Revamping the `CallMonitor` Listener
Final Report

Franck van Breugel

April 1, 2020

The listener `CallMonitor` of JPF prints for each method that is called

- the ID of the thread that executed the call,
- the depth of the stack,
- the name of the class,
- the name of the method, and
- its arguments.

Consider, for example, the following app.

```
public class Example {
  public static void main(String [] args) {
    first(1, true);
  }

  private static void first(int i, boolean b) {
    second(i + 1);
  }

  private static void second(int i) {
    // do nothing
  }
}
```

When JPF is run on the above app with the `CallMonitor` listener, it produces a lot of output including

```
0:    Example.first(1,true)
0:     Example.second(2)
```

Both method invocations are executed by thread 0, which is the main thread. The number of spaces following `0:` indicates the depth of the stack. Hence, the method `second` is invoked within the method `first`.

Milestones

The class lacks documentation and some of the variable names are cryptic. The class also does not use JPF's reporting system. In this project, I have

1. documented and cleaned up the code,

2. modified the class so that it makes use of JPF's reporting system,
3. developed JPF tests, and
4. described how to use the listener.

Below, I will discuss each milestone in some detail.

Code Documentation and Clean Up and JPF's Reporting System

Below, you find the code of the revamped `CallMonitor` listener. The changes are colour coded: addition of documentation, non-cryptic variable names, code related to JPF's reporting system, and other changes.

```
1  /*
2  * Copyright (C) 2014, United States Government, as represented by the
3  * Administrator of the National Aeronautics and Space Administration.
4  * All rights reserved.
5  *
6  * The Java Pathfinder core (jpf-core) platform is licensed under the
7  * Apache License, Version 2.0 (the "License"); you may not use this file except
8  * in compliance with the License. You may obtain a copy of the License at
9  *
10 *      http://www.apache.org/licenses/LICENSE-2.0.
11 *
12 * Unless required by applicable law or agreed to in writing, software
13 * distributed under the License is distributed on an "AS IS" BASIS,
14 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
15 * See the License for the specific language governing permissions and
16 * limitations under the License.
17 */
18 package gov.nasa.jpf.listener;
19
20 import gov.nasa.jpf.Config;
21 import gov.nasa.jpf.JPF;
22 import gov.nasa.jpf.ListenerAdapter;
23 import gov.nasa.jpf.jvm.bytecode.JVMInvokeInstruction;
24 import gov.nasa.jpf.report.Publisher;
25 import gov.nasa.jpf.report.PublisherExtension;
26 import gov.nasa.jpf.vm.ClassInfo;
27 import gov.nasa.jpf.vm.Instruction;
28 import gov.nasa.jpf.vm.MethodInfo;
29 import gov.nasa.jpf.vm.ThreadInfo;
30 import gov.nasa.jpf.vm.VM;
31 import gov.nasa.jpf.vm.VMListener;
32
33 import java.io.PrintWriter;
34
35 /**
36  * This listener monitors method invocations.  When JPF finishes, it publishes
37  * for each method invocation,
38  * <ul>
39  * <li>the ID of the thread that executed the method invocation,</li>
```

```
40  * <li>the depth of the stack of that thread,</li>
41  * <li>the name of the class to which the method belongs,</li>
42  * <li>the name of the method, and</li>
43  * <li>the arguments of the method.</li>
44  * </ul>
45  * For example, consider the following.
46  * <pre>
47  * 0:  Example.first(1,true)
48  * 0:  Example.second(2)
49  * </pre>
50  * Both method invocations are executed by thread 0, which is the main thread.
51  * The number of spaces following 0:  indicates the depth of the stack.
52  * Hence, the method second is invoked within the method first.
53  *
54  * @author Unknown
55  * @author Franck van Breugel
56  */
57 public class CallMonitor extends ListenerAdapter
58     implements VMListener,  PublisherExtension  {
59   private StringBuilder result;
60
61   /**
62    * Initializes this listener.
63    *
64    * @param configuration JPF's configuration.
65    * @param jpf JPF.
66    */
67   public CallMonitor(Config configuration, JPF jpf) {
68     this.result = new StringBuilder();
69     jpf.addPublisherExtension(Publisher.class, this);
70   }
71
72   /**
73    * Whenever a method is invoked, information about the call is recorded.
74    *
75    * @param vm JPF's virtual machine.
76    * @param thread the thread that executed the instruction.
77    * @param next the next instruction to be executed.
78    * @param executed the executed instruction.
79    */
80   @Override
81   public void instructionExecuted(VM vm, ThreadInfo thread , Instruction next ,
82       Instruction executed ) {
83     if (executed  instanceof JVMInvokeInstruction) {
84       if (executed.isCompleted(thread ) && !thread .isInstructionSkipped()) {
85         JVMInvokeInstruction call = (JVMInvokeInstruction) executed ;
86         this.publishCall(call, thread);
87       }
88     }
89   }
90
```

```
91   /**
92    * Records information of the given call, executed by the given thread.  In
93    * particular, it records the thread ID, the depth of the stack, the name of
94    * the class, the name of the method, and its arguments.
95    *
96    * @param call the method invocation.
97    * @param thread the thread that executed the method invocation.
98    */
99   private void publishCall(JVMInvokeInstruction call, ThreadInfo thread) {
100    MethodInfo method  = call.getInvokedMethod();
101    Object[] argument  = call.getArgumentValues(thread);
102    ClassInfo clazz  = method .getClassInfo();
103
104    // thread ID
105    this.result .append(thread .getId());
106    this.result .append(": ");
107
108    // stack depth
109    int depth  = thread .getStackDepth();
110    for (int d = 0; d < depth ; d++) {
111      this.result .append(" ");
112    }
113
114    // class name
115    if (clazz  != null){
116      this.result .append(clazz .getName());
117      this.result .append('.');
118    }
119
120    // method name
121    this.result .append(method .getName());
122    this.result .append('(');
123
124    // arguments
125    int length  = argument .length;
126    for (int i = 0; i < length ; i++) {
127      if (argument [i] != null) {
128        this.result .append(argument [i].toString());
129      } else {
130        this.result .append("null");
131      }
132      if (i < length  - 1) { // no comma after the last argument
133        this.result .append(',');
134      }
135    }
136    this.result .append(')');
137
138    this.result .append('\n');
139  }
140
141  /**
```

```
142     * When JPF has finished, the information of all method calls is published.
143     *
144     * @param publisher JPF's publisher.
145     */
146    @Override
147    public void publishFinished(Publisher publisher) {
148      PrintWriter output = publisher.getOut();
149      publisher.publishTopicStart("method invocations");
150      output.print(this.result);
151      publisher.publishTopicEnd("method invocations");
152    }
153  }
```

The code related to JPF's reporting system is based on Chapter 8 of the notes. The other changes are the following.

- It has been made explicit that the class `CallMonitor` implements the interface `VMListener`.
- Part of the `instructionExecuted` has been moved into the `publishCall` method.

JPF Tests

To implement the tests, we use the following skeleton presented during one of the lectures.

```
PrintStream out = null;
ByteArrayOutputStream stream = null;
if (!isJPFRun()) {
  out = System.out;
  stream = new ByteArrayOutputStream();
  System.setOut(new PrintStream(stream));
}
if (verifyNoPropertyViolation(CONFIGURATION)) {
  ...
} else {
  System.setOut(out);
  String output = stream.toString();
  ...
}
```

where the constant is defined by

```
private static final String[] CONFIGURATION = {
  "+listener=gov.nasa.jpf.listener.CallMonitor",
  "+classpath=...",
  "+native_classpath=..."
};
```

In total, I developed 14 tests. The tests can be divided into the following groups:

- an "empty test" that checks that the main thread (numbered 0) invokes a method,
- tests that check the invocation of static and non-static methods with zero or one arguments,
- a test that checks that two threads invoke methods,
- a test that checks for two method invocations at the same depth, and
- a test that checks for two method invocations at different depths.

Potential Future Improvements

The `CallMonitor` listener provides a lot of output. It might be helpful to provide the listener with parameters so that that the user can customize the listener to filter on particular method invocations. For example, one could filter on method invocations

- by a particular thread,
- on a particular object,
- on instances of a particular class, or
- of a particular method.