Exhaustive Testing of Exception Handlers with Enforcer

Cyrille Artho[1], Armin Biere[2], and Shinichi Honiden[3]

[1] Research Center for Information Security (RCIS),
National Institute of Advanced Industrial Science and Technology (AIST),
Tokyo, Japan
[2] Johannes Kepler University, Linz, Austria
[3] 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. The Enforcer tool identifies such potential failures and automatically tests all relevant outcomes of such actions. It combines the structure of unit tests with 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 [28,29]. 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 [14,29]. Uncovered (and thus untested) code may still contain faults. Tested code is known to have executed successfully under at least one occasion. For this reason, it is desirable to test each block of code at least once.

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 [2]. 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. 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.

Assume that a mechanism exists to identify a test case that executes a particular call. In that case, testing both outcomes of that call 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 exception handlers, or adding extra flags to conditionals that could trigger outcomes that are normally not reachable by modeling 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 exception handling code to run artificially. Enforcer automates the tasks of fault injection and controlling the fault for the selected test cases.

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

Factoring out exception 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.

Similar tools exist that inject faults into the program and thus improve coverage of exception handlers [7,17]. 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 repeats only one unit test per uncovered exception, yielding a run-time of $O(m + n)$, improving performance by several orders of magnitude [2]. The tool operates in three stages, which are described in detail in previous work [2]:

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.

This document as organized as follows: Section 2 gives the necessary background about sources of failures considered here, and possible implementation approaches. Section 3 gives an overview of the implementation of the tool, while Section 4 introduces its usage. Section 5 describes related work. Section 6 concludes, and Section 7 outlines future work.

2 Background

This section introduces the necessary terminology used in the rest of this article, covering exceptions, fault injection, and various technologies used by our approach.

2.1 Exceptions

An *exception* as commonly used in many programming languages [19,26,27,33] indicates an extraordinary condition in the program, such as the unavailability 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" [19]. 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 *exception 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 [7,16]. 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.

When an exception is thrown at run-time, the current stack frame is cleared, and its content is replaced with a single instance of type `Exception` (or a subtype thereof). 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.

2.2 Fault injection

Fault injection [21] refers to influencing program behavior by simulation of failures in hardware or software. In hardware and software, fault injection is most useful when applied to prototypes or implementations. This allows for accurate evaluation of fault handling mechanisms. Fault injection thus also serves to study and measure the quality of exception handling and dependability mechanisms [21].

On a hardware level, fault injection can simulate various kinds of hardware corruption, such as bit flips or stuck-at faults (where the output of a gate is stuck at a particular value). These types of fault injection are important for producing highly reliable hardware, where redundancy can compensate for physical component failures caused by power surges, radiation, or other influences. Certain types of software fault injection have the same target, e. g. simulation of storage data corruption or other machine-level defects [21]. The specific notion of faults in hardware as just described is slightly different from the more common one used in the context of automatic test pattern generation (ATPG). ATPG aims at finding test patterns that verify a chip against faults that are induced during the physical production process [6].

This article targets higher-level problems, arising from failures of system calls. An unexpected result of a system call is typically caused by incorrect parameters or by an underlying network communication problem. Because such failures are difficult to produce, they are often poorly tested. Testing a failure of a system call may take a large amount of effort (during the test process) to set up. In the case of network communications, shutting down network resources will affect other processes on the same host and is therefore not practical in a real-life situation.

Because of these problems, faults are typically injected into an application by an automated tool, which relieves the tester from the burden of injecting faults and capturing their effect. In modern programming languages, such faults manifest themselves on the application level as exceptions [7,16,17]. They can be simulated by modifying the application or library code.

Code instrumentation consists of modifying existing application code or injecting additional code into an application, in order to augment its behavior. The goal is typically to leave the original behavior unchanged, or to change it in a very limited way. It corresponds to a generic form of aspect-oriented programming [22], which organizes code instrumentation into a finite set of operations. *Program steering* [23] allows overriding normal execution flow. Program steering typically refers to altering program behavior using application-specific properties [23], or as schedule perturbation [32], which covers non-determinism in thread schedules. This technique is very similar to fault injection. Instead of faults, either cross-cutting code or randomized delays are injected.

2.3 Problem scope

A *unit test* is a procedure to verify individual modules of code. A *test harness* executes unit tests. *Test suites* combine multiple unit tests (also called *test cases)*

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.*

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 [31]. Treating each checked method call individually allows distinction between exception handling before and after a resource, or several resources, have been allocated. Figure 2 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`. 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. Therefore, 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
}
```

Fig. 2. Illustration of the all-e-deacts coverage criterion.

The first source of potential failures considered by our fault injection tool 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 User Datagram Protocol (UDP) connection, or communication timeout. While the presence of key actions such as resource deallocations can be checked statically [12,36], static analysis is imprecise in the presence of complex data structures. Testing can analyze the exact behavior.

The second source of potential failures are 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 test-specific code.

Our tool, Enforcer, is written in Java and geared towards failures which are signaled by Java exceptions. Certain operations, such as hash code collisions, are also very difficult to test, but do not throw an exception if they occur. A hash code collision occurs when two distinct data objects have the same hash code. Exhaustive testing of such a collision could be addressed by similar means, but is not covered by our tool that deals with exceptions only.

Finally, there exist hard-to-test operations that 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.

3 Implementation

Exceptions can be tested manually. This approach guarantees optimal performance but is not practical when many exceptions have to be tested. Therefore, testing should be automated. Automation is possible for unit test suites having idempotent tests. These are tests that can be executed independently many times, producing the same result each time. For such tests, Enforcer automates testing of exceptional outcomes while still maintaining optimal performance.

3.1 Manual testing of exceptions

The use of unit tests for fault injection has been inspired by unit testing in the JNuke project. JNuke contains a Java model checker implemented in C. Exceptions or failed system calls are handled using conventional control flow constructs. JNuke heavily uses unit testing to ensure its quality [3]. Test coverage of exceptional outcomes is achieved by manual instantiations of a given test.

Figure 3 illustrates manual fault injection in JNuke. Function `unpack` takes a 32-bit integer value containing a floating point number encoded according to the IEEE standard, and returns its floating point value (lines 6 – 20). Internal functions convert this value to a floating point value of the desired precision. This precision can be configured prior to compilation. For performance reasons, floating point values with a precision of 32 bits are usually represented with the same number of bits on the target machine. However, the way floating point numbers are represented may differ, even for the same size. Therefore, it is possible that an underflow occurs for small values: a small number is truncated to zero. This is very rare, but occurred on some less common hardware architectures in our tests. When an underflow is detected, a brief message is printed to the screen (lines 12 – 18).

Most processors will never generate an underflow for 32-bit floating point numbers because an identical representation is used internally. This makes it impossible to test the "exception handler" (in lines 15 – 17) that is triggered for an underflow.

```
/* convert constant pool representation (IEEE 754) to C float */
/* endian conversion performed by caller */

static int force_insufficient_precision = 0;

JFloat unpack (unsigned int bytes) {
    JFloat result;

    result = convert_bytes(bytes);

    /* check against underflow */
    if (underflow_detected(result, bytes)
        /* manual fault injection */
        || force_insufficient_precision) {
        fprintf (stderr, "unpack: Cannot represent float value"
                 " due to insufficient precision.\n");
        return 0;
    }
    return result;
}

/* test cases */
int testSuccess (JNukeTestEnv * env) {
    int res;
    res = !JFloat_compare (unpack(FLOAT_1_0), 1.0);
    return res;
}

int testFailure (JNukeTestEnv * env) {
    int res;
    /* activate fault injection */
    force_insufficient_precision = 1;

    res = !JFloat_compare (unpack(SMALL_FLOAT), 0.0);
    /* return value always == 0 due to fault injection */

    /* disable fault injection */
    force_insufficient_precision = 0;
    return res;
}
```

Fig. 3. Manual testing of a library call that can generate an exception.

In order to avoid a gap in test coverage, fault injection was implemented manually. An extra test executes the "exception handler". The test code is implemented in lines 22 – 40. The first test case, `testSuccess`, succeeds on any platform. The second test case, `testFailure`, is designed to execute the "exception handler" that is triggered on underflow. On some architectures, this happens if a small value known to cause an underflow is used. However, this is not sufficient to test the function in question on most platforms. Therefore, fault injection is used to simulate an underflow (flag `force_insufficient_precision` on lines 14, 32, and 38). This manual approach works fairly well when used sparsely, but carries many intrinsic problems:

- Fault injection code is added inside the application and test code. This reduces readability, and may even interfere with normal code. Non-interference of fault injection code can only be checked through inspection (or the use of preprocessor directives in C).
- Not necessarily the same test case is used to test the success and failure cases.
- Fault injection has to be done manually: Fault injection has to be enabled (and disabled!) at the right points in the test code.

For the remainder of this paper, we will again use Java to illustrate exception handling. Typical code tested by Enforcer uses I/O operations that can throw an exception in Java. Figure 4, which is an extended version of the example in Fig. 1, shows how two unit tests can execute the successful and the failure scenario of a given operation. The application code of this example (lines 1–15) contains a `try` block because the I/O operation in line 10 might fail. Usually, it is successful, so a normal unit test will never execute the corresponding `catch` block.

In order to address this problem, the test code (lines 17 – 27) contains two test cases: The first test case is a conventional unit test, which executes the method in question. Because it is assumed that the I/O operation will return successfully, another test case is added. This test case covers the failure scenario by setting a flag `socketShouldFail` prior to execution of the application code. The artificially induced failure results in an exception being thrown. Again, care has to be taken to reset that flag after test execution. Since that flag is `static` (global), it would affect all subsequent test cases if it was not reset.

In the example in Fig. 4, manual fault injection results in about nine additional lines of code (when not counting whitespace). This 60 % increase is of course an extreme figure that is much lower in average real-world scenarios. In the JNuke project where similar code was used [3], only about 0.25 % of the total code contributes to such testing of I/O failures or similar operations. This figure is rather low because JNuke does not use a lot of I/O operations, and I/O code is limited to 7 out of about 165 classes. Nonetheless, in one class, the extra code needed for a manual approach already affects the readability of the application code, due to various flags and conditionals being introduced. Three methods of the application code of that class have an overhead of over 30 % of extra test

```
 1 class ApplicationCode {
       static boolean socketShouldFail = false;

       public void operationUsingIO() {
 5         ServerSocket socket;
           try {
               if (socketShouldFail) {
                   throw new IOException();
               }
10             socket = new ServerSocket();
           } catch (IOException e) {
               // exception handling code
           }
       }
15 }

   class TestApplicationCode extends TestCase {
       public void testOperationUsingIO_success() {
           operationUsingIO();
20     }

       public void testOperationUsingIO_failure() {
           ApplicationCode.socketShouldFail = true;
           operationUsingIO();
25         ApplicationCode.socketShouldFail = false;
       }
   }
```

Fig. 4. Manual testing of a library call that can generate an exception.

code covering I/O failures. This hampers readability and maintainability of that code, and clearly calls for automation of testing I/O failures.

3.2 Automation

Java-based applications using JUnit [25] 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 [7,16]. Artificially generated exceptions are initialized with a special string denoting that this exception was triggered by Enforcer.

A key goal of the tool is to avoid re-execution of the entire test suite after coverage measurement. In order to achieve this, the test process executes in three stages:

1. Code instrumentation, at compile time or at class load time.
2. Execution of unit tests. Coverage information is now gathered.
3. Re-execution of certain tests, with selective fault injection.

Coverage information gathered during test execution serves to identify a unit test u that executes a checked method call, which could trigger a particular exception e. For each such exception that has not been triggered during normal unit testing, Enforcer then chooses to re-run one unit test (such as u) with exception e being injected. This systematically covers all exceptions by re-running one unit test of choice for each exception.

This approach assumes that unit tests are independent of each other and idempotent. What this means is that a unit test should not depend on data structures that have been set up by a previous unit test. Instead, standardized set-up methods such as `setUp` (in JUnit) must be used. Furthermore, a unit test should not alter persistent data structures permanently, i. e., the global system state after a test has been run should be equal to the system state prior to that test. This behavior is specified in the JUnit contract [25] and usually adhered to in practice, although it is not directly enforced by JUnit. JUnit does not check against global state changes and executes all unit tests in a pre-determined order. Therefore, it is possible to write JUnit test suites that violate this requirement, and it is up to the test engineer to specify correct unit tests. Tools such as DbUnit [11] can ensure that external data, such as tables in a data base, also fulfill the requirement of idempotency, allowing for repeated test execution.

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.

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 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. Instrumentation is performed directly on Java bytecode [37]. Injected code has to be designed carefully for the two dynamic stages to work together. Details about the architecture of the tool are described in previous work [2].

3.3 Comparison to stub-based fault injection

Faults can also be injected at library level. By using a stub that calls the regular code for the successful case and throws an exception for the failure case, code instrumentation would not be necessary. The only change in the code would consist of the "redirection" of the original method call to the stub method. Code modification could therefore be done at the callee rather than at each call site. This way, only library code would be affected, and the application could be run without modification.[4]

A stub-based approach is indeed a much simpler way to achieve fault injection. However, it cannot be used to achieve selective fault injection, which is dependent on coverage information. Coverage of exception handlers (`catch` blocks) concerns method calls and their exception handlers. As there are usually several method calls for a given method, coverage of checked method calls can only be measured at the call site, not inside the callee. The same argument holds for code that performs selective fault injection, using run-time information to activate injected exceptions when necessary.

Therefore, most of the functionality heavily depends on the caller context and concerns the call site. While it is conceivable to push such code into method call (into the library code), this would require code that evaluates the caller context for coverage measurement. Such code requires run-time reflection capabilities and would be more complex than the architecture that is actually implemented [2]. Therefore, it does not make sense to implement unit test coverage inside the library.

For this reason, the application code is modified for coverage measurement. By also implementing fault injection at the caller, modification of the callee (the library) is avoided. Avoiding modification of the library makes it possible to use

[4] This idealizing proposition assumes that the application itself does not provide any library-like functionality, i. e., the application does not interface directly with any system calls or produce other types of exceptions that cannot be tested conventionally.

native and signed (tamper-proof) library code for testing. A stub-base approach cannot modify such code.

Coverage data in Enforcer is context-sensitive. It is not only important which blocks of code (which checked method calls and which exception handlers) are executed, but also by which unit test. This requires more information than a typical coverage measurement tool provides. After all, the test to cover previously uncovered exceptions is chosen based on this coverage information. Such a test, if deterministic, is known to execute the checked method call in question. As coverage information includes a reference to a test case, that test case can be used for fault injection.

3.4 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. Therefore, the overhead of coverage measurement is proportional to the number of checked method calls that are executed at run-time. This figure is negligible in practice.

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, but still only one repeated test per distinct fault is required [2]. 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. If a test suite contains m unit tests and n uncovered exceptions, then our approach will therefore execute $m + n$ unit tests. This number is usually not much larger than m, which makes the tool scalable to large test suites.

Large projects contain thousands of unit tests; previous approaches [7,16,17] would re-execute them all for each possible failure, repeating m tests n times, for a total number of $m \cdot (n + 1)$ unit test executions.[5] Our tool only re-executes one unit test for each failure. This improves performance by several orders of magnitude and allows Enforcer to scale up to large test suites. Moreover, 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.

3.5 Nested control structures

Enforcer treats nested exceptions by repeating the same unit test that covered the initial exception. Using an iterative approach, incomplete coverage inside an

[5] This includes the original test run where no faults are injected.

exception handler can be improved for exception handlers containing checked method calls inside nested `try/catch` statements [2].

However, an exception handler may contain nested control structures other than `try/catch`, such as an `if/else` statement, or more complex control structures. When encountering an `if/else` statement inside an exception handler, Enforcer works as follows: After the initial test run, a chosen unit test is repeated to cover the exception handler in question. This way, the `if` condition is evaluated to one of the two possible outcomes. Unfortunately, the other outcome cannot be tested by repeating the same test case, as a deterministic test always results in the same outcome of the predicate. Program steering cannot be used to force execution of the other half of the `if/else` block: A bit flip in the `if` condition would evaluate its predicate to a value that is inconsistent with the program state.[6]

Therefore, a different unit test would have to be chosen, one where the predicate evaluates to a different value. A priori, it cannot be guaranteed that such a unit test exists in the first place, as this would imply a solution to the reachability problem. A search for a solution would therefore have to be exhaustive. An exhaustive search executes all unit tests that could possibly trigger a given exception, hoping that one of the tests (by chance) covers the alternative outcome. This is undesirable, because it negates the performance advantage of Enforcer, which is based on the premise of only choosing one test per exception. Other possible solutions are:

- Manual specification of which unit test to choose for those special cases.
- Refactoring the exception handler in question so it can be tested manually.
- Finally, possible values for which a given `if` condition evaluates to a different value could be generated by perturbing an existing unit test case. Such a guided randomization would be similar to "concolic testing", which combines concrete and symbolic techniques to generate test cases based on a previous test run [18,30].

The problem of efficiently treating complex control structures inside exception handlers is therefore still open.

4 Usage of the Enforcer tool

The Enforcer tool is fully automated. The functionality of the tool is embedded into the application by code instrumentation. Code instrumentation can occur after compilation or at load time. After execution of the normal unit test suite, exception coverage is shown, and repeated tests are executed as necessary. To facilitate debugging, an additional feature exists to reduce the log file output that an application may generate. As experiments show, the Enforcer tool successfully finds faults in real applications.

[6] Because of this, the elegant approach to nested exceptions is defeated for the purpose of treating conventional control structures.

4.1 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. 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.

Alternatively, Enforcer can be invoked at load-time. In this mode, the new Java instrumentation agent mechanism is used [34]. 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 loaded at run-time.

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 Fig. 5).

4.2 Evaluation of results

Figure 5 shows a typical output of the Enforcer tool. 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. This output, resulting in a dot for each successful test, is not shown in the figure. 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.

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, while five belong to untested or dead code. 12 uncovered paths from an executed method call to their corresponding exception handler exist. Therefore, 12 tests

```
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: 20
*** Tests with uncovered catch blocks to execute: 12
............
Time: 0.301
OK (12 tests)
*** Total number of executed method calls: 32
*** Total number of executed catch blocks: 32
*** Tests with uncovered catch blocks to execute: 0
```

Fig. 5. Enforcer output when running the wrapped JUnit test suite.

are run again; the second run covers the remaining paths. Note that the second run may have covered additional checked method calls in nested `try/catch` blocks. This would have allowed increased coverage by launching another test run to cover nested exceptions [2].

4.3 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 [2]. 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 [2]. If no fault injection tool is used, then this output will never appear during testing and therefore does not constitute a problem in production usage.

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 the presence of many such stack traces 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.

4.4 Experiments

Table 1 shows the condensed results of experiments performed [2]. 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. (The reason why certain calls are not covered is because tests were not always fully deterministic, due to concurrency problems or unit tests not being idempotent.)

With previous tools [7,17], 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. Enforcer found 12 faults in the given example applications, using only existing unit tests and no prior knowledge of the applications [2].

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)	Faults found
Echomine	170	6.3	8.0	61	54	8 %	100 %	**9**
Informa	119	33.2	166.6	139	136	2 %	80 %	**2**
jConfig	97	2.3	4.7	169	162	3 %	61 %	**1**
jZonic-cache	16	0.4	0.7	8	6	25 %	100 %	0
SFUtils	11	76.3	81.6	6	2	67 %	100 %	0
SixBS	30	34.6	94.3	31	28	10 %	94 %	0
Slimdog	10	228.6	n/a[a]	15	14	7 %	n/a	**1**
STUN	14	0.06	0.7	0	0	0 %	0 %	0
XTC	294	28.8	35.5	112	112	0 %	92 %	0

[a] Repeated unit tests could not be carried out successfully because the unit test suite was not idempotent, and not thread-safe. Enforcer found one fault before the test suite had to be aborted due to a deadlock.

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 [28]. In traditional software testing, *coverage* metrics such as statement coverage [14,29] 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.

5.1 Static analysis and model checking

Static analysis investigates properties "at compile time", without executing the actual program. Non-deterministic decisions are explored exhaustively by verifying all possible outcomes. An over-approximation of all possible program behaviors is computed based on the abstract semantics of the program [10]. For analyzing whether resources allocated are deallocated correctly, there exist static analysis tools which consider each possible exception location [36].

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 [9] and device drivers [5]. Another project whose goal is very close to that of Enforcer verifies proper resource deallocation in Java programs [24]. It uses the Java PathFinder model checker [35] on an abstract version of the program, which only contains operations relevant to dealing with resources.

Static analysis (and model checking, if used on an abstract version of the program) 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. Execution of a concrete program in a model checker is possible, at least in theory [35]. However, model checking suffers from the state space explosion problem: The size of the state space is exponential in the size of the system. In addition to program abstraction, which removes certain low-level or local operations from a program, non-exhaustive model checking can be applied. By using heuristics during state space exploration, states where one suspects an error to be likely are given preference during the search [20]. The intention is to achieve a high probability of finding faults even if the entire state space is far larger than what can be covered by a model checker.

Such heuristics-based model checking, also called *directed model checking,* shows interesting similarities to our approach. By being able to store a copy of the program state before each non-deterministic decision, it can also selectively test for success or failure of an I/O call. Furthermore, model checking covers all interleavings of thread executions. The difference to fault injection is that model checking achieves such coverage of non-determinism on a more fine-grained level (for individual operations rather than unit tests) and includes non-determinism

induced by concurrency. Model checking does not have to re-execute code up to a point of decision, but can simply use a previously stored program state to explore an alternative. However, the overhead of the engine required to store and compare a full program state still makes model checkers much slower than normal execution environments such as a Java virtual machine [4]. Model checkers also lack optimization facilities such as just-in-time compilers, because the work required to implement such optimizations has so far always been beyond the available development capacity.

5.2 Fault injection

Even though fault injection cannot cover concurrency and has to re-execute a program up to a given state, it scales much better in practice than model checking. Fault injection in software is particularly useful for testing potential failures of library calls [21]. In this context, random fault injection as a black-box technique has shown to be useful on an application level [15]. Because our goal is to achieve a high test coverage, we target white-box testing techniques.

In software, two types of fault injection exist: low-level fault injection and high-level fault injection. Low-level fault injection targets mechanisms on an operating system level, such as timeouts and processor interrupts. Such fault injection tools simulate these mechanisms by leveraging corresponding functionality of the operating system or hardware [8]. Low-level tools are easier to deploy, but inherently more limited than high-level fault injection tools, which focus on the application level, where a failure of underlying system code typically manifests itself as an exception [7,16,17]. High-level tools are more powerful because they can modify the application code itself. In most cases, the higher-level semantics of injected faults makes failure analysis easier.

Java is a popular target for measuring and improving exception handling, as exception handling locations are well defined [19]. Our approach of measuring exception handler coverage corresponds to the *all-e-deacts* criterion [31]. 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 [17]. 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 [17], 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. Finally, our tool can dynamically discover the need for combined occurrences of failures when exception handling code should be reached. Such a dynamic analysis is comparable to another fault injection approach [7], but the aim of that project is totally different: It analyzes failure

dependencies, while our project targets code execution and improves coverage of exception handling code.

Similar code injection techniques are involved in program steering [23], which allows overriding the normal execution flow. However, such steering is usually very problematic 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 [23], or as schedule perturbation [13,32], which covers non-determinism in thread schedules. Our work is application-independent and targets non-determinism induced by library calls rather than thread scheduling.

6 Conclusions

Calls to system libraries may fail. Such failures are very difficult to test. Our work uses fault injection to achieve coverage of such failures. During initial test execution, coverage information is gathered. This information is used in a repeated test execution to execute previously untested exception handlers. The process is 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 exception handlers and uncovered several faults. Furthermore, our approach may even partially replace test case generation.

7 Future work

Improvements and extensions to the Enforcer tool are still being made. 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. An improvement w. r. t. execution time could be achieved by an analysis of test case execution time, in order to select the fastest test case for re-execution. Furthermore, an analysis of unit test dependencies could help to eliminate selection of problematic unit tests, which violate the JUnit contract specifying that a test should be self-contained and idempotent.

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 [1,19]. 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. It is also possible that such ideas can be expanded to coverage of control structures inside exception handlers, which is still an open problem.

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 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.
2. C. Artho, A. Biere, and S. Honiden. Enforcer – efficient failure injection. In *Proc. Int'l Conference on Formal Methods (FM 2006)*, Canada, 2006.
3. C. Artho, A. Biere, S. Honiden, V. Schuppan, P. Eugster, M. Baur, B. Zweimüller, and P. Farkas. Advanced unit testing – how to scale up a unit test framework. In *Proc. Workshop on Automation of Software Test (AST 2006)*, Shanghai, China, 2006.
4. C. Artho, V. Schuppan, A. Biere, P. Eugster, M. Baur, and B. Zweimüller. JNuke: Efficient Dynamic Analysis for Java. In *Proc. 16th Int'l Conf. on Computer Aided Verification (CAV 2004)*, volume 3114 of *LNCS*, pages 462–465, Boston, USA, 2004. Springer.
5. T. Ball, A. Podelski, and S. Rajamani. Boolean and Cartesian Abstractions for Model Checking C Programs. In *Proc. 7th Int'l 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.
6. M. Bushnell and V. Agrawal. *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits*. Kluwer, 2000.
7. 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.
8. J. Carreira, H. Madeira, and J. Gabriel Silva. Xception: A technique for the experimental evaluation of dependability in modern computers. *Softw. Engineering*, 24(2):125–136, 1998.
9. 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.
10. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th ACM Symposium on Principles of Programming Languages (POPL 1977)*, pages 238–252, Los Angeles, USA, 1977. ACM Press.
11. DBUnit, 2007. `http://www.dbunit.org/`.

12. D. Engler and M. Musuvathi. Static analysis versus software model checking for bug finding. In *Proc. 5th Int'l Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI 2004)*, volume 2937 of *LNCS*, pages 191–210, Venice, Italy, 2004. Springer.
13. E. Farchi, Y. Nir, and S. Ur. Concurrent bug patterns and how to test them. In *Proc. 20th IEEE Int'l Parallel & Distributed Processing Symposium (IPDPS 2003)*, page 286, Nice, France, 2003. IEEE Computer Society Press.
14. N. Fenton and S. Pfleeger. *Software metrics (2nd Ed.): a rigorous and practical approach*. PWS Publishing Co., Boston, USA, 1997.
15. 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.
16. 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 Int'l Conf. on Dependable Systems and Networks (DSN 2003)*, pages 595–604, San Francisco, USA, 2003.
17. C. Fu, B. Ryder, A. Milanova, and D. Wonnacott. Testing of Java web services for robustness. In *Proc. ACM/SIGSOFT Int'l Symposium on Software Testing and Analysis (ISSTA 2004)*, pages 23–34, Boston, USA, 2004.
18. P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proc. ACM Int'l Conf. on Programming Language Design and Implementation (PLDI 2005)*, pages 213–223, Chicago, USA, 2005.
19. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley, 2005.
20. A. Groce and W. Visser. Heuristics for model checking java programs. *Int'l Journal on Software Tools for Technology Transfer (STTT)*, 6(4):260–276, 2004.
21. M. Hsueh, T. Tsai, and R. Iyer. Fault injection techniques and tools. *IEEE Computer*, 30(4):75–82, 1997.
22. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. *LNCS*, 2072:327–355, 2001.
23. M. Kim, I. Lee, U. Sammapun, J. Shin, and O. Sokolsky. Monitoring, checking, and steering of real-time systems. In *Proc. 2nd Int'l Workshop on Run-time Verification (RV 2002)*, volume 70 of *ENTCS*. Elsevier, 2002.
24. X. Li, H. Hoover, and P. Rudnicki. Towards automatic exception safety verification. In *Proc. 14th Int'l Symposium on Formal Methods (FM 2006)*, volume 4085 of *LNCS*, pages 396–411, Hamilton, Canada, 2006. Springer.
25. J. Link and P. Fröhlich. *Unit Testing in Java: How Tests Drive the Code*. Morgan Kaufmann Publishers, Inc., 2003.
26. B. Meyer. *Eiffel: the language*. Prentice-Hall, Inc., Upper Saddle River, USA, 1992.
27. Microsoft Corporation. *Microsoft Visual C# .NET Language Reference*. Microsoft Press, Redmond, USA, 2002.
28. G. Myers. *Art of Software Testing*. John Wiley & Sons, Inc., 1979.
29. D. Peled. *Software Reliability Methods*. Springer, 2001.
30. K. Sen and G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *18th Int'l Conf. on Computer Aided Verification (CAV 2006)*, volume 4144 of *LNCS*, pages 419–423, Seattle, USA, 2006. Springer. (Tool Paper).
31. 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.

32. S. Stoller. Testing concurrent Java programs using randomized scheduling. In *Proc. 2nd Int'l Workshop on Run-time Verification (RV 2002)*, volume 70(4) of *ENTCS*, pages 143–158, Copenhagen, Denmark, 2002. Elsevier.
33. B. Stroustrup. *The C++ Programming Language, Third Edition*. Addison-Wesley Longman Publishing Co., Inc., Boston, USA, 1997.
34. Sun Microsystems, Santa Clara, USA. *Java 2 Platform Standard Edition (J2SE) 1.5*, 2004. `http://java.sun.com/j2se/1.5.0/`.
35. W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2):203–232, 2003.
36. 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.
37. A. White. SERP, an Open Source framework for manipulating Java bytecode, 2002. `http://serp.sourceforge.net/`.