Testing I/O Failures with Enforcer

Cyrille Artho

National Institute of Informatics, Tokyo, Japan

Abstract. Testing application behavior in the presence of I/O failures is extremely difficult. The resources used for testing usually work without failure. Failures typically cannot be initiated on the test suite level and are usually not tested sufficiently. Essentially, each interaction of the application with the environment can result in a failure, making failures hard to test. The Enforcer tool identifies such potential failures and automatically tests all relevant outcomes of such actions. It combines the structure of unit tests, coverage information, and fault injection. By taking advantage of a unit test infrastructure, performance can be improved by orders of magnitude compared to previous approaches. This paper introduces the usage of the Enforcer tool.

1 Introduction

Testing is a scalable, economic, and effective way to uncover faults in software [4]. Coverage measurement tools provide a quantitative measure of the quality of a test suite [4]. Uncovered (and thus untested) code may still contain faults. A severe limitation of testing is non-determinism, given by both the thread schedule and the actions of the environment. 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.

Similar tools exist that inject faults into the program and thus improve coverage of exception handlers [2,3]. However, previous tools have not taken the structure of unit tests into account and thus required re-running the entire test suite for each uncovered exception. Therefore, for m unit tests and n uncovered exceptions, previous approaches had a run-time of $O(m \cdot n)$. Enforcer only repeats one unit test per uncovered exception, yielding a run-time of $O(m+n)$, improving performance by several orders of magnitude [1]. The tool operates in three stages, which are described in detail in previous work [1]:

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, forcing execution to take new paths. This has to be taken into account by coverage measurement code, in order to require only a single instrumentation step.

2 Usage of the Enforcer tool

Fundamentally, the three steps described above 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 executed at compile-time, after compilation of the application source code, or at class load time. Enforcer supports both modes of operation. 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. Alternatively, Enforcer can be invoked at load-time. In this mode, the new Java instrumentation agent mechanism is used [5]. 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 not yet very mature and sometimes produces incorrect results when dynamic class loading 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 1).

The output can be interpreted as follows: 37 method calls that declare exceptions were present in the code executed. Out of these, 32 were actually executed, and five belong to untested or dead code. Two uncovered paths from an executed method call to their corresponding exception handler exist. Therefore two tests are run again; the

```
Time: 0.402
OK (29 tests)
*** Total number of instrumented method calls: 37
*** Total number of executed method calls: 32
*** Total number of executed catch blocks: 30
*** Tests with uncovered catch blocks to execute: 2
..
Time: 0.001
OK (2 tests)
*** Total number of executed method calls: 32
*** Total number of executed catch blocks: 32
*** Tests with uncovered catch blocks to execute: 0
```

Fig. 1. Enforcer output when running the wrapped JUnit test suite.

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].

Table 1 shows the condensed results of experiments performed [1]. For each case, the number of tests, the time to run the tests, and the time to run them under Enforcer (where the outcome of exceptions are tested in addition to normal testing) are shown. Note that out of a certain number of calls to I/O methods, typically only a small fraction are covered by original tests. Enforcer can cover most of the missing calls, at an acceptable run-time overhead of factor 1.5 – 5. With previous tools [2,3], re-execution of the entire test suite would have lead to an overhead proportional to the number of test cases, which is orders of magnitudes higher even for small projects. For instance, the Informa test suite comprises 119 tests taking about 33 seconds to run. When analyzing that test suite with a previous-generation fault injection tool, the entire test suite would have had to be run another 136 times (once for each unexecuted exception), taking at about an hour and a half, rather than three minutes when using Enforcer.

Table 1. Results of unit tests and injected exception coverage.

Application or library	# tests	Time [s]	Time, Enforcer [s]	# exec. calls	# unex. catch	Cov. (orig.)	Cov. (Enforcer)
Echomine	170	6.3	8.0	61	54	8 %	100 %
Informa	119	33.2	166.6	139	136	2 %	80 %
jZonic-cache	16	0.4	0.7	8	6	25 %	100 %
SFUtils	11	76.3	81.6	6	2	67 %	100 %
SixBS	30	34.6	94.3	31	28	10 %	94 %
STUN	14	0.06	0.7	0	0	0 %	0 %
XTC	294	28.8	35.5	112	112	0 %	92 %

3 Conclusions and Future work

Enforcer covers potential I/O failures of an application. Coverage measurement determines which method calls may fail, but did not fail under a given test run. After execution of the default test suite, a new test suite containing candidates for improved coverage is automatically constructed and executed. Exceptions are simulated through fault injection. Future work includes precision and performance improvements.

Currently a potential for false positives exists because the exact type of a method is not always known at instrumentation time. Instrumentation then conservatively assumes that I/O failures are possible in such methods. This precision could be improved by adding a run-time check that verifies the signature of the actual method called. Of course this would incur some extra overhead.

While the performance of the tool is very satisfactory, and coverage measurement incurs little overhead, one improvement is still possible. In the existing version, the unit test currently running is determined via reflection techniques. This is somewhat expensive and uses functions that are only available in JDK 1.5 or newer. Moving this functionality into the test wrapper is both conceptually more elegant, slightly faster, and will enable Enforcer to run on older versions of Java.

References

1. C. Artho, A. Biere, and S. Honiden. Enforcer – efficient failure injection. In *Proc. Intl. Conference on Formal Methods (FM 2006)*, Canada, 2006.
2. G. Candea, M. Delgado, M. Chen, and A. Fox. Automatic failure-path inference: A generic introspection technique for Internet applications. In *Proc. 3rd IEEE Workshop on Internet Applications (WIAPP 2003)*, page 132, Washington, USA, 2003. IEEE Computer Society.
3. C. Fu, B. Ryder, A. Milanova, and D. Wonnacott. Testing of Java web services for robustness. In *Proc. ACM/SIGSOFT Intl. Symposium on Software Testing and Analysis (ISSTA 2004)*, pages 23–34, Boston, USA, 2004.
4. D. Peled. *Software Reliability Methods*. Springer, 2001.
5. Sun Microsystems, Santa Clara, USA. *Java 2 Platform Standard Edition (J2SE) 1.5*, 2004. `http://java.sun.com/j2se/1.5.0/`.