Wikiprint Book

Title: Trace Server user guide

Subject: Java Path Finder - projects/jpf-trace-server

Version: 9

Date: 03/15/2013 05:20:12 PM

Table of Contents

Trace Server user guide **3**
Contents 3
Description 3
Trace Emitter 3
Trace Filter 5
Trace Query 5
Trace Analyzer 6
Trace Printer 7
Console Trace Printer 7
Generic Console Trace Printer 8
Trace Report Shell Panel 9
How to download. 10
How to build. 10
Installation 10
Running Trace Server. 11
Running tests. 12
Repository 12
Trace Server as part of the JPF Summer Projects 12

Trace Server user guide

A JPF extension for storing the execution trace, by Igor Andjelkovic and Cyrille Artho.

For any information or to report problems, please contact:

Igor Andjelkovic <igor.andjelkovic "at" gmail.com>

Contents

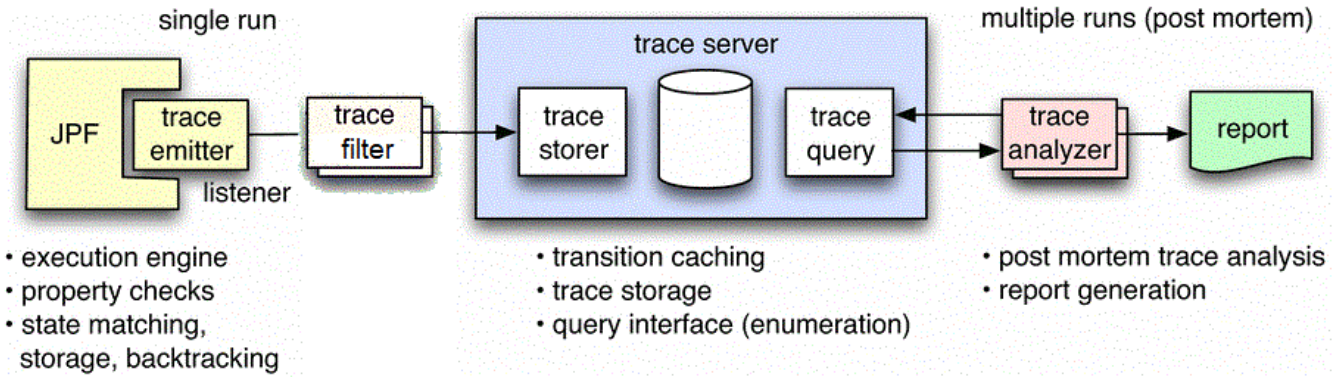
- Description
  - Trace Emitter
  - Trace Filter
  - Trace Query
  - Trace Analyzer
    Trace Printer
    - Console Trace Printer
    - Generic Console Trace Printer
  - Trace Report Shell Panel
- How to download
- How to build
- Installation
- Running trace server
- Running tests
- Repository
- Trace Server as part of the JPF Summer Projects

Description

During the execution of the SUT (System Under Test), JPF generates trace that consists of executed instructions and is kept in memory, competing with SUT heap and state storage. Production code SUT, can create trace that contain millions of steps (`Instruction` objects). The trace cannot be augmented with custom properties, nor trace output can be changed. In order to store custom trace data, one has to implement `gov.nasa.jpf.Listener`, create its own data structure and deal with state backtracking and restoration. Trace analysis is only possible "on-the-fly", i.e. when JPF is still running.

One possible solution of the problem described above is to configure a `trace server` interface that can listen on all JPF notifications (events) and stores them in a database. Once such database is created, post mortem analyzers can be used to find out about defects etc. Listeners like the `DeadlockAnalyzer` really should be implemented that way, since there is no need to run them while JPF is still searching for a deadlock. Post mortem analyzers does 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. JPF extensions would also benefit from the ability to augment traces with their custom information. The Trace Server framework should therefore be extensible enough to allow listeners to augment a trace with extra data.

Trace Server consists of several building blocks, shown at the picture below:



Trace Emitter

The entry point of the system is the Trace Emitter. It is responsible for system initialization and starting, and is an interface between JPF and the rest of the system.

The system can be configured by providing the following parameters to the used `.jpf` configuration file, e.g `runTraceServer.jpf`:

```
# trace emitter class, must be provided in order to store the trace
listener=gov.nasa.jpf.traceEmitter.DefaultTraceEmitter

# database location (folder), it will be created if it doesn't already exist.
# not used when trace_storer is not persistent, like "inMemory"
# it can be omitted, default value is "db", so database will be created in the current folder
traceServer.db_location = dbTrace

# use local(true) or remote(false) trace server
# default value is "true"
traceServer.local_storer = true

# trace storer type, "inMemory" or "neo4j"
# default value is "neo4j"
traceServer.trace_storer = inMemory

# can be used to start recording when SUT main() method is called
# initial instructions will be skipped
# default value is "true"
traceServer.skip_init = true

# remote server's name, not used when trace storer is local
# default value is "localhost"
traceServer.host = localhost

# remote server's port number, not used when trace storer is local
# default value is "4444"
traceServer.port = 4444
```

`Trace emitter` is responsible for creating event based on notifications from JPF listener. The base `trace emitter` stores only events that are necessary for creating the trace structure. In order to add more events to the trace, one has to extend the `gov.nasa.jpf.traceEmitter.TraceEmitter` and override the appropriate methods inherited from JPF listener interface. For example, `ObjInsnEmitter` allows `instructionExecute` and `objectLocked` events to be stored:

```
public class ObjInsnEmitter extends TraceEmitter {

 public ObjInsnEmitter(Config config, JPF jpf) {
   super(config, jpf);
 }

 public void instructionExecuted(JVM vm) {
   Instruction insn = vm.getLastInstruction();
   // we are skipping init instructions
   // recording starts when main() is called
   MethodInfo mi = insn.getMethodInfo();
   if (skipInit) {
     if (mi == miMain) {
       skipInit = false; // start recording
     } else {
       return; // skip
     }
   }
   Event event = this.createInstructionEvent(insn, mi,
       eventType.instructionExecuted);
   event.addProperty(PropertyCollection.INSTRUCTION_SOURCE_LINE,
       getLineString(insn));
```

```
  // redefine the default value for property INSTRUCTION_OPCODE
  // PropertyCollection represents predefined collection of PropertyID objects
  event.addProperty(PropertyCollection.INSTRUCTION_OPCODE, insn.toString());
  traceFilter.processEvent(event, eventType.instructionExecuted);
 }

 public void objectLocked(JVM vm) {
  Event event = this.createObjectEvent(vm, eventType.objectLocked);
  event.addProperty(PropertyCollection.THREAD_ID, vm.getLastThreadInfo().getIndex());
  traceFilter.processEvent(event, type);
 }
```

Base trace emitter provides `createEvent` methods for creating events that have to be initialized with default set of properties. A property is added to event by calling `event.addProperty()` and providing property id (represented with the `PropertyID` class) and property value (Java primitive or String types are allowed). To use `ObjInsnEmitter`, replace `listener=gov.nasa.jpf.traceEmitter.DefaultTraceEmitter` with `listener=gov.nasa.jpf.traceEmitter.DefaultTraceEmitter` in the above defined .jpf file, since only one trace emitter is allowed. (You can have as many other listeners as you want.)

Trace Filter

As seen above, trace emitter uses `trace filter` to `process` each event. A filter is capable of forwarding events in the same way they are received. It can then act as an emitter to another trace filter, or a trace storer, allowing pipelining multiple trace storers. `TraceFilter` is implemented as a Chain of Responsibility design pattern. If the filter wants to filter out events, it does so by not forwarding data down the chain. Thus, the chain will break, and event will be discarded. For example, to write `InstructionFilter` that filters all `instructions` except for `NEW` ones, one needs to extends `TraceFilter` and override `processInstructionExecuted` (to filter any other event type, just override corresponding `process` method):

```
public class ConsolePrintFilter extends TraceFilter {
 public void processInstructionExecuted(Event event) {
  String insnOpcode = (String) event.getProperty(PropertyCollection.INSTRUCTION_OPCODE);
  if (insnOpcode.equals("new") {
    forward(event, eventType.instructionExecuted);
  }
 }
}
```

To register `InstructionFilter` add this snippet to the `.jpf` defined above:

```
# add trace filter classes (full name required)
traceServer.trace_filter=gov.nasa.jpf.example.InstructionFilter
```

Since trace emitter communicates only with the trace filter, the trace filter is responsible for forwarding events to the trace storer. This is done automatically by the system. In order to use more than one trace storer, one needs to add `traceServer.additional_trace_storers` property, and the system will handle that automatically (by using `GeneralTailTraceFilter`):

```
# add more trace storers (different than the traceServer.trace_storer value)
traceServer.additional_trace_storers=neo4j
```

Trace Query

We have the database, so we need the query mechanism to pull the interesting data from it. `TraceQuery` provides database independent query API, for querying, among others, the `last path`, all the paths that `ended` and for querying the entire event space. To do that, one would have to provide a `TracePredicate`, which will be called upon to decide whether currently processed event from the trace should be included in the result. For example, to query for `objectLocked` and `objectUnlocked` events only, one should write:

```
TracePredicate predicate = new TracePredicate() {
 public boolean filter(Event currentEvent) {
  EventTypes.eventType eType = currentEvent.getEventType();
  switch (eType) {
    case objectLocked:
```

```
      case objectUnlocked: {
        return true;
      }
    }
    return false;
  }
};

// don't invert the events in the result
boolean reversePath = false;
EventIterator path = query.getLastPath(predicate, reversePath);

for (Event event : path) {
  ...
}
```

Trace Analyzer

The whole system is designed for trace data analysis. The trace analyzer should query the database, do some *analysis* and report the results. Currently, each analyzer can define its own reporting mechanism, or can use `trace printer`.

```
public class ExampleAnalyzer extends TraceAnalyzer {
  private void analyzeAll() { ... }
  private void analyzeThreads() { ... }
  private void analyzeMethods() { ... }

  // analyze only the "NEW" instructions,
  // and print the result if some condition is satisfied
  private void analyzeInstructions() {
    TracePredicate predicate = new TracePredicate() {
      public boolean filter(Event currentEvent) {
        return currentEvent.getEventType() == EventTypes.eventType.instructionExecuted;
      }
    };

    // don't invert the events in the result
    boolean reversePath = false;
    EventIterator path = query.getLastPath(predicate, reversePath);

    for (Event event : path) {
      String opcode = (String) event.getPropety(PropertyCollection.INSTRUCTION_OPCODE);
      if (opcode.equals("new") && ...) {
        System.out.println(event.getPropety(...));
        ...
      }
    }
  }

  // start the analysis
  // what would be analyzed depends on how the analyzer was configured
  public void analyze() {
    query.startTraceQuery();
    if (format.equals(INSTRUCTIONS)) {
      analyzeInstructions();
    } else if (format.equals(METHODS)) {
      analyzeMethods();
    } else if (format.equals(THREADS)) {
      analyzeThreads();
    } else {
      analyzeAll();
    }
```

```
    query.stopTraceQuery();
  }

  // configure the analyzer with some parameters
  // for this analyzer, we have the type of analysis that is going to be performed
  public void configureAnalyzer(Object... args) {
    if (args.length > 0) {
      format = (String) args[0];
    } else {
      format = ALL;
    }
  }
}
```

Trace analysis may be performed during the execution (on-the-fly) or post-mortem, after the program has terminated. To use it on-the-fly, one needs to add:

```
# trace query that analyzers will be use to query the database
# "inMemory" are "neo4j" are the possible choices, default is "inMemory"
traceServer.trace_query=inMemory

# add trace analyzers (full name required)
traceServer.trace_analyzer=gov.nasa.jpf.example.ExampleAnalyzer

# provides parameters for the analyzers
traceServer.trace_analyzer.params=instructions
```

Trace Printer

We have the execution trace, we know how to query it for various information, we can analyze the data, the only thing we left is the printing.

Console Trace Printer

`ConsoleTracePrinter` prints the new trace in old JPF's printer fashion. The main difference is that the new printer can print custom data added to the trace, by defining `PropertyCollection.TRACE_EXTRA_DATA` property for each event that one would like to augment with extra data. The extra data will be printed right after the default event's data is printed. Console printer configuration parameters are:

```
# register console trace printer as a publisher
report.publisher=consoleTracePrinter
report.consoleTracePrinter.class=gov.nasa.jpf.traceServer.printer.ConsoleTracePrinter
# print trace when property is violated
report.consoleTracePrinter.property_violation=output,trace

# Show the steps from inside the transition. Default is true.
report.consoleTracePrinter.show_steps=true

# Show the source code for executed instruction (line + location).
# Default is true.
report.consoleTracePrinter.showSource=true

# Show the instruction location from the source file. Used only if
# showLocation is set to true. Default is true.
report.consoleTracePrinter.showLocation=true

# Show the method name and the instruction opcode.
# Default is false.
report.consoleTracePrinter.showCode=true

# Show the method name. Used only if {@link #showLocation} is set to true.
# Default is false.
report.consoleTracePrinter.showMethod=true
```

```
# Show the choice generator information for transitions.
# Default is true.
report.consoleTracePrinter.showCG=true

# Show the extra data added to trace by using
# PropertyCollection.TRACE_EXTRA_DATA property. Default is true.
report.consoleTracePrinter.showExtraData=true
```

If you run the trace server with the `new console printer`, and above parameters, output snippet might look like this:

```
...

------------------------------------------------------ transition #5 thread: 2
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {Thread-0,>Thread-1}
 oldclassic.java:123            : count = event2.count;          // <race> violates event2 monitor encapsulation
   SecondTask.run()V
     getfield
     putfield
 oldclassic.java:126            : System.out.println("  2");
     getstatic
     ldc
     invokevirtual java.io.PrintStream.println(Ljava/lang/String;)V
 oldclassic.java:127            : event1.signal_event();       // updates event1.count
     aload_0
     getfield
     invokevirtual Event.signal_event()V
 ...
```

If trace emitter is used that will add `System.out.println()` arguments to the trace by using the `PropertyCollection.TRACE_EXTRA_DATA` property, the same path that shown above would look like this:

```
...

------------------------------------------------------ transition #5 thread: 2
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {Thread-0,>Thread-1}
 oldclassic.java:123            : count = event2.count;          // <race> violates event2 monitor encapsulation
   SecondTask.run()V
     getfield
     putfield
 oldclassic.java:126            : System.out.println("  2");
     getstatic
     ldc
     invokevirtual java.io.PrintStream.println(Ljava/lang/String;)V
       "  2"
 oldclassic.java:127            : event1.signal_event();       // updates event1.count
     aload_0
     getfield
     invokevirtual Event.signal_event()V
 ...
```

The `System.out.println()` argument (in this case " 2") is printed right after the `invokevirtual` instruction.

Generic Console Trace Printer

Provides a more powerful printing mechanism. It extends `ConsoleTracePrinter`, but it uses `TracePrinter` to print the trace.

`TracePrinter` provides different printer for every group of events (see `gov.nasa.jpf.traceServer.traceStorer.EventTypes.eventGroupType` for more details). To configure trace printer, add following parameters:

```
# register genericConsoleTracePrinter as a publisher
report.publisher=genericConsoleTracePrinter
report.genericConsoleTracePrinter.class=gov.nasa.jpf.traceServer.printer.GenericConsoleTracePrinter
report.genericConsoleTracePrinter.property_violation=output,trace

# print instruction events
traceServer.tracePrinter.instruction.show=true

# don't add extra data to the output
# default value is true for all event types
traceServer.tracePrinter.instruction.printExtraData=false

# print object events
traceServer.tracePrinter.object.show=true
# print thread events
traceServer.tracePrinter.thread.show=true
```

To print event group, one must set `traceServer.tracePrinter.eventType.show` to `true`, where `eventType` is type from `eventGroupType`. Each event printer can be replaced by extending the `EventPrinter`, base printer class. The new printer have to be registered in order to be used within trace printer,by setting the `traceServer.tracePrinter.eventType.class` parameter, where `eventType` is type from `eventGroupType`:

```
traceServer.tracePrinter.object.class=gov.nasa.jpf.traceServer.extensions.NewObjectPrinter
```

Output example from the `NewObjectPrinter` that prints location of object's initialization (trace printer is configured to print instruction events as well):

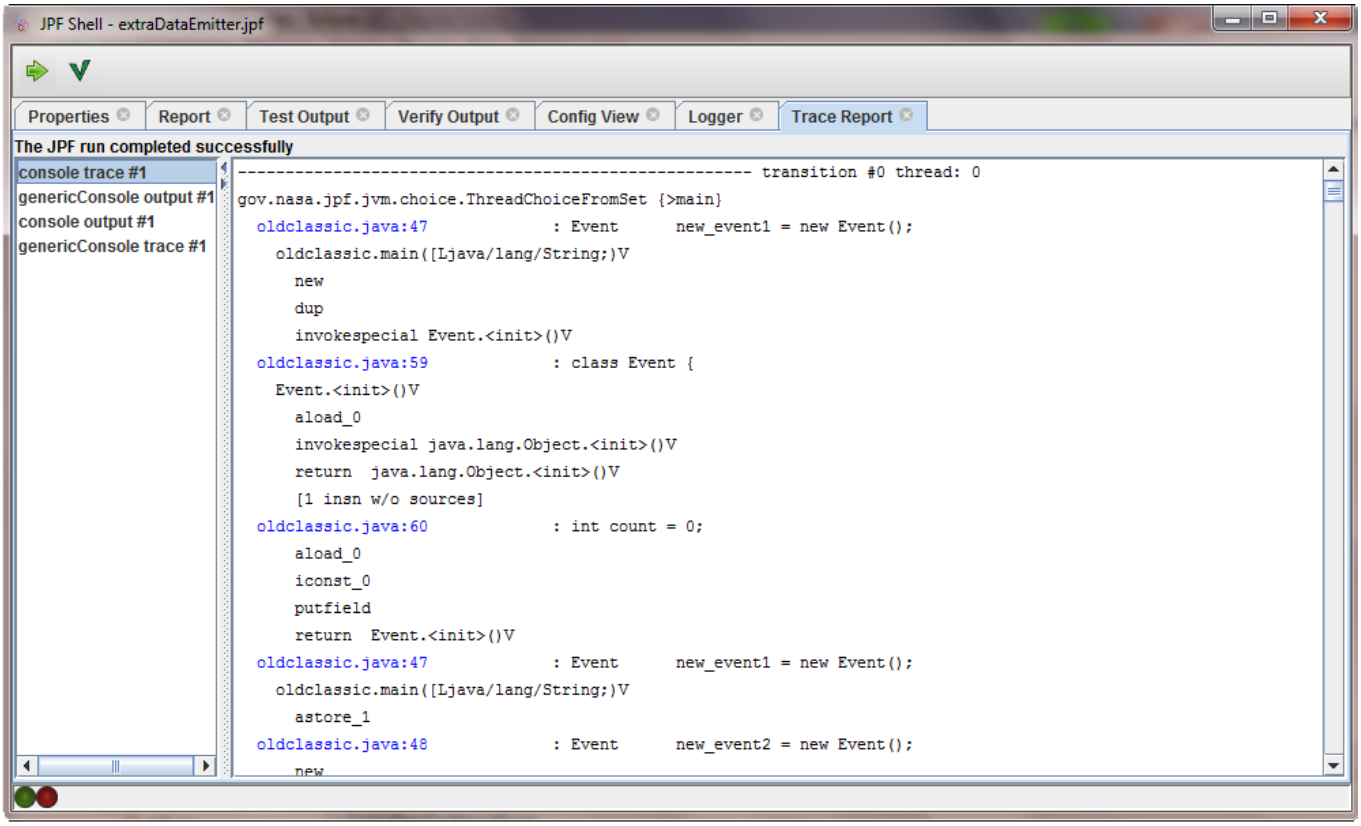
```
...
instructionExecuted
 oldclassic.java:130
   SecondTask.run()V
     invokevirtual Event.wait_for_event()V
objectLocked
 291 # LEvent; # init at: oldclassic.java:48
instructionExecuted
 oldclassic.java:79
   Event.wait_for_event()V
     aload_0
objectUnlocked
 291 # LEvent; # init at: oldclassic.java:48
objectWait
 291 # LEvent; # init at: oldclassic.java:48
...
```

If events from the same group still have to be printed in a different way, this can achieved by writing something like this:

```
public class StatePrinter extends EventPrinter {
...
 public void print(PrintWriter out, Event event) {
   if (event.getEventType() == eventType.stateAdvanced) {
     out.println("===================== stateId: "
         + event.getProperty(PropertyCollection.STATE_ID));
     super.print(out, event);
   } else if (event.getEventType() == eventType.stateRestored) {
     out.println(...);
   }
 }
}
```

Trace Report Shell Panel

Reports generated with `console trace printer` and `generic console trace printer` can be viewed in the new `Shell` panel (from the `jpf-shell`), called `Trace Report`.



Trace printer's results are divided into browseable topics. If no trace printer is defined, `console trace printer` is automatically added, and corresponding report panel is launched. To use `trace report panel`, add following to the `.jpf` file:

```
# turn on the Shell
shell=.shell.basicshell.BasicShell
```

This will turn on the default `Shell`, and the Trace report panel will be added to the Shell.

How to download.

You can obtain the source code of the jpf-trace-server using Mecurial (hg):

```
hg clone http://bitbucket.org/igor.andjelkovic/jpf-trace-server
```

The jpf-trace-server comes with a `jpf.properties` file for configuration with JPF. The file should work if you have checked out jpf-trace-server as a subdirectory of the overall jpf repository, with jpf-core being another subdirectory. For example, JPF resides in project/jpf-core, the extension in project/jpf-trace-server. Note: you will have to have site.properties.

How to build.

You can either build Java PathFinder from the command line with Ant, or from within Eclipse. To compile the jpf-trace-server project, we recommend building sources with Apache Ant. As said above, jpf-core needs to reside in project/jpf-core if the extestions is in project/jpf-trace-server. To build with Ant, switch to the directory where the jpf-trace-server extension is located (where build.xml file is located), and run

```
ant
```

which should compile all jpf-trace-server sources.

Installation

When you have built the jpf-trace-server project, one jar file is created under directory `build`:

- `jpf-trace-server.jar`: The implementation of the trace server.

The jar file should be under directory `build` if you build the project with Apache Ant.

Running Trace Server.

If you want to run your own program, the easiest way to execute JPF and use trace-server with several options is to create an application property file. For example, you can create property file like this:

```
target = [Application]
target_args = [application_args]
classpath = [classpath to your application]
sourcepath = [source path to your application]

listener=gov.nasa.jpf.traceEmitter.DefaultTraceEmitter
traceServer.db_location = dbTrace
traceServer.local_storer = true
traceServer.trace_storer = inMemory
traceServer.skip_init = true

report.publisher=genericConsoleTracePrinter
report.genericConsoleTracePrinter.class=gov.nasa.jpf.traceServer.printer.GenericConsoleTracePrinter
report.genericConsoleTracePrinter.property_violation=output,trace


traceServer.tracePrinter.instruction.show=true
traceServer.tracePrinter.instruction.printExtraData=false

traceServer.tracePrinter.object.show=true
traceServer.tracePrinter.thread.show=true

report.console.property_violation = error,trace

...
```

You should set `JPF_CORE` to the directory you installed the jpf-core project, as described above. Suppose that you save the above property file as `myapplication.jpf`, the command starting JPF becomes like this:

```
java -jar ${JPF_CORE}/build/RunJPF.jar myapplication.jpf
```

For other arguments to trace-server, see all comments mentioned above.

In order to run a analyzer post mortem (works only if neo4j trace storer was used to save the trace), you should run something like tihs:

```
JPF_TRACE_SERVER=".."
JPF_CORE="../../jpf-core/build/jpf.jar"
NEO_KERNEL="${JPF_TRACE_SERVER}/lib/neo4j-kernel-1.0.jar"
NEO_GERONIMO="${JPF_TRACE_SERVER}/lib/geronimo-jta_1.1_spec-1.1.1.jar"

CLASSPATH="${JPF_TRACE_SERVER}/build/main:${NEO_KERNEL}:${NEO_GERONIMO}:${JPF_CORE}"

java -classpath "${CLASSPATH}" gov.nasa.jpf.traceAnalyzer.DeadlockAnalyzer dbTrace neo4j
```

This will run the `DeadlockAnalyzer` with the database location and trace query name as parameters. The meaning of each variable is described below:

- `JPF_TRACE_SERVER_HOME` : jpf-trace-server base directory
- `JPF_CORE` : jpf-core binaries
- `NEO_KERNEL and NEO_GERONIMO` : neo4j libraries

Running tests.

Test for the jps-trace-server are located in the `bin/folder`. In order to run them, you need to run the shell script `test.sh`. This will run all the tests. To run some of them separately, you will have to copy/paste the output of the script. If you run the `test.sh` you will get the output as following:

```
$ ./test.sh

./traceServer.sh \
../src/examples/deadlockAnalyzerEmitter.jpf \
+traceServer.trace_storer=inMemory \
+traceServer.trace_analyzer=gov.nasa.jpf.traceAnalyzer.DeadlockAnalyzer \
+traceServer.trace_analyzer.params=essential \
> ../log/deadlAE-inMem-DeadlA-essen.log
# ok

./traceServer.sh \
../src/examples/deadlockAnalyzerEmitter.jpf \
+traceServer.trace_storer=inMemory \
+traceServer.trace_analyzer=gov.nasa.jpf.traceAnalyzer.DeadlockAnalyzer \
+traceServer.trace_analyzer.params=column \
> ../log/deadlAE-inMem-DeadlA-colum.log
# ok

...
```

The single test is the:

```
./traceServer.sh \
../src/examples/deadlockAnalyzerEmitter.jpf \
+traceServer.trace_storer=inMemory \
+traceServer.trace_analyzer=gov.nasa.jpf.traceAnalyzer.DeadlockAnalyzer \
+traceServer.trace_analyzer.params=essential \
> ../log/deadlAE-inMem-DeadlA-essen.log
# ok
```

so you can copy that and run the test alone.

> `#ok` means that the test has succeded. Output of the test was written to `../log/deadlAE-inMem-DeadlA-essen.log`, so you can look at the .log file to see what the test was doing. The test passes if the output (.log file) is the same as the corresponfing `.out` file, in this case `../log/deadlAE-inMem-DeadlA-essen.out`. `.out` files are located in the `log` folder, subdirectory of the jpf-trace-server home.

Test.sh has a mode `test.sh -n`, which only lists what it would run, without executing it. That way you would get the list of tests with all the parameters needed for test to run.

Repository

The sources for this project are available from a Mercurial repository on http://babelfish.arc.nasa.gov/hg/jpf/jpf-trace-server

Trace Server as part of the JPF Summer Projects

The official JPF Summer Project page: http://babelfish.arc.nasa.gov/trac/jpf/wiki/summer-projects/2010-trace-server