Revamping the CallMonitor Listener

EECS 4315

Franck van Breugel

March 29, 2020

CallMonitor listener

The listener `CallMonitor` of Java PathFinder (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.

CallMonitor listener

Consider 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
  }
}
```

CallMonitor listener

Run JPF on the following application properties file.

```
target = Example
classpath = <path to Example.class>
listener = gov.nasa.jpf.listener.CallMonitor

@using jpf-shell
shell = gov.nasa.jpf.shell.basicshell.BasicShell
```

CallMonitor listener

JPF produces the following output

```
...
0:   Example.main([Ljava.lang.String;@bb)
0:    Example.first(1,true)
0:     Example.second(2)
...
```

CallMonitor listener

JPF produces the following output

```
...
0:   Example.main([Ljava.lang.String;@bb)
0:    Example.first(1,true)
0:     Example.second(2)
...
```

- All methods are called by thread 0, the main thread.
- The number of spaces following `0:` indicates the depth of the stack.

CallMonitor listener

JPF's `CallMonitor` listener

- lacks documentation,
- contains variable names that are cryptic,
- does not use JPF's reporting system, and
- lacks tests.

Documentation

Old:

```
/**
 * this isn't yet a useful tool, but it shows how to track
 * method calls with their corresponding argument values
 */
```

Documentation

Old:

```
/**
 * this isn't yet a useful tool, but it shows how to track
 * method calls with their corresponding argument values
 */
```

New:

```
/**
 * This listener monitors method invocations. When JPF
 * finishes, it publishes for each method invocation,
 * the ID of the thread that executed the method
 * invocation, the depth of the stack, the name of the
 * class, the name of the method, and its arguments.
 *
 * @author Unknown
 * @author Franck van Breugel
 */
```

Documentation

Old:

Documentation

Old:

New:

```
/**
 * Whenever a method is invoked, information about the
 * call is recorded.
 *
 * @param vm JPF's virtual machine
 * @param thread the thread that executed the instruction
 * @param next the next instruction to be executed
 * @param executed the executed instruction
 */
```

Cryptic variable names

Old:

```
... ti ...
...
... mi ...
...
... ci ...
...
... sb ...
```

Cryptic variable names

Old:

```
... ti ...
...
... mi ...
...
... ci ...
...
... sb ...
```

New:

```
... thread ...
...
... method ...
...
... clazz ...
...
... result ...
```

JPF's reporting system

```
private StringBuffer result;

public CallMonitor(Config configuration, JPF jpf) {
  ...
  jpf.addPublisherExtension(Publisher.class, this);
}

public void publishFinished(Publisher publisher) {
  PrintWriter output = publisher.getOut();
  publisher.publishTopicStart("method invocations");
  output.print(this.result);
  publisher.publishTopicEnd("method invocations");
}
```

CallMonitor listener

With the revamped `CallMonitor` listener, JPF produces the following output

```
...
================================ method invocations
...
0:   Example.main([Ljava.lang.String;@bb)
0:    Example.first(1,true)
0:     Example.second(2)
...
```

Testing

Developed ten tests.

```
private static void staticMethod() {}

@Test
public void staticMethodTest() {
  ...
  if (verifyNoPropertyViolation(CONFIGURATION)) {
    staticMethod();
  } else {
    // check if output contains the String
    // "0:.*CallMonitorTest.staticMethod()"
  }
}
```

To do

- Develop further tests. In particular,
  - tests with nested method calls, and
  - tests with multiple threads.
- Write a report.