# Enforcer – Efficient Failure Injection

Cyrille Artho[1], Armin Biere[2], and Shinichi Honiden[1]

[1] National Institute of Informatics, Tokyo, Japan
[2] Johannes Kepler University, Linz, Austria

**Abstract.** Non-determinism of the thread schedule is a well-known problem in concurrent programming. However, other sources of non-determinism exist which cannot be controlled by an application, such as network availability. Testing a program with its communication resources being unavailable is difficult, as it requires a change on the host system, which has to be coordinated with the test suite. Essentially, each interaction of the application with the environment can result in a failure. Only some of these failures can be tested. Our work identifies such potential failures and develops a strategy for testing all relevant outcomes of such actions. Our tool, Enforcer, 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. Our tool has been tested on several real-world programs, where it found faults without requiring extra test code.

## 1 Introduction

Testing is a scalable, economic, and effective way to uncover faults in software [19,21]. Even though it is limited to a finite set of example scenarios, it is very flexible and by far the most widespread quality assurance method today. Testing is often carried out without formal rigor. However, coverage measurement tools provide a quantitative measure of the quality of a test suite [7,21]. Uncovered (and thus untested) code may still contain faults.

In practice, the most severe limitation of testing is non-determinism, given by both the thread schedule and the actions of the environment. It may cause a program to produce different results under different schedules, even with the same input. Non-determinism has been used in model checking to model choices that have several possible outcomes [25]. Usually, three kinds of non-determinism are distinguished [20]: non-determinism arising from different thread schedules, from choices made by the environment, and from abstractions within an application. The latter is an artefact of abstraction in static analysis and not of concern here. Non-determinism arising from different thread schedules has been tackled by previous work in run-time verification and is subject to ongoing study [1,23]. This paper focuses on non-determinism arising from unexpected failures by the environment, such as system library calls.

For system calls, there are usually 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. If a mechanism exists, though, testing both outcomes will be very efficient, only requiring a duplication of a

particular test case. Existing ad-hoc approaches include factoring out small blocks of code in order to manually test error handlers, or adding extra flags to conditionals that could trigger outcomes that are normally not reachable by modelling test data alone. Figure 1 illustrates this. In the first example, any exception handling is performed by a special method, which can be tested separately, but does not have access to local variables used by the caller. In the second example, which has inspired our work, the unit test has to set a special flag which causes the error handling code to run artificially.

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

Factoring out error handling.

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

Extra conditional for testing.

**Fig. 1.** Two manual approaches for exception handler coverage.

The Java programming language uses exceptions to signal failure of a library or system call [12]. The ideas in this paper are applicable to any other programming language suppporting exceptions, such as C++ [24], Eiffel [17], or C# [18]. When an exception is thrown, the current stack frame is cleared, and its content replaced with a single instance of type `Exception`. This mechanism helps our goal in two ways:

- Detection of potentially failed system calls is reduced to the analysis of exceptions.
- No special context data is needed except for information contained in the method signature and the exception.

Our tool is built on these observations. It systematically analyzes a program for untested exceptional outcomes of library method calls by using fault injection [13]. Automatically instrumented code measures coverage of unit tests w.r.t. exceptions, utilizing the Java reflection API to extract information about the current test case. After execution of the test suite, a number of tests is re-executed with fault injection enabled, triggering previously untested exceptions. Our tool wraps invocation of repeated tests automatically, i.e., only one launch of the test suite is required by the user.

Similar tools have analyzed exception handling in Java code and improved coverage by fault injection [4,11]. Previous tools have not been able to connect information about unit tests with exception coverage. Our tool gathers method signature information statically and the remaining data at run-time. Being integrated with unit testing, it avoids re-executing the entire program many times, and therefore can scale to test suites of large programs. It also supports tuples of failures when analyzing test outcomes at run-time. Our tool is fully automated and can test the outcome of significant failure scenarios in real software. By doing so, it finds faults in previously untested code, without requiring a single extra line in the test setup or test code.

The contribution of our work is as follows:

- We present a fully automated, high-performance approach at gathering specialized coverage information which is integrated with JUnit.
- Fault injection is based on a combined static and dynamic analysis.
- Tuples of faults are supported based on dynamically gathered data.

Section 2 gives the necessary background about sources of failures considered here, and possible implementation approaches. Section 3 describes our implementation used for experiments, of which the results are given in Section 4. Section 5 describes related work. Section 6 concludes and outlines future work.

## 2 Background

An *exception* as commonly used in many programming languages [12,17,18,24] indicates an extraordinary condition in the program, such as the inavailability of a resource. Exceptions are used instead of error codes to return information about the reason why a method call failed. Java also supports *errors*, which indicate "serious problems that a reasonable application should not try to catch" [12]. A method call that fails may "throw" an exception by constructing a new instance of `java.lang.Exception` or a subtype thereof, and using a `throw` statement to "return" this exception to the caller. At the call site, the exception will override normal control flow. The caller may install an exception *handler* by using the `try`/`catch` statement. A `try` block includes a sequence of operations that may fail. Upon failure, remaining instructions of the `try` block are skipped, the current method stack frame is replaced by a stack frame containing only the new exception, and control is transferred to the exception handler, indicated in Java by the corresponding `catch` block. This process will also be referred to as *error handling*.

The usage and semantics of exceptions covers a wide range of behaviors. In Java, exceptions are used to signal the unavailability of a resource (e.g., when a file is not found or cannot be written), failure of a communication (e.g., when a socket connection is closed), when data does not have the expected format, or simply for programming errors such as accessing an array at an illegal index. Two fundamentally different types of exceptions can be distinguished: *Unchecked* exceptions and *checked* exceptions. Unchecked exceptions are of type `RuntimeException` and do not have to be declared in a method. They typically concern programming errors, such as array bounds overflows, and can be tested through conventional means. On the other hand, checked exceptions have to be declared by a method which may throw them. Failure of external operations results in such checked exceptions [4,10]. This work therefore focuses on checked exceptions. For the remainder of this paper, a *checked method call* refers to a call to a method which declared checked exceptions.

*Code instrumentation* consists of injecting additional code into an application, adding extra behavior to it while not changing the original behavior or only changing it in a very limited way. It corresponds to a generic form of aspect-oriented programming [14], which organizes code instrumentation into a finite set of operations. A *unit test* is a procedure to verify individual modules of code. A *test harness* executes unit tests. *Test suites* combine multiple unit tests into a single set. Execution of a single unit test is

defined as *test execution*, running all unit tests as *test suite execution*. In this paper, a *repeated test suite* denotes an automatically generated test suite that will re-execute certain unit tests, which will be referred to as repeated tests.

Program steering [15] allows overriding normal execution flow. Program steering typically refers to altering program behavior using application-specific properties [15], or as schedule perturbation [23], which covers non-determinism in thread schedules. Fault injection [13] refers to influencing program behavior by simulations failures in hardware or software.

*Coverage* information describes whether a certain piece of code has been executed or not. In this paper, only coverage of checked method calls is relevant. The goal of our work was 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 [22]. Treating each checked method call individually allows distinction between error handling before and after a resource, or several resources, have been allocated.

The first source of potential failures considered here are input/output (I/O) failures, particularly on networks. The problem is that a test environment is typically set up to test the normal behavior of a program. While it is possible to temporarily disable the required resources by software, such as shell scripts, such actions often affect the entire system running, not just the current application. Furthermore, it is difficult and error-prone to coordinate such system-wide changes with a test harness. The same applies to certain other types of I/O failures, such as running out of disk space, packet loss on a UDP connection, or communication timeout. While the presence of key actions such as resource deallocations can be checked statically [6,26], static analysis is imprecise in the presence of complex data structures. Testing can analyze the exact behavior.

The second goal is to cover potential failures of external programs. It is always possible that a system call fails due to insufficient resources or for other reasons. Testing such failures when interacting with a program through inter-process communication such as pipes is difficult and results in much testing-specific code.

Our tool, Enforcer, is written in Java and geared towards failures which are signalled by Java exceptions. Some operations showing a similar behavior are not available in Java: In C programs, pointer arithmetic can be used. The exact address returned by memory allocation cannot be predicted by the application, causing portability and testing problems for low-level operations such as sorting data by their physical address. Other low-level operations such as floating point calculations may also have different outcomes on different platforms.

The idea of using program steering to simulate rare outcomes may even be expanded further. Previous work has made initial steps towards verifying the contract required by hashing and comparison functions, which states that equal data must result in equal hash codes, but equal hash codes do not necessarily imply data equality [2,12]. The latter case is known as a hash code collision, where two objects containing different data have the same hash code. This case cannot be tested effectively since hash keys may vary on different platforms and test cases to provoke such a collision are hard to write for non-trivial hash functions, and practically impossible for hash functions that are cryptographically secure. Other mathematical algorithms have similar properties, and are subject of future work.

## 3 Implementation

Java-based applications using JUnit [16] for unit testing have been chosen as the target for this study. Java bytecode is easy to understand and well-documented. JUnit is widely used for unit testing. In terms of programming constructs, the target consists of any unthrown exceptions, i.e., checked method calls where a corresponding `catch` statement exists and that `catch` statement was not reached from an exception originating from said method call. Only checked exceptions were considered because other exceptions can be triggered through conventional testing [4,10]. Artificially generated exceptions are initialized with a special string denoting that this exception was triggered by Enforcer.

A key goal of the tool is not to have to re-execute the entire test suite after coverage measurement. Therefore the project executes in three stages:

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.

As a consequence of treating each checked method call rather than just each unit test individually, a more fine-grained behavior is achieved. Each unit test may execute several checked method calls. Our approach allows for re-executing individual unit tests several times within the repeated test suite, injecting a different exception each time. This achieves better control of application behavior, as the remaining execution path after an exception is thrown likely no longer coincides with the original test execution. Furthermore, it simplifies debugging, since the behavior of the application is generally changed in only one location for each repeated test execution. Unit tests themselves are excluded from coverage measurement and fault injection, as exception handlers within unit tests serve for diagnostics and are not part of the actual application. We did not consider random fault injection [8], as our goal is to achieve high coverage in a reliable way, and to take advantage of the structure of unit tests for making fault injection scalable. Simply injecting exceptions at random would require re-running the entire test suite, and does not necessarily guarantee high coverage.

The intent behind the creation of the Enforcer tool is to use technologies that can be combined with other approaches, such that the system under test (SUT) can be tested in a way that is as close to the original test setup as possible, while still allowing for full automation of the process. Code instrumentation fulfills this requirement perfectly, since the code generated can be executed on the same platform as the original SUT. Instrumentation is performed directly on Java bytecode [27]. This has the advantage that the source code of libraries is not required.

### 3.1 Re-execution of test cases

After execution of the original test suite, coverage information is evaluted. For each exception that was not thrown, the test case that covered the corresponding checked

method call is added to the repeated test suite. Execution of the repeated test suite follows directly after coverage evaluation. Instrumented code handling test execution re-executes the repeated test suite as long as uncovered exceptions exist, and progress is being made w.r.t. coverage (for nested `try`/`catch` blocks, see below). Each time, a new repeated test suite is constructed on the fly by the Enforcer run-time library, and then executed.

### 3.2 Injecting exceptions

The final change to the application by instrumentation will force tests to go through a different path when re-executing. Two points have to be taken into consideration: Where the exception should be thrown, and how.

In our implementation, exceptions are triggered just before checked method calls. A `try` block may include several checked method calls. By generating an exception before each corresponding checked method call, steering simulates actions that were successful up to the last critical operation. If the program is deterministic, it can be assumed that the `try` block will not fail before that point in repeated test execution, as all inputs leading up to that point are equal.

```
try {

    /* fault injection code */
    if (enforcer.rt.Eval.reRunID == __ID__) { // __ID__ = static
        throw new ...Exception();
        // Exception type depends on catch block argument.
    }
    curr_id = __ID__; /* to register exception coverage */

    /* checked method call in the original code */
    call_method_that_declares_checked_exceptions();

    /* coverage code */
    enforcer.rt.Coverage.recordMethodCoverage(__ID__);

    // same instrumentation for each checked method call

} catch(...Exception e) {
    enforcer.rt.Coverage.recordCatchCoverage[curr_id] = true;
    // one instrumentation for each catch block

    /* original catch block follows */
}
```

**Fig. 2.** Instrumented code in `try`/`finally` blocks.

Generating exceptions when running the repeated test suite is achieved by inserting code before and after checked method calls. It is possible that the same test case calls several such methods, but only a single exception should be artificially triggered for

each test execution. Achieving this is difficult because the checked method call ID is not known by the test suite or the test case at run time. Due to this, a test wrapper is used to wrap each test and set the necessary steering information prior to each individual test execution. Figure 2 shows the resulting code to be added to each `try`/`catch` block, which records coverage in the initial test execution and applies program steering when executing the repeated test suite. At each checked method call, code is inserted before and after that method call. Note that the value of `__ID__` is determined statically and replaced by a unique constant each time when instrumentation takes place.

The inserted code before each checked method call injects faults. It compares its static ID to the index of the exception to be generated. This index, `reRunID`, is set by the test wrapper. Due to the uniqueness of the ID, it is therefore possible to instrument many checked method calls, but still only inject a fault in a single such method call. If the IDs match, an exception of the appropriate type is constructed and thrown. A number of possible constructors for exception instances are supported, covering all commonly used exception constructors where reasonable default arguments can be applied. Sometimes the signature of a called method cannot be determined at compile time. In such cases it is conservatively assumed that the method may throw an exception of the type declared in the `catch` clause.[3]

### 3.3 Coverage measurement

Coverage of exceptions thrown is recorded by instrumented code inside each `try` block, and at the beginning of each `catch` block. Coverage within `try` blocks is recorded as follows: Whenever a checked method call that may throw an exception returned successfully, the test case further up in the calling chain is recorded, such that this test case can be re-run later. This is performed by a call to the Enforcer run-time library with the static ID of the checked method call as argument (see Figure 2). The run-time library evaluates the stack trace in order to find the class and method name of the current unit test.

Coverage information about executed exception handlers is recorded by inserting code at the beginning of each `catch` block. Before each checked method call, the ID of that method is stored in local variable `curr_id`. This allows the coverage measurement code within the exception handler to know which checked method caused an exception. A `try` block may contain several checked method calls, each one requiring instrumentation; the corresponding `catch` block, however, only requires a single instrumentation, because the usage of `curr_id` allows for registering coverage of several checked method calls.

### 3.4 Extension to nested exception handlers

Nested exceptions can be responsible for program behavior that only occurs in extremely rare circumstances, such as when both a disk and a network failure are present.

[3] This assumption has to be made if the type of the method cannot be determined due to incompleteness of alias analysis, or usage of dynamic class loading. It may introduce false positives.

A graceful recovery from such failures is difficult to implement, and therefore we found it very important to support combined failures by injection of tuples of faults.

Nested `try` statements cause no additional problems for the algorithm described above. Figure 3 shows an example with two nested `try` blocks. There are three possible final values for *i* in this program: 2, when no exception occurs; 3, when the inner exception $e_2$ occurs; and 4, if the outer exception $e_1$ is thrown. Both `try` statements are reachable when exceptions are absent. Therefore, if either $e_1$ or $e_2$ are not covered by the normal test suite, our algorithm either forces $e_1$ after *i* has been set to 1, or $e_2$ when *i* equals 2.

```
int i = 0;
try {                          // try 1
    call_method_throwing_exceptions();
    i = 1;
    try {                      // try 2
        call_method_throwing_exceptions();
        i = 2;
    } catch (Exception e2) { // catch 2
        i = 3;
    }
} catch (Exception e1) {     // catch 1
    i = 4;
}
```

**Fig. 3.** Nested `try` statements.

However, the design described so far is limited to `try` blocks which do not occur inside other exception handlers. Fortunately, even this case of nesting can be covered quite elegantly. In nested `try` blocks, execution of the inner `try` block may depend on the outer `catch` block being executed. Suppose the outer `catch` block is not executed by initial tests, but only by the repeated test suite. The repeated test suite may again not cover the inner `catch` block. Figure 4 illustrates such difficulties arising with nested `try`/`catch` statements. The compiler generates two exception handlers for this code.

When no exceptions occur in this example, the final value of *i* equals 1. Let us call that scenario run 0, the default test execution without steering. Subsequent re-runs of this test will try to force execution through each `catch` block. The outer `catch` blocks can be triggered with the algorithm described so far. Repeated test execution 1 thus forces corresponding `catch` clause 1 to be executed, setting *i* to 2. Furthermore, coverage measurement will now register the repeated test as a candidate for covering `catch` block 2. This will constitute the new repeated test suite containing run 2, which has the goal of forcing `catch` block 2 to be reached. However, injecting exception $e_2$ requires reaching `catch` block 1. This is only the case in run 1; run 2 therefore would never reach the fault injection code if only $e_2$ was injected. In order to solve this problem, one has to inject *sets* of faults, not just single faults. In the example of Figure 4, $e_1$ has to be injected for both runs 1 and 2. Coverage measurement in run 1 registers that run 1 has executed `try` block 2; therefore both $e_1$ and $e_2$ are injected in

```
int i = 0;
try {                        // try 1
    call_method_throwing_exceptions();
    i = 1;
} catch (Exception e1) {     // catch 1
    try {                    // try 2
        call_method_throwing_exceptions();
        i = 2;
    } catch (Exception e2) { // catch 2
        i = 3;
    }
}
```

**Fig. 4.** A `try` block inside an exception handler.

run 2. In our implementation, we restricted the nesting depth of exception handlers to one, as this does not require nested dynamic data structures for the run-time library. In practice, a nesting depth greater than two is rare, and can be supported by using vectors of sets.

Because of such initially uncovered `try` blocks, coverage of nested exceptions may require the construction of several repeated test suites. The first one includes a unit test for each uncovered checked method call. Execution of this repeated test suite may cover other previously unreached `try` blocks, which are target of the next iteration. The iteration of repeated test suites terminates when no progress is made for coverage. Hence, certain unit tests may be executed several times within the same iteration (for different exceptions) and across iterations.

### 3.5 Complexity

The complexity incurred by our approach can be divided into two parts: Coverage measurement, and construction and execution of repeated test suites. Coverage is measured for each checked method call. The code which updates run-time data structures runs in constant time. This overhead is of coverage measurement is proportional to the number checked method calls executed at run-time.

Execution of repeated test suites may incur a larger overhead. For each uncovered exception, a unit test has to be re-executed. However, each uncovered exception incurs at most one repeated test. Nested exceptions may require multiple injected faults for a repeated test. The key to a good performance is that only one unit test, which is known to execute the checked method call in question, is repeated. Large projects contain hundreds or thousands of unit tests; previous approaches [4,10,11] would re-execute them all for each possible failure, while our tool only re-executes one unit test for each failure. This improves performance by several orders of magnitude and allows our tool to scale up to large test suites. Moreoever, the situation is even more favorable when comparing repeated tests with an ideal test suite featuring full coverage of exceptions in checked method calls. Automatic repeated execution of test cases does not require significantly more time than such an ideal test suite, because the only minor overhead

that could be eliminated lies in the instrumented code. Compared to manual approaches, our approach finds faults without incurring a significant overhead, with the additional capability of covering outcomes that are not directly testable.

## 4 Experiments

To ensure solid quality of the implementation, 30 test classes were written to test different aspects and problem cases for code instrumentation, coverage measurement, and test execution. Due to rigorous testing, the tool is mature enough to be applicable to large and complex programs. Therefore, several real-world applications and libraries were used to demonstrate the usefulness of the approach. Unfortunately, realistic Java programs using JUnit-based test suites are hard to come by. A web search for Java applications and JUnit returns tools and libraries enhancing JUnit, but not applications using it. Therefore a different approach was chosen: Based on the listing of all Java program on freshmeat.net [9], 1647 direct links to downloadable archives could be extracted. These resulted in 926 successful automatic downloads, where no registration or manual redirection was used. Out of these applications, 100 used JUnit test suites and also employed at least some networking functionality. Further criteria, such as the use of multiple threads and the absence of a GUI, were used to narrow down the selection to 29 applications. Out of these, nine could be compiled and run on Java 1.5 with no or minor modifications, and no installations of third-party libraries or tools that were not shipped with the original archives.

| Application or library | Description | # classes | Size [LOC] | # test classes | Test code size [LOC] |
|---|---|---|---|---|---|
| Echomine | Communication services API | 144 | 14331 | 46 | 3550 |
| Informa | News channel API | 150 | 20682 | 48 | 6855 |
| jConfig | Configuration library | 77 | 9611 | 39 | 2974 |
| jZonic-cache | Caching library | 26 | 2142 | 14 | 737 |
| SFUtils | Sourceforge utilities | 21 | 6222 | 9 | 1041 |
| SixBS | Java beans persistency | 34 | 4666 | 9 | 1072 |
| Slimdog | Web application testing framework | 30 | 1959 | 11 | 616 |
| STUN | Extensible programming system | 27 | 1706 | 3 | 229 |
| XTC | Napster search tool | 455 | 77114 | 57 | 8070 |

**Table 1.** Applications of which the unit tests were used in the experiments.

The main reason for this low number is the fact that the entire pool of applications included many projects that have been abandoned or not yet been completed. Table 1 shows an overview of the applications used. The first two columns briefly describe each appliation, while the other columns give an indication of the size of each project, showing the number of classes and the lines of code used for them. This information is shown separately for unit test code. The presence of helper classes was responsible for a rather large number of test classes in some cases.

Enforcer was then used on these example applications. Table 2 gives an overview of the test results. Tests were executed on a dual-processor 2.7 GHz PowerPC G5 with 8 GB of RAM and 512 KB of L2 cache per CPU running Mac OS 10.4.5. The table is devided into three parts. The first part shows the test results when running the given test suite. A test failure in JUnit corresponds to an incorrect value of a property, while uncaught exceptions are shown as errors. Note that failures or errors can either be caused due to incorrect code or missing components in the installation. Although it was attempted to fix any installation-related errors, not all cases could be covered.

| Application or library | # tests | # fail. | # err. | Time [s] | Time, instr. [s] | Time, re-ex. [s] | # instr. calls | # exec. calls | # unex. catch | Cov. (orig.) | # unr. catch | Cov. (instr.) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Echomine | 170 | 2 | 0 | 6.3 | 6.3 | 1.7 | 165 | 61 | 54 | 8 % | 0 | 100 % |
| Informa | 119 | 15 | 32 | 33.2 | 34.4 | 132.2 | 306 | 139 | 136 | 2 % | 28 | 80 % |
| jConfig | 97 | 3 | 0 | 2.3 | 4.7 | n/a | 299 | 169 | 162 | 3 % | 65 | 61 % |
| jZonic-c. | 16 | 2 | 0 | 0.4 | 0.7 | 0.02 | 22 | 8 | 6 | 25 % | 0 | 100 % |
| SFUtils | 11 | 1 | 3 | 76.3 | 81.6 | 0.001 | 112 | 6 | 2 | 67 % | 0 | 100 % |
| SixBS | 30 | 0 | 0 | 34.6 | 55.6 | 38.7 | 56 | 31 | 28 | 10 % | 2 | 94 % |
| Slimdog | 10 | 4 | 0 | 228.6 | 233.6 | n/a | 41 | 15 | 14 | 7 % | n/a | n/a |
| STUN | 14 | 0 | 0 | 0.06 | 0.7 | 0 | 2 | 0 | 0 | 0 % | 0 | 0 % |
| XTC | 294 | 0 | 0 | 28.8 | 30.6 | 4.9 | 168 | 112 | 112 | 0 % | 9 | 92 % |

**Table 2.** Results of unit tests and injected exception coverage.

Part two of the table shows the overhead of the instrumentation code for measuring test coverage. Original and instrumented execution time of the normal test suite are shown first.[4] The final execution time measurement shows the time needed to execute repeated test suites. This figure depends much on the coverage of the test suite and the nature of exception handlers, and is given for completeness; it cannot be used to draw conclusions about the quality of the test suite or the Enforcer tool. A better measure is actual coverage of exceptions in checked method calls, as shown by part three of Table 2.

Part three shows details about code instrumentation and coverage. The number of instrumented checked method calls is given first, followed by the number of checked method calls executed by unit tests. Usually a large number of checked method calls never triggered an exception, as shown by the next column, “unexec. catch”. The following column indicates the percentage of executed checked method calls that did generate an exception. As can be seen, that percentage is typically very low. These untested exception cases may each cause previously undiscovered failures and were targetted by the Enforcer tool. In most cases, Enforcer could successfully force inject exceptions; in some cases, deeply nested exceptions or the lack of a fully deterministic test setup prevented full coverage. The rightmost two columns show the number of such uncovered checked method calls, and the final exception coverage after Enforcer was used.

[4] The time required for code instrumentation itself was negligible.

As can be easily seen, Enforcer could often change a nearly nonexistent coverage to a nearly full coverage. However, it depends on a test suite that is able to execute checked method calls in the first place. This criterion is fulfilled if full statement coverage is achieved, which is often the case for larger projects [1] but was not the case for the given programs.

In some cases, injected exceptions affected background threads that were assumed to be running throughout multiple test cases. When these threads failed to terminate properly, or to restart, the test suite would wait indefinitely for them. This was the case for applications jConfig and Slimdog. In jConfig, such problems prevented higher coverage. For Slimdog, two tests had to be disabled even when running without instrumentation, because the multi-threaded test code was too fragile to execute reliably. In test setup, the background thread may allocate a port but then fail to complete initialization, throwing an exception. JUnit does not release any resources allocated in such a failed setup. This problem has been discussed in the mailing list and is going to be addressed in the future. Stopping and restarting the background thread before each test run is expected to fix this problem, at the cost of slowing down test execution.

The overhead caused by coverage measurement was usually negligible, as can be seen by comparing columns one and two of part two of Table 2. SixBS is an exception, where coverage measurement caused a resulting overhead of factor two. The reason for this is that instrumentation significantly increased the run time of the thread controlling the XML parser. This thread contains several exception handlers but relatively little other code, hence amplifying the usual effect of instrumentation on run-time. Reducing the overhead is going to entail the use of additional data structures in order to avoid expensive calls to the Java reflection API at run time.

Our tool generated a total number of 352 exceptions for checked method calls in all applications. The majority of these exceptions (200 instances) concerned I/O, either on a network or a file. 56 exceptions were generated as parse exceptions, while 69 exceptions were of generic type `java.lang.Exception` and could not be classified more closely. Finally, 27 exceptions were of other types, such as `IllegalAccessException`. Exceptions that do not concern I/O were not originally the target of our tool. Nonetheless, the fact that these were also triggered frequently shows that our tool may partially replace test case generation when no tests exist for certain exceptional scenarios.

In most of the 352 cases where an exception was injected, the application ultimately rethrows the exception in question, usually in a slightly different form. It was not possible for us to tell whether this simple behavior was adequate. Because these exceptions were encountered within unit tests, it is possible that the main application front end performs some kind of cleanup before shutting down. However, in general, a call to a low-level library should take exceptions into account. Otherwise, an I/O exception can lead to the termination of the entire thread, and usually the entire program. If untested parts of the application catch such exceptions where unit tests do not, then the unit tests are incomplete since they do not reflect the behavior of the application, failing to account for exceptional behavior. However, considering the fact that some benchmark programs were libraries to be used by an application, rethrowing exceptions may be acceptable in some cases. Therefore we did not analyze these 352 cases in detail. Many of them were redundant, as triggering the same exception handlers from different places in

the same `try` block often produces equivalent results. Some cases were false positives arising from incomplete type information at instrumentation time.

Much more interesting than rethrown exceptions were exceptions that were triggered by failed error handling. These exceptions were not just rethrown, but caused by another part of the program that tried to deal with the initial exceptions.[5] A few of these cases resulted in rethrown exceptions, which were not counted for the reasons stated above. Table 3 shows the failures resulting from incorrect error handlers. Each unique program location was only counted once. We found 12 faults in the nine given applications this way. As can be seen, the lack of testing in error handlers caused typical programming errors to appear (null pointers, illegal arguments, failed class casts). In applications jConfig and Slimdog, the error handling code tried to re-open a socket that was already in use, which resulted in termination of the entire test suite. That defect therefore masked other potential failures. Informa contained various problems in its fallback code concerning I/O (file not found, generic I/O exception, feed manager failure). These problems could perhaps be solved by a different configuration; we used the configuration that came with the default installation. Certainly, it is clear that for some of the given applications, our tool did not only significantly improve coverage of exceptions, but also found several defects in the code.

| App./lib. | FileNotFound | NullPointer | IO | FeedManager | IllegalArgument | Bind | ClassCast | Total |
|---|---|---|---|---|---|---|---|---|
| Echomine | | 1 | | | | | 1 | 2 |
| Informa | 1 | 4 | 2 | 1 | 1 | | | 9 |
| jConfig | | | | | | 1 | | 1 |
| Slimdog | | | | | | 1 | | 1 |
| Total | 1 | 5 | 2 | 1 | 1 | 1 | 1 | 12 |

**Table 3.** Failures resulting from incorrect error handling.

To summarize, our tool was very successful in improving the exception coverage of realistic test suites in a variety of projects. Coverage measurement usually only caused a minor overhead. Without writing any additional code, extra faults were found, where error handlers for exceptions contained defects. With the exception of certain multithreading problems, normal operation of the application tests was not affected by steering. Some of the triggered exceptions should be tested by conventional means. It can be expected that a higher-quality test suite will not have any such uncovered exceptions left, so our tool would likely produce even better results for thoroughly tested code.

## 5 Related work

Test cases are typically written as additional program code for the system under test. White-box testing tries to execute as much program code as possible [19]. In traditional

[5] Distinguishing these “secondary” exceptions was trivial as the injected exceptions were all marked as such by having a special message string.

software testing, *coverage* metrics such as statement coverage [7,21] have been used to determine the effectiveness of a test suite. The key problem with software testing is that it cannot guarantee execution of parts of the system where the outcome of a decision is non-deterministic. In multi-threading, the thread schedule affects determinism. For external operations, the small possibility of failure makes testing that case extremely difficult. Traditional testing and test case generation methods are ineffective to solve this problem.

Static analysis investigates properties "at compile time", without executing the actual program. Non-deterministic decisions are explored exhaustively by verifying all possible outcomes. For analyzing whether resources allocated are deallocated correctly, there exist static analysis tools which consider each possible exception location [26]. However, static analysis can only cover a part of the program behavior, such as resource handling. For a more detailed analysis of program behavior, code execution (by testing) is often unavoidable.

Model Checking explores the entire behavior of a system by investigating each reachable state. Model checkers treat non-determinism exhaustively. Results of system-level operations have been successfully modeled this way to detect failures in applications [5] and device drivers [3]. However, model checking suffers from the state space explosion problem: The size of the state space is exponential in the size of the system.

Therefore approaches that directly tackle testing are very promising, as potential failures of library calls are independent of non-deterministic thread scheduling. Such failures can be simulated by fault injection [13]. Random fault injection is a black-box technique and useful on an application level [8]. Our goal was to achieve a high test coverage, and therefore we target white-box testing techniques.

Java is a popular target for measuring and improving error handling, as error handling locations are relatively well defined [4,10,11]. Our approach of measuring exception handler coverage corresponds to the *all-e-deacts* criterion [22]. The static analysis used to determine whether checked method calls may generate exceptions have some similarity with a previous implementation of such a coverage metric [11]. However, our implementation does not aim at a precise instrumentation for the coverage metric. We only target checked exceptions, within the method where they occur. As the generated exceptions are created at the caller site, not in the library method, an interprocedural analysis is not required. Unreachable statements will be reported as instrumented, but uncovered checked method calls. Such uncovered calls never incur an unnecessary test run and are therefore benign, but hint at poor coverage of the test suite. Furthermore, unlike some previous work [11], our tool has a run-time component that registers which unit test may cause an exception. This allows us to re-execute only a particular unit test, which is orders of magnitude more efficient than running the entire test suite for each exception site. Furthermore, our tool can dynamically discover the need for combined occurrences of failures when error handling code should be reached. Such a dynamic analysis is comparable to another fault injection approach [4], but the aim of that project is totally different: It analyzes failure dependencies, while our project targets code execution and improves coverage of error handling code.

Similar code injection techniques are involved in program steering [15], which allows overriding the normal execution flow. However, such steering is usually very prob-

lematic because correct execution of certain basic blocks depends on a semantically consistent program state. Thus program steering has so far only been applied using application-specific properties [15], or as schedule perturbation [23], which only covers non-determinism in thread schedules. Our work is application-independent and targetted to fault injection.

## 6 Conclusions and future work

In software, non-deterministic decisions are not only taken by the thread scheduler, but also by the environment. Calls to system libraries may fail. Such failures can be nearly impossible to test. Our work uses fault injection to achieve coverage of such untestable properties. During test execution, coverage information is gathered. This information is used in a repeated test execution to execute previously untested exception handlers. The process can be fully automated and still leads to meaningful execution of exception handlers. Unlike previous approaches, we take advantage of the structure of unit tests in order to avoid re-execution an entire application. This makes our approach orders of magnitude faster for large test suites. The Enforcer tool which implements this approach has been successfully applied to several complex Java applications. It has executed previously untested error handlers and uncovered several faults. Furthermore, our approach may even partially replace test case generation.

The area of such generic program steering likely has further applications that have not yet been covered. Future work includes elimination of false positives by including run-time information for method calls whose signature is unknown. Another improvement is analysis of test case execution time, in order to select the fastest test case for re-execution. The treatment of difficult-to-test outcomes can be expanded to other properties mentioned in this paper. Finally, we are very interested in applying our Enforcer tool to high-quality commercial test suites. It can be expected that exception coverage will be incomplete but already quite high, unlike in cases tested so far. This will make evaluation of test results more interesting.

## References


1. C. Artho. *Combining Static and Dynamic Analysis to Find Multi-threading Faults Beyond Data Races*. PhD thesis, ETH Zürich, 2005.
2. C. Artho and A. Biere. Applying static analysis to large-scale, multithreaded Java programs. In *Proc. 13th Australian Software Engineering Conference (ASWEC 2001)*, pages 68–75, Canberra, Australia, 2001. IEEE Computer Society Press.
3. T. Ball, A. Podelski, and S. Rajamani. Boolean and Cartesian Abstractions for Model Checking C Programs. In *Proc. 7th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001)*, volume 2031 of *LNCS*, pages 268–285, Genova, Italy, 2001. Springer.
4. 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.
5. C. Colby, P. Godefroid, and L. Jagadeesan. Automatically closing open reactive programs. In *Proc. SIGPLAN Conf. on Programming Language Design and Implementation (PLDI 1998)*, pages 345–357, Montreal, Canada, 1998.

6. D. Engler and M. Musuvathi. Static analysis versus software model checking for bug finding. In *Proc. 5th Intl. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI 2004)*, volume 2937 of *LNCS*, pages 191–210, Venice, Italy, 2004. Springer.
7. N. Fenton and S. Pfleeger. *Software metrics (2nd Ed.): a rigorous and practical approach*. PWS Publishing Co., Boston, USA, 1997.
8. Justin E. Forrester and Barton P. Miller. An empirical study of the robustness of windows NT applications using random testing. In *4th USENIX Windows System Symposium*, pages 59–68, Seattle, USA, 2000.
9. Freshmeat, 2005. `http://freshmeat.net/`.
10. C. Fu, R. Martin, K. Nagaraja, T. Nguyen, B. Ryder, and D. Wonnacott. Compiler-directed program-fault coverage for highly available Java internet services. In *Proc. 2003 Intl. Conf. on Dependable Systems and Networks (DSN 2003)*, pages 595–604, San Francisco, USA, 2003.
11. 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.
12. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Second Edition*. Addison-Wesley, 2000.
13. M. Hsueh, T. Tsai, and R. Iyer. Fault injection techniques and tools. *IEEE Computer*, 30(4):75–82, 1997.
14. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. *LNCS*, 2072:327–355, 2001.
15. M. Kim, I. Lee, U. Sammapun, J. Shin, and O. Sokolsky. Monitoring, checking, and steering of real-time systems. In *Proc. 2nd Intl. Workshop on Run-time Verification (RV 2002)*, volume 70 of *ENTCS*. Elsevier, 2002.
16. J. Link and P. Fröhlich. *Unit Testing in Java: How Tests Drive the Code*. Morgan Kaufmann Publishers, Inc., 2003.
17. B. Meyer. *Eiffel: the language*. Prentice-Hall, Inc., Upper Saddle River, USA, 1992.
18. Microsoft Corporation. *Microsoft Visual C# .NET Language Reference*. Microsoft Press, Redmond, USA, 2002.
19. G. Myers. *Art of Software Testing*. John Wiley & Sons, Inc., 1979.
20. C. Pasareanu, M. Dwyer, and W. Visser. Finding feasible abstract counter-examples. *Intl. Journal on Software Tools for Technology Transfer (STTT)*, 5(1):34–48, 2003.
21. D. Peled. *Software Reliability Methods*. Springer, 2001.
22. S. Sinha and M. Harrold. Criteria for testing exception-handling constructs in Java programs. In *Proc. IEEE Intl. Conf. on Software Maintenance (ICSM 1999)*, page 265, Washington, USA, 1999. IEEE Computer Society.
23. S. Stoller. Testing concurrent Java programs using randomized scheduling. In *Proc. 2nd Intl. Workshop on Run-time Verification (RV 2002)*, volume 70(4) of *ENTCS*, pages 143–158, Copenhagen, Denmark, 2002. Elsevier.
24. B. Stroustrup. *The C++ Programming Language, Third Edition*. Addison-Wesley Longman Publishing Co., Inc., Boston, USA, 1997.
25. W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2):203–232, 2003.
26. W. Weimer and G. Necula. Finding and preventing run-time error handling mistakes. In *Proc. 19th ACM SIGPLAN Conf. on Object-Oriented Programming Systems, Languages & Applications (OOPSLA 2004)*, pages 419–431, Vancouver, Canada, 2004. ACM Press.
27. A. White. SERP, an Open Source framework for manipulating Java bytecode, 2002. `http://serp.sourceforge.net/`.