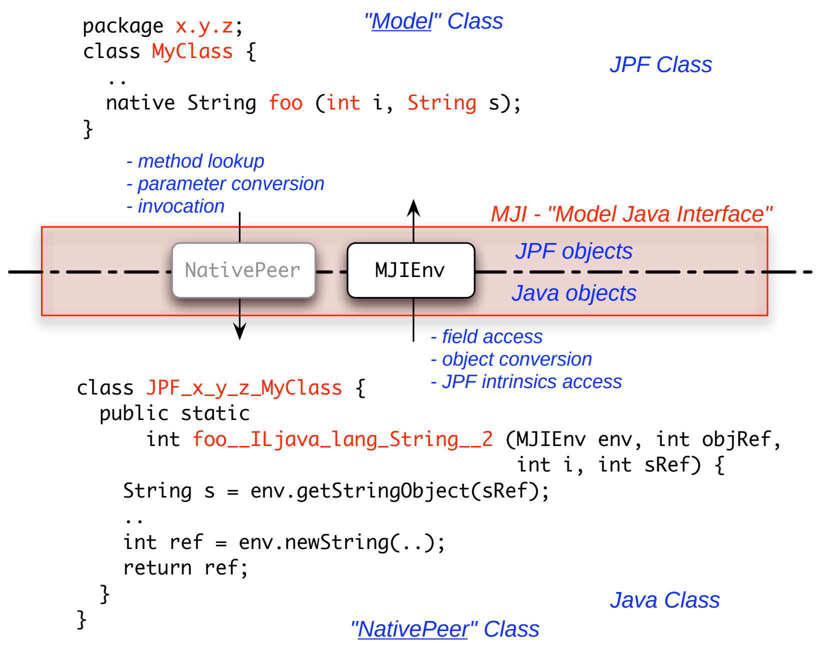
Besided these standard usages, there exist more exotic applications like collecting information about JPF state space exploration and making it available both to JPF and the verification target.

MJI Components

The basic functionality of MJI consists of a mechanism to intercept method invocations, and delegate them by means of Java reflection calls to dedicated classes. There are two types of classes involved, residing in different layers:

- Model Class - this is the class executed by JPF, which might be completely unknown to the host JVM
- Native Peer Class - this is the class containing the implementations of the methods to intercept, and to execute in the host JVM

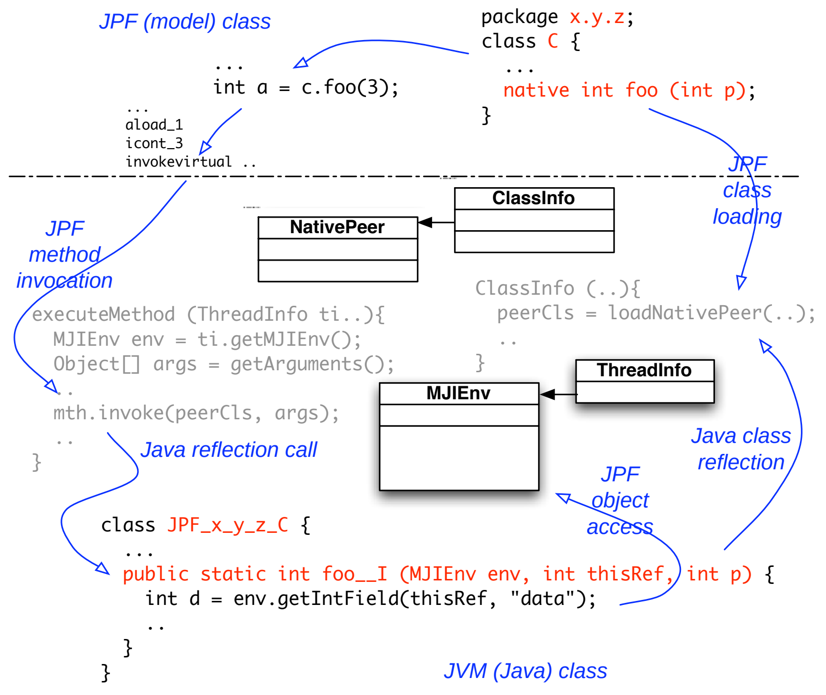
As part of the JPF implementation, MJI automatically takes care of determining which method invocations have to be intercepted, looking up the corresponding



```
package x.y.z;
class MyClass {
  ..
  native String foo (int i, String s);
}
```

```
class JPF_x_y_z_MyClass {
  public static
      int foo__ILjava_lang_String__2 (MJIEnv env, int objRef,
                                      int i, int sRef) {
    String s = env.getStringObject(sRef);
    ..
    int ref = env.newString(..);
    return ref;
  }
}
```

But this would be not very useful without being able to access the JPF object model (or other JPF intrinsics), from inside the NativePeer methods. Instead of requiring all NativePeer implementations to reside in a JPF internal package, there exists a interface class MJIEnv that can be used to get back to JPF in a controlled way. NativePeers residing in gov.nasa.jpf.jvm (i.e. the same package like MJIEnv) can basically reach all internal JPF features. Outside this package, the available API in MJIEnv is mostly restricted to the access JPF object (getting and setting values).



```
package x.y.z;
class C {
  ...
  native int foo (int p);
}
```

```
...
int a = c.foo(3);
```

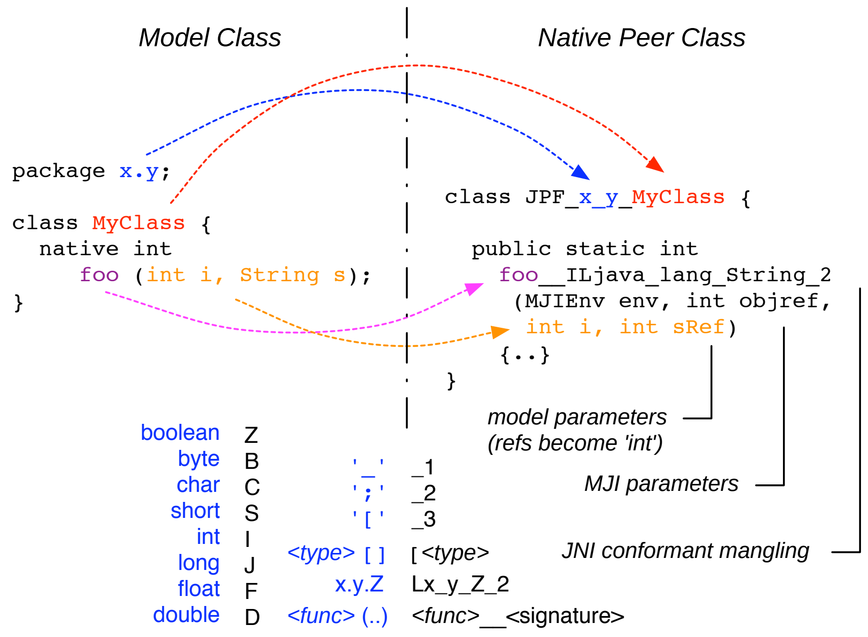
```
...
aload_1
icont_3
invokevirtual ..
```

```
ClassInfo (..){
  peerCls = loadNativePeer(..);
  ..
}
```

```
executeMethod (ThreadInfo ti..){
  MJIEnv env = ti.getMJIEnv();
  Object[] args = getArguments();
  ..
  mth.invoke(peerCls, args);
  ..
}
```

```
class JPF_x_y_z_C {
  ...
  public static int foo__I (MJIEnv env, int thisRef, int p) {
    int d = env.getIntField(thisRef, "data");
    ..
  }
}
```

Before a NativePeer method can be used, JPF has to establish the correspondence between the Model Class and the NativePeer. This takes place at load time of the Model Class. MJI uses a special name mangling scheme to lookup NativePeers, using the Model Class package name and class name to deduce the NativePeer class name.



Since the model class package is encoded in the NativePeer class name, the package of the NativePeer can be choosen freely. In analogy to JNI, NativePeer method names include the signature of the model method by encoding its parameter types. If there is no potential ambiguity, i.e. mapping from NativePeer to model class methods is unique, signature encoding is not required.

All native methods in a NativePeer have to be "public static" - there is no correspondence between JPF and JVM objects. Instead, MJI automatically adds two parameters: MJIEnv and objRef (classRef in case of static Model Class methods). The MJIEnv object can be used to get back to JPF, the objRef is a handle for the corresponding JPF "this" object (or the java.lang.Class object in case of a static method).

Going beyond the JNI analogy, MJI can also be used to intercept

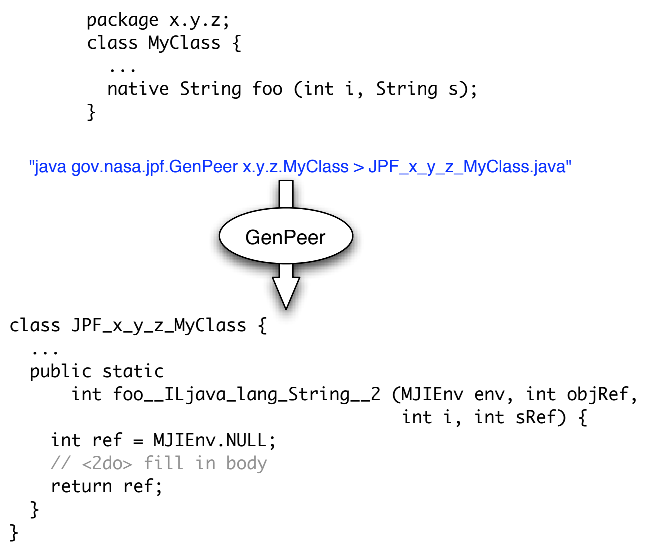
- non-native methods (i.e. the lookup process is driven by the methods found in the NativePeer, not the "native" attributes in the Model Class. This can be particularly useful in case the class is used from both as a Model Class and a JVM class (e.g. gov.nasa.jpf.jvm.Verify), using a method body that directly refers to the NativePeer class
- class initialization (the corresponding NativePeer method has to be named $clinit (MJIEnv env, int clsRef)
- constructors (the corresponding method name stem has to be $init__<sig>(MJIEnv env,int objRef, <ctor-params>) (normal signature mangling rules apply)

It is important to note that type correspondence does NOT include references. All references (object types) on the JPF side are transformed in handles (int values) on the JVM side. The passed in MJIEnv parameter has to be used to convert/analyze the JPF object. Since MJI per default uses the standad Java reflection call mechanism, there is a significant speed penalty (lookup, parameter conversion etc.), which again is a analogy to JNI.

Even if it is not directly related to MJI, it should be mentioned that some JPF specific Model Classes cannot be loaded via the CLASSPATH (e.g. java.lang.Class), since they contain JPF based code that is not compatible with the host JVM (e.g. relying on native methods that refer to JPF functionality). Such classes should be kept in separate directories / jars that are specified with the JPF command line option "-jpf-bootclasspath" or "-jpf-classpath". This is mostly the case for system classes. On the other hand, Model Classes don't have to be JPF specific. It is perfectly fine to provide a NativePeer for a standard Java class (e.g. java.lang.Character), if only certain methods from that standard class needs to be intercepted. NativePeer classes can contain any number of non-"native" methods and fields, but those should not be "public static" to avoid problems lookup problems.

Tools

To ease the tedious process of manually mangle method names, MJI includes a tool to automatically create skeletons of NativePeer classes from a given Model class, called "GenPeer". The translation process uses Java reflection, i.e. the Model Class needs to be in the CLASSPATH and is specified in normal dot notation (i.e. not as a file).



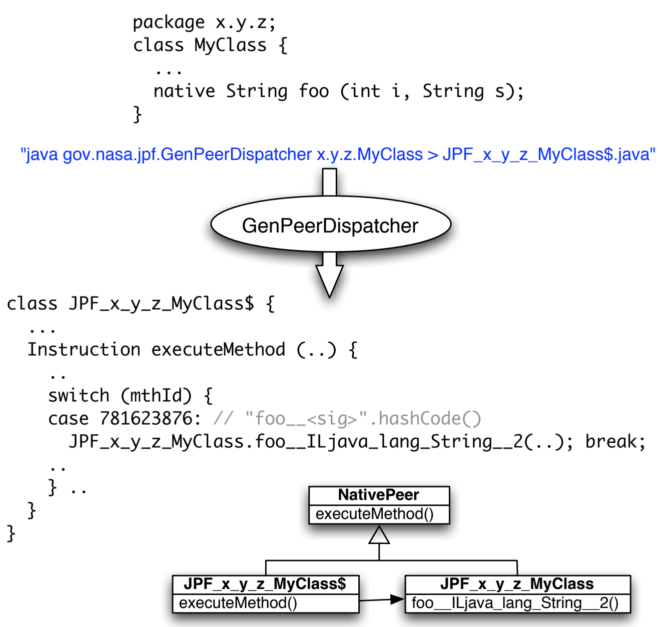
```
package x.y.z;
class MyClass {
  ...
  native String foo (int i, String s);
}
```

"java gov.nasa.jpf.GenPeer x.y.z.MyClass > JPF_x_y_z_MyClass.java"

```
class JPF_x_y_z_MyClass {
  ...
  public static
      int foo__ILjava_lang_String__2 (MJIEnv env, int objRef,
                                      int i, int sRef) {
    int ref = MJIEnv.NULL;
    // <2do> fill in body
    return ref;
  }
}
```

There exist a number of command line options that can be displayed by calling GenPeer without arguments. GenPeer per default writes to stdout, i.e. the output has to be redirected into a file.

Since NativePeer method invocations impose a significant (reflection call) overhead, there exists a mechanism to bypass the JVM method dispatching and the parameter conversion, called "NativePeer Dispatcher". This is implemented as a completely transparent NativePeer proxy that can be automatically generated by means of the GenPeerDispatcher tool



```
package x.y.z;
class MyClass {
  ...
  native String foo (int i, String s);
}
```

"java gov.nasa.jpf.GenPeerDispatcher x.y.z.MyClass > JPF_x_y_z_MyClass$.java"

```
class JPF_x_y_z_MyClass$ {
  ...
  Instruction executeMethod (..) {
    ..
    switch (mthId) {
    case 781623876: // "foo__<sig>".hashCode()
      JPF_x_y_z_MyClass.foo__ILjava_lang_String__2(..); break;
    ..
    } ..
  }
}
```

Depending on further improvements of target JVMs, this might not be required in the future and should only be considered for performance critical methods. The generated dispatcher class has the same name like the corresponding NativePeer, with a "$" suffix. It is looked up via the standard CLASSPATH (i.e. should be kept in the same directory / archive like the NativePeer).

Example

The following example is an excerpt of a JPF regression test, showing how to intercept various different method types, and using MJIEnv to access JPF objects.

Model class

```
public class TestNativePeer {
  static int sdata;

  static {
    // only here to be intercepted
  }

  int idata;

  TestNativePeer (int data) {
    // only here to be intercepted
  }

  public void testClInit () {
    if (sdata != 42) {
      throw new RuntimeException("native 'clinit' failed");
    }
  }

  public void testInit () {
    TestNativePeer t = new TestNativePeer(42);
    if (t.idata != 42) {
      throw new RuntimeException("native 'init' failed");
    }
  }

  native int nativeInstanceMethod (double d, char c, boolean b, int i);

  public void testNativeInstanceMethod () {
    int res = nativeInstanceMethod(2.0, '?', true, 40);
    if (res != 42) {
      throw new RuntimeException("native instance method failed");
    }
  }

  native long nativeStaticMethod (long l, String s);

  public void testNativeStaticMethod () {
    long res = nativeStaticMethod(40, "Blah");
    if (res != 42) {
      throw new RuntimeException("native instance method failed");
    }
  }

  native void nativeException ();

  public void testNativeException () {
    try {
      nativeException();
    } catch (UnsupportedOperationException ux) {
      String details = ux.getMessage();

      if ("caught me".equals(details)) {
        return;
      } else {
        throw new RuntimeException("wrong native exception details: " + details);
      }
    } catch (Throwable t) {
      throw new RuntimeException("wrong native exception type: " + t.getClass());
    }
    throw new RuntimeException("no native exception thrown");
  }
}
```

NativePeer class:

```
public class JPF_gov_nasa_jpf_jvm_TestNativePeer {

  public static void $clinit (MJIEnv env, int rcls) {
    env.setStaticIntField(rcls, "sdata", 42);
  }

  public static void $init__I (MJIEnv env, int robj, int i) {
    env.setIntField(robj, "idata", i);
  }

  public static int nativeInstanceMethod (MJIEnv env, int robj,
                                          double d, char c, boolean b, int i) {
    if ((d == 2.0) && (c == '?') && b) {
      return i + 2;
    }
    return 0;
  }

  public static long nativeStaticMethod (MJIEnv env, int rcls,
                                         long l, int stringRef) {
    String s = env.getStringObject(stringRef);
    if ("Blah".equals(s)) {
      return l + 2;
    }
    return 0;
  }

  public static void nativeException (MJIEnv env, int robj) {
    env.throwException("java.lang.UnsupportedOperationException", "caught me");
  }
}
```

Coding Conventions

JPF is an open system. In order to keep the source format reasonably consistent, we strive to keep the following minimal set of conventions

- 2 space indentation (no tabs)
- opening brackets in same line (class declaration, method declaration, control statements)
- no spaces after opening '(', or before closing ')'
- method declaration parameters indent on column
- all files start with copyright and license information
- all public class and method declarations have preceding Javadoc comments

The following code snippet illustrates these rules.

```
/* <copyright notice goes here>
 * <license referral goes here>
 */

/**
 * this is my class declaration example
 */

public class MyClass {
  /**
   * this is my public method example
   */
  public void foo (int arg1, int arg2,
                   int arg3) {
    if (bar) {
      ..
    }
  }
  ..
}
```

For convenience reasons, we include a jalopy.xml configuration file to format sources, but do not support it as a separate Ant target, to avoid accidental reformatting of a huge amount of CVS sources, and minimize dependencies for libraries required by the build process.

We consider modularity to be of greater importance than source format. With its new configuration scheme, there is no need to introduce dependencies of core classes towards optional extensions anymore. If you add something that is optional, and does not seamlessly fit into an existing directory, keep it separate by adding new directories. The core JPF classes should not contain any additional dependencies to external code.

JPF Related Papers

JPF has been both a research target and a system in use for a number of years. A broad collection of papers and reports is available, including the following list

Core Papers

"Model Checking Programs". W. Visser, K. Havelund, G. Brat, S. Park and F. Lerda. Automated Software Engineering Journal.Volume 10, Number 2, April 2003. <http://ase.arc.nasa.gov/jpf/ase00FinalJournal.pdf>

"Addressing Dynamic Issues of Program Model Checking". F. Lerda and W. Visser. Proccedings of SPIN2001. Toronto, May 2001. <http://ase.arc.nasa.gov/jpf/spin01.ps.gz>

Testing and Symbolic Execution

"Generalized Symbolic Execution for Model Checking and Testing" . S. Khurshid, C. S. Pasareanu and W. Visser. Proceedings of TACAS 2003. Warsaw, Poland, April 2003. <http://ase.arc.nasa.gov/jpf/tacas-symex.ps>

"Test Input Generation with Java PathFinder ". W. Visser, C. Pasareanu, S. Khurshid. Proceedings of ISSTA 2004. Boston, MA, July 2004. <http://ase.arc.nasa.gov/jpf/issta-visser.pdf>

"Verification of Java Programs Using Symbolic Execution and Invariant Generation". C. Pasareanu and W. Visser . Proceedings of SPIN 2004. Barcelona, Spian, April 2004 . LNCS 2989. <http://ase.arc.nasa.gov/jpf/spin04.ps>

"Experiments with Test Case Generation and Runtime Analysis" . C. Artho, D. Drusinsky, A. Goldberg, K. Havelund, M. Lowry, C. Pasareanu, G. Rosu, W. Visser. Proceedings of Abstract State Machines 2003 . Taormina, Italy, March 2003. LNCS 2589. <http://ase.arc.nasa.gov/jpf/asm03-invited.pdf>

Heuristic Search

"Heuristics for Model Checking Java Programs ". A. Groce and W. Visser. International Journal on Software Tools for Technology Transfer (STTT). To Appear 2004. <http://ase.arc.nasa.gov/jpf/stttheur.ps>

"Model Checking Java Programs using Structural Heuristics" . A. Groce and W. Visser. Proceedings of ISSTA 2002. Rome, Italy. July 2002. <http://ase.arc.nasa.gov/jpf/issta02_paper.ps>

"Heuristic Model Checking for Java Programs". A. Groce and W. Visser. Proceedings of SPIN 2002. Grenoble, France. April 2002 . <??>

Explaining Counter Examples

"What Went Wrong: Explaining Counterexamples". A. Groce and W. Visser. Proceedings of SPIN 2003. Portland, Oregon. May 2003. <??>

Using Java PathFinder

"Verifying Time Partitioning in the DEOS Scheduling Kernel" . J. Penix, W. Visser. C. Pasareanu, E. Engstrom, A. Larson and N. Weininger . Formal Methods in Systems Design Journal. To Appear 2004. <http://ase.arc.nasa.gov/jpf/final-Penix-Visser.ps>

"Experimental Evaluation of Verification and Validation Tools on Martian Rover Software". G. Brat, D. Drusinsky, D. Giannakopoulou, A. Goldberg, K. Havelund, M. Lowry, C. Pasareanu, A. Venet, R. Washington and W. Visser. Formal Methods in Systems Design Journal . Volume 25, Number 2-3, September 2004. <http://ase.arc.nasa.gov/jpf/fmsdjournal.pdf>

"Model Checking Multi-Agent Programs with CASP" . R. Bordini, M. Fisher, C. Pardavila, W. Visser, M. Wooldridge . Proceedings of CAV 2003. Boulder, Colorado, July 2003 . LNCS 2725. <http://ase.arc.nasa.gov/jpf/agentscav.ps>

"Assume-guarantee Verification of Source Code with Design-Level Assumptions". D. Giannakopoulou, C. S. Pasareanu, J. M. Cobleigh. Proceedings of the the 26th International Conference on Software Engineering (ICSE). Edinburgh, Scotland. May 2004. <http://ase.arc.nasa.gov/jpf/icse04.pdf>

Misc

"Program Model Checking as a New Trend". K. Havelund and W. Visser. International Journal on Software Tools for Technology Transfer (STTT). Volume 4, Number 1, October 2002. <http://ase.arc.nasa.gov/jpf/sttt-spin2000.pdf>

"Finding Feasible Abstract Counter-Examples" . C. Pasareanu, M. Dwyer and W. Visser. International Journal on Software Tools for Technology Transfer (STTT) Volume 5, Number 1, November 2003. <http://ase.arc.nasa.gov/jpf/tacas_sttt01.ps>

"Combining Static Analysis and Model Checking for Software Analysis". G. Brat and W. Visser. Proceedings of ASE2001. San Diego, November 2001. <http://ase.arc.nasa.gov/jpf/mcsa.ps.gz>

"Applying Predicate Abstraction to Model Check Object-Oriented Programs". W. Visser, S. Park and J. Penix. 3rd ACM SIGSOFT Workshop on Formal Methods in Software Practice. August 2000. <http://ase.arc.nasa.gov/jpf/fmsp00.ps.gz>