Testing Exceptions with Enforcer

Cyrille Artho

February 23, 2010

National Institute of Advanced Industrial Science and Technology (AIST),
Research Center for Information Security (RCIS)

Abstract

Java library calls, for instance for network I/O, may result in exceptions. Exhaustive testing of exceptions is very difficult. Enforcer automatically tests all relevant outcomes of actions that may cause an exception, and combines the structure of unit tests with coverage information and fault injection. Enforcer supports Java 1.5–1.6 and JUnit 3.8.1.

1 Introduction

Testing is very flexible and by far the most widespread quality assurance method today. However, a severe limitation of testing is non-determinism, such as possible exceptions occurring for input/output (I/O) operations. As a result of this, the outcome of exceptions is often poorly tested. In a typical test suite, many possible exceptions (such as for input/output) are not covered. Each method call that may throw exceptions may have an effect on the program, but systematic testing of this is hard.

The Enforcer tool targets non-determinism given by potential I/O failures of the underlying system [1]. Typically the successful case is easy to test, while the failure case can be nearly impossible to trigger. For instance, simulating network outage is non-trivial. Enforcer does not try to cause underlying system calls to fail, but instead it *injects* a failure into the program at run-time to create the same behavior that would have resulted from a system failure.

For system calls, there are two basic outcomes: success or failure. Typically the successful case is easy to test, while the failure case can be nearly impossible to trigger. For instance, simulating network outage is non-trivial. Existing ad-hoc approaches to test exceptions in such cases include factoring out small blocks of code, to test exception handlers manually, or adding extra flags to conditionals that trigger outcomes that are normally not tested. Figure 1 illustrates the latter. The unit test has to set a special flag which causes the exception handling code to run artificially. This works well for a few cases but can be automated.

Enforcer automates the testing of exceptions using fault injection. It is very efficient and repeats only one unit test per uncovered exception. The tool operates in three stages [1, 2]:

```
try {
    if (testShouldFail) {
        throw new IOException();
    }
    socket = new ServerSocket();
} catch (IOException e) {
    // error handling code
}
```

Figure 1: Manual approach for exception handler coverage.

1. Code instrumentation, at compile time or at class load time. This includes injecting code for coverage measurement and for execution of the repeated test suite.
2. Execution of unit tests. Coverage information is now gathered.
3. Re-execution of certain tests, using selective fault injection. This has to be taken into account by coverage measurement code, in order to require only a single instrumentation step.

2 Getting Started

2.1 Compatibility

Enforcer has been tested under Java 1.5 and 1.6, Linux and Mac OS X. It should work on any similar configuration. However, Enforcer was designed and implemented against JUnit 3.8.1. Newer versions of JUnit (4.X) unfortunately use a different way to declare unit tests, and are *not* supported in this release of Enforcer.

Furthermore, Enforcer uses the "serp" bytecode instrumentation library. This library is hosted on `http://serp.sourceforge.net/` but currently unavailable at that site. A working binary version of `serp.jar` is therefore provided as part of the distribution. Version 3.8.1 of JUnit is also included; JUnit can be found at `http://junit.org/`.

2.2 Installation

2.2.1 From source code

The build process is `make`-based; the `Makefile` is generated from `Makefile.in` by a simple `./configure` shell script that analyzes dependencies. Usage:

```
./configure
make
```

This produces a file called `enforcer.jar`, which is also provided as a binary release.

2.2.2 Binary (jar file) release

To use Enforcer, place the following three .jar files in the `CLASSPATH`:

- `enforcer.jar`
- `serp.jar`
- `junit3.8.1.jar`

Note: `serp.jar` is not the latest version of serp. The latest version contains a bug that prevents Enforcer from working.

2.3 Running the tool

Fundamentally, the three steps required to run the tool (instrumentation, coverage measurement, repeated test execution) can be broken down into two categories: Static code analysis, which instruments method calls that may throw exceptions, and run-time analysis. Run-time analysis includes coverage measurement and re-execution of certain unit tests.

Code instrumentation is entirely static, and can be performed after compilation of the application source code, or at class load time. Enforcer supports both modes of operation.

2.3.1 Static instrumentation

Static instrumentation takes a set of class files as input and produces a set of instrumented files as output. Class files requiring no changes are not copied to the target directory, because the Java classpath mechanism can be used to load the original version if no new version is present. Static instrumentation is invoked as follows:

```
java -jar enforcer.jar [-d <outdir>] [--options...]  class
[class...]
```

The output directory is where the instrumented class files are written. These modified files have to override the default class files, so the output directory containing instrumented class files has to be specified before the directory containing the compiled files in the classpath. If the JUnit test suite is normally executed by running

```
   java -classpath ${CLASSPATH} [-options...]  <target>
```

then the output directory (`<outdir>` in the instrumentation command above) has to be prepended to the classpath:

```
   java -classpath <outdir>:${CLASSPATH} [-options...]  <target>
```

2.3.2 Dynamic (load-time) instrumentation

Alternatively, Enforcer can be invoked at load-time. In this mode, the new Java instrumentation agent mechanism is used [4]. Instrumentation is invoked by appending the argument "`-javaagent:enforcer.jar`" to the `java` command. Load-time instrumentation is very elegant in the sense that it does not entail the creation of temporary files and reduces the entire usage of the enforcer tool to just adding one extra command line option. There is no need to specify a set of input class files because each class file is instrumented automatically when needed, i. e., when it is loaded at run-time. Unfortunately, the new agent instrumentation interface of Java 1.5 is may produce incorrect results when dynamic class loading, or a custom class loader, is used.

Execution of the original and repeated tests is performed automatically. If the instrumentation agent mechanism cannot be used, the command to run the JUnit test suite has to be executed as a second step after instrumentation. This does not require any additional arguments, since the extra behavior is inserted as program code. However, it requires a correct `CLASSPATH` setting, which contains the output directory of the prior instrumentation step. It is assumed that the test suite is started via `main` through a call to `junit.textui.TestRunner.run(TestSuite suite)` or a similar call, e. g. using the GUI test runner. Tests are automatically wrapped such that coverage information can be gathered.

For execution of repeated tests, no special reset mechanism is necessary. In JUnit, each test is self-contained; test data is initialized from scratch each time prior to execution of a test. Therefore re-execution of a test just recreates the original data set. After execution of the original and repeated tests, a report is printed which shows the number of executed methods calls that can throw an exception, and the number of executed catch clauses which were triggered by said method calls. If instrumentation occurs at load time, then the number of instrumented method calls is also shown. The Enforcer output is shown once the JUnit test runner has finished (see Figure 3).

2.4 Example

Figure 2 shows an example Java method with four operations that may throw checked exceptions. This code is contained in the source code distribution in `Example.java`. Most of the method calls in this example additionally may throw a `SecurityException`, which is an *unchecked* exception; therefore, exception handlers for that exception are not mandatory. As can be expected, when testing this method, using a unit test from `TestExample.java`, no exceptions occur. Exceptions only occur if a temporary file cannot be created because the user has no access permission, or the disk is full (or faulty). During development, this is highly unlikely to happen, so these exception handlers typically go untested.

Enforcer allows the developer to test the presence or absence of all four exceptions. It works as follows:

1. The initial test run works without fault injection and successfully completes the `try` block.
2. A unit test executing the example method is chosen, and run with fault injection:
   (a) The initial method call, `createTempFile`, is forced to fail with an `IOException`.
   (b) In a new test run, the constructor of `FileOutputStream` fails with a `FileNotFoundException`.
   (c) The third test runs successfully up to `write`, where an `IOException` occurs.

```
public static void twiddleTempFile() {
  try {
    File tempFile = File.createTempFile("enforcer", "");
    // (1) IOException may occur
    assert (tempFile.canRead());
    assert (tempFile.canWrite());
    FileOutputStream out = new FileOutputStream(tempFile);
    // (2) FileNotFoundException may occur
    out.write(new String("Test").getBytes());
    // (3) IOException may occur
    out.close();
    // (4) IOException may occur
    tempFile.delete();
  } catch (FileNotFoundException e) {
    System.err.println("Cannot access temp. file.");
    e.printStackTrace();
  } catch (IOException e) {
    System.err.println("Cannot create, write, or close temp. file.");
    e.printStackTrace();
  }
}
```

Figure 2: A simple example with four possible exceptions.

(d) The fourth test run fails at `close`. Note that `delete` declares no checked exceptions, so no extra test is needed there.

2.5 Evaluation of results

Figure 3 shows the output of the Enforcer tool when using
`java -javaagent:enforcer.jar TestExample` or `make demo`.
First, the initial test suite is run. If it is deterministic, it produces the exact same output as when run by the normal JUnit test runner. After the JUnit test suite has concluded, JUnit reports the total run time and the test result (the number of successful and failed tests). After the completed JUnit test run, Enforcer reports exception coverage.

If coverage reported is less than 100 %, a new test suite is created, which improves coverage of exceptions. This repeated test suite is then executed in the same way the original test suite was run, with the same type of coverage measurements reported thereafter (see Figure 3).

The output can be interpreted as follows: 4 method calls that declare exceptions were present in the code executed. Out of these, all four were executed, but no exceptions occurred. Therefore, four tests are run again; the second run covers the remaining paths. Note that the second run may have covered additional method calls in nested `try`/`catch` blocks. This would have allowed increased coverage by launching another test run to cover nested exceptions [1].

```
java -javaagent:enforcer.jar TestExample
.
Time:  0.223
OK (1 test)
*** Total number of instrumented method calls:  4
*** Total number of executed method calls:  4
*** Total number of executed catch blocks:  0
*** Tests with uncovered catch blocks to execute:  4
....
Time:  0.002
OK (4 tests)
*** Total number of executed method calls:  4
*** Total number of executed catch blocks:  4
*** Tests with uncovered catch blocks to execute:  0
```

Figure 3: Enforcer output for the example test suite (omitting the standard error channel).

3 Advanced Usage/Details

3.1 Configuration options

Various configuration options, to fine-tune or debug Enforcer, exist. They are documented at the end of the `README` file. One more complex option, the suppression of stack traces, is explained in more detail below.

3.2 Suppression of stack traces

Exception handlers often handle an exception locally before escalating that exception to its caller. The latter aspect of this practice is sometimes referred to as re-throwing an exception, and is quite common [1]. In software that is still under development, such exception handlers often include some auxiliary output such as a dump of the stack trace. The reasoning is that such handlers are normally not triggered in a production environment, and if triggered, the stack trace will give the developer some immediate feedback about how the exception occurred. Several projects investigated in a previous case study used this development approach [1]. If no fault injection tool is used, then this output will never appear during testing and therefore does not constitute a problem in practice.

Unfortunately, this methodology also entails that a fault injection tool will generate a lot of output on the screen if it is used on such an application. While it is desirable to test the behavior of all exception handlers, the output containing the origin of an exception typically does not add any useful information. If an uncaught exception occurs during unit testing, the test in which it occurs is already reported by the JUnit test runner. Conversely, exceptions which are caught and then re-thrown do not have to be reported unless the goal is to get some immediate visual information about when the exception is handled (as debugging output).

The stack trace reported by such debugging output may be rather long, and can make it difficult to evaluate the new test log when Enforcer is used. Therefore, it may be desirable to suppress all exception stack traces that are directly caused by injected faults. The latest version of the Enforcer tool implements this feature. “Primary” exceptions, which were injected into the code, are not shown; “secondary” exceptions, which result as a consequence of an incorrectly handled injected fault, are reported. In most cases, this allows for a much easier evaluation of the output. Of course it is still possible to turn this suppression feature off in case a complete report is desired, by setting environment variable `ENFORCER_SUPPR_ARTIF_EXC` to `0`.

3.3 Coverage of exceptions

Coverage information describes whether a certain piece of code has been executed or not. Enforcer considers the coverage of method calls where an exception may occur. The goal is to test program behavior at each location where exceptions are handled, for each possible occurrence of an exception. This corresponds to the *all-e-deacts* criterion [3]. Treating each checked method call individually allows distinction between exception handling before and after a resource, or several resources, have been allocated. Figure 4 illustrates the purpose of this coverage criterion: In the given `try`/`catch` block, two operations may fail. Both the `read` and the `close` operation may throw an `IOException`. Given the fact that the application is likely going to be in a very different state before and after reading from the resource, and again after having closed the stream, it is desirable to test three possible scenarios: Successful execution of statements (1) and (2), success of statement (1) but failure of statement (2), and failure of statement (1), whereupon statement (2) is never executed. This corresponds to full branch coverage if the same semantics are encoded using `if` statements.

```
try {
    /* (1) */ input = stream.read();
    /* (2) */ stream.close();
} catch (IOException e) {
    // exception handling
}
```

Figure 4: Illustration of the all-e-deacts coverage criterion.

Enforcer also supports nested exceptions, with a nesting depth of up to two exceptions [1]. Deeper nesting can be implemented in principle, with a slightly higher overhead at run-time.

References

[1] C. Artho, A. Biere, and S. Honiden. Enforcer – efficient failure injection. In *Proc. Int’l Conference on Formal Methods (FM 2006)*, Canada, 2006.

[2] C. Artho, A. Biere, and S. Honiden. Exhaustive testing of exception handlers with enforcer. *Post-proceedings of 5th Int. Symposium on Formal Methods for Components and Objects (FMCO 2006)*, 4709:26–46, 2006.

[3] S. Sinha and M. Harrold. Criteria for testing exception-handling constructs in Java programs. In *Proc. IEEE Int'l Conf. on Software Maintenance (ICSM 1999)*, page 265, Washington, USA, 1999. IEEE Computer Society.

[4] Sun Microsystems, Santa Clara, USA. *Java 2 Platform Standard Edition (J2SE) 1.5*, 2004. `http://java.sun.com/j2se/1.5.0/`.