Testing and Understanding Error Recovery Code in Java Applications

Chen Fu and Barbara G. Ryder

Division of Computer and Information Sciences
Rutgers, the State University of New Jersey
110 Frelinghuysen Road, Piscataway, NJ 08854-8019, USA
{chenfu, ryder}@cs.rutgers.edu

Abstract. Server applications are expected to handle lower level faults and keep them from bringing down the whole system. Java provides a program-level exception handling mechanism in response to error conditions (that are translated into exceptions by Java VM). However, exception handling code is often widely scattered throughout an application and untested. This paper presents a program visualization tool ExTest that shows precisely all the handlers for exceptions triggered by certain kinds of operations, and for each of these handlers all the witness paths of how the operation would be triggered. Thus ExTest helps programmers understand the exception handling behavior of Java programs and also facilitates testing exception handling code.

1 Introduction

The Java programming language provides a program level exception handling mechanism in response to error conditions that happen during program execution. Subsystem faults (e.g. disk failure, network congestion) are translated into exceptions (e.g. java.io.IOException, java.net.SocketException) by the Java Virtual Machine [1]. Proper handling of these exceptions in program code is extremely important for reliability and fault-tolerance in server applications built in Java.

An exception handling mechanism helps separate exception handling code from code that implements functionalities during normal execution. However, exception handling code that deals with certain kinds of faults is still widely scattered over the whole program and mixed with other exception handling code, or even irrelevant code, making it hard to understand the behavior of the program under certain system fault conditions.

Moreover, exception handling blocks, especially those corresponding to system faults, are often left untested, for the reason that they can not be triggered by just tuning input data of the program. In our previous work [2], we proposed a white box testing metric for the exception handling behavior of the program. Supporting program analysis algorithms together with a testing framework using fault injection [3] were presented. Although very precise analysis is used in identifying exception def-uses, false positives can not be fully eliminated. It is a tedious and difficult job to identify the real false positives when they can not be exercised during the test.

In this paper we present ExTest a program visualization tool built on top of Eclipse[4]. Based on program analysis introduced in [2], it groups together handlers
that handle exceptions triggered by a set of fault-sensitive operations\(^1\). Thus it facilitates navigation of the program code that relates to exceptions triggered by certain operations of interest. It also shows all program paths via which these operations can be reached from the call site in a try block, helping a user to understand the exception handling structure and to identify spurious exception def-uses.

The rest of this paper is organized as following: In Sec. 2 we give a brief overview of exception def-use analysis, which was introduced in [2]. Section 3 shows the structure and the functionality of ExTest in detail. In Sec. 4 we discuss the related work in the area of understanding and injecting exception handling code in Java programs. Our future research work is discussed in Sec. 5.

2 Background

In [2], we proposed a def-use testing methodology for exception handling code, which is analogous to the all-uses logic of traditional def-use testing [5]. We repeat some of the key concepts here: In any given program execution, each fault-sensitive operation may produce an exception that reaches some subset of the program’s catch blocks. We can treat these fault-sensitive operations as the definition points of exceptions, and catch blocks as the exception definitions. We define exception-catch (e-c) links:

**Definition** An e-c link \(e \rightarrow c\) in a program, we say there is an e-c link \(e \rightarrow c\) if a statement or a native method that may be affected by some fault \(e\) is actually transferred control to by throwing an exception \(c\) from a given e-c link, if a set \(P\) of fault-sensitive operations may produce exceptions \(c\) in a set \(C\) of catch blocks in a program, we say there is an e-c link \(e \rightarrow c\) if could possibly trigger \(e\) and \(c\) could possibly trigger \(e\).

Currently in the experiments conducted, we select to contain all the native methods in JDK library that do network or disk I/O. In the rest of this paper we make this assumption unless explicitly stated otherwise\(^2\).

Figure 1 shows the organization of our exception def-use testing system, which comprises two kinds of program analysis: static (compile time) and dynamic (run time) analysis. The static analysis calculates the possible e-c links for a program, which will be introduced shortly. The dynamic analysis monitors program execution, calls for fault injection to trigger an exception at an appropriate time, and records test coverage: The compiler uses the set of e-c links to identify where to place instrumentation that will communicate with the fault injection engine during execution. This communication will be in the form of a fault injection of a particular fault when execution reaches the try-catch block of an e-c link. The injected fault will cause an exception to be thrown upon execution of a fault-sensitive operation of the e-c link. The compiler also instruments the program to record the execution of the corresponding catch block. The tester runs the program and gathers the observed e-c links from each run. The testing goal is to drive the program into different part of the code so that fault injection can help exercise all the e-c links found in the program. Finally, the test harness calculates the overall coverage information for this test suite: the observed e-c links vs. the possible e-c links.

\(^1\) Either a throw statement or a native method that may be affected by some fault a hardware or OS failure and produce some exception.  
\(^2\) The analysis and the testing framework are not dependent on the \(P\) selected.
Next we introduce the static analysis that finds the possible e-c links in a Java program. Two algorithms are developed: **Exception-ow** and **DataReach** [2].

**Exception-ow** is a data ow analysis, similar to Reaching Definitions [6], but runs on call graphs instead of control ow graphs. Each path is propagated along call edges in the reverse direction until some try-catch block is hit. The path is traced back to the call site and catches the exception thrown by ; thus an e-c link is met that encloses the method where the exception comes from. The exception propagation path can be collected on demand after the analysis finishes.

It is obvious that using a more precise analysis for call graph construction such as points-to analysis [7, 8] helps to reduce the number of infeasible e-c links found by exception-ow analysis. However, in practice even a very precise call graph building algorithm introduces many infeasible e-c links. Figure 2 is an example of typical uses of the Java I/O packages. Figure 3 illustrates how infeasible e-c links are introduced even given a fairly precise call graph. As we can see, the catch in read file only handles exceptions result from disk failure and the catch in readNet only handles those triggered by network faults. But exception-ow information is merged in BufferedInputStream.read1() and propagated to both readFile and readNet.

We developed **DataReach** analysis to reduce the number of infeasible e-c links produced. The intuition was to use data reachability, usually obtained using points-to analysis, to constrain control-ow reachability. For example, continuing with Fig. 2, we can prove SocketInputStream.read() is not reachable from the call site fsr.read() in method readFile, by showing that during the lifetime of the call fsr.read(), no object of type SocketInputStream may be either loaded from any static/instance field of some class/object, or created by a new statement. In this way we can show that all the control-ow paths associated with this e-c link are not
void readFile(String s) {
    byte[] buffer = new byte[256];
    try {
        InputStream f = new FileInputStream(s);
        InputStream fsrc = new BufferedInputStream(f);
        for (...) 
          c = fsrc.read(buffer);
    } catch (IOException e) { ... }
}
void readNet(Socket s) {
    byte[] buffer = new byte[256];
    try {
        InputStream n = s.getInputStream();
        InputStream ssrcc = new BufferedInputStream(n);
        for (...) 
          c = ssrcc.read(buffer);
    } catch (IOException e) { ... }
}

Fig. 2. Code Example for Java I/O Usage

3 Visualization Tool ExTest

In addition to being used by the exception def-use testing system, the information produced by these analysis, if carefully organized and visually displayed in an integrated development environment (IDE), can greatly facilitate both testing and program understanding of the exception handling code. We developed an Eclipse plug-in ExTest, which invokes these analysis and organizes the output data into tree views to let users browse e-c links and trace exception propagation paths.
3.1 Motivation

When studying the exception handling behavior of Java programs, we found that, as mentioned in Sec. 1, exception handlers dealing with certain kinds of faults are often scattered in the program and mixed with handlers that take care of other kinds of error conditions.

```java
void readString(String s){
    String buffer = s;
    try{
        InputStream n = new StringBufferInputStream(s);
        InputStream in = new BufferedInputStream(n);
        for (...)
            c = in.read(buffer);
    }catch (IOException e){ ... }
}
```

Fig. 4. Unreachable Catch Block

For instance, a catch clause that handles an I/O exception may appear at each program point where some I/O channel is active. Each of these catch clauses may handle I/O exceptions triggered by different fault-sensitive operations (e.g. file write or socket creation), as shown in Fig. 2. Worse, some of these catch clauses never handle any I/O exception: Suppose in the program containing the code in Fig. 2, there is another method `readString`, shown in Fig. 4. The catch block in this method handling I/O exception will never be triggered, because the code in the corresponding try block only reads from a string buffer in the memory — no actual I/O operation involved. Yet the try-catch structure is necessary for the program to compile.

If a programmer wants to learn this program's behavior under disk failure, she needs to find all the catch clauses that may handle exceptions that result from disk faults. Suppose a powerful lexical search tool with Java language knowledge as well as program specific information (e.g. type) is available. Then she can easily locate all the catch clauses that handle `IOException` or more general types of exceptions, but she still has to manually inspect at least all the try-catch blocks in both Fig. 2 and Fig. 4, and sometimes code reachable from them too, instead of the one in method `readFile` that actually handles the exception result from disk failure. The problem becomes much more severe in non-toy Java server applications.

Using the analysis mentioned in Sec. 2, we can compute all the potential `e-c links` of the program. Each `e-c link` tells us the fault-sensitive operation that triggers the exception and where it is handled. Thus, we can help solving the above problem by grouping `e-c links` according to their value. Since the number of fault-sensitive operations that relate to disk I/O is small, one can just browse `e-c links` starting with

---

3 This program □ a single Java class containing the `main` method calling all of these three methods (also defined in this class) in Fig. 2 and 4 □ is used as a running example in the following discussion about `ExTest`. 
these operations to get a good estimate of all the try-catch blocks that are related to disk I/O.

Ours is a program analysis which computes a safe approximation of program behavior. False positives are unavoidable, which means for some of the e-c links ( ), the exception thrown at never reaches . It is up to the human programmer to decide whether an e-c link is actually spurious. This is especially important for exception def-use testing, because spurious e-c links can never be exercised during any test. Our program analysis provides exception propagation path data for all e-c links. Displaying these paths visually in Eclipse IDE helps to identify the spurious ones. In order for identifying spurious e-c links, all propagation paths for a given e-c link must be shown.

3.2 Tool Structure

Our program analysis are implemented as modules in the Soot Java Analysis and Transformation Framework [8] version 2.0.1. Upon user request, ExTest (i) calculates environments of the current working program (e.g. class paths) using Eclipse services, (ii) starts another process running Soot with our modules enabled, and (iii) reads the output data of the Soot modules after the process finishes.

In the Eclipse IDE, we want the users to be able to explore the e-c links (e.g. by grouping all the catch clauses and their relationships with the fault-sensitive operations) as well as the witness paths that demonstrate the feasibility of an e-c link. The data generated by the Soot modules are organized in an XML file, which contains all the e-c links found in the given program and information about the paths needed by ExTest to perform the intended functionality.

3.3 Browsing e-c links

Each record of an e-c link in the output data of our Soot modules contains the following information: the ID of , the position of in source code and the call site(s) in the corresponding try block which may lead to the execution of . These e-c links can be grouped in two ways: by or by . We implemented both of them by means of two different views in Eclipse: the Handlers view and the Triggers view.

Figure 5(a) shows the Handlers view: a tree-view in Eclipse where e-c links are grouped by the try-catch blocks. These try-catch blocks are further grouped by their definition positions: the methods, classes, packages in which they are defined. Each try-catch block can be expanded to show all the fault-sensitive operations that may trigger exceptions reaching the catch. The last try-catch block in the figure is highlighted and expanded. It is defined in package iotest.mixed, class Mixed and method readFile. We can see that one method call in the try block reaches a fault-sensitive operation in the JDK: File Read.

Figure 5(b) shows the Triggers view, where the e-c links are grouped by the fault-sensitive operations. By expanding the File Read operation we can see that only one try-catch block in the program handles an exception thrown by read of a file. So if a user is interested in program behavior under a disk fault, she can just concentrate on this one catch block.
Thanks to the environment provided by Eclipse IDE, these two tree views can be interactively explored. The try-catch block, the statements in the block that may lead to the fault-sensitive operation, etc., can be opened and highlighted in the Java source files, upon double-clicking on the corresponding items in the view. For example, in both views, we can see the actual code for the try-catch block #0 by double-clicking on the line.

3.4 Displaying All Paths for an e-c link

We also want to display all paths that show how p in an e-c link (\textsection 2.4) can be reached from the try block that corresponds to it. Selecting and displaying only the (the shortest) path for each e-c link is not enough, especially with the presence of the false positives. In order for a programmer to decide that an e-c link is spurious, she has to make sure that all the control flow paths from the corresponding try to are actually infeasible. So it is necessary to display all these paths to be practically useful. But the total number of such paths can be exponential to the size of the program \cite{9}. Clearly, the approach of gathering and dumping all these paths into an output file after the analysis finishes will not scale.

As mentioned in Sec. 2, Exception-flow analysis records the propagation paths of each \textsection 2.4 by annotating nodes on the call graph. These annotations can be trivially changed into annotations on call edges. Since the set of fault-sensitive operations is pre-selected according to the kind of faults that are of interest to the user (not depending
on the program being analyzed), the size of the annotated call graph is at most linear in the size of the original call graph. Figure 6(a) shows the annotated call graph for the example in Figure 4. Edges in the call graph are annotated with IDs of the fault-sensitive operations according to the result of Exception-ow analysis.

The problem with this approach is that the results of DataReach are ignored. As stated before, Exception-ow analysis alone would leave too many false positives in the graph; with this approach, the user must manually explore many unnecessary call edges to decide that some fault-sensitive operations are infeasible. So we need to take the advantage of DataReach to reduce this workload.

Recall that DataReach proves that some of the e-c links are infeasible by showing the infeasibility of all the control-flow paths of these e-c links. To be able to incorporate its results into the annotated call graph, we modified DataReach so that for each e-c link (i.e., an annotation on all the call edges associated with a certain method) that are confirmed only if it survives DataReach test, the DataReach annotations are written with the graph. With this approach, many spurious annotations can be removed. Figure 6(b) shows the annotated call graph with the unconfirmed annotations removed.

With the annotated call graph, the paths can be generated on demand in ExTest. Suppose one of the users chooses to trace the paths of some e-c link . ExTest can retrieve the outgoing edges departing from the try block that are annotated with , and the target methods can be displayed to the user. Then the user can choose to trace one of the methods, and ExTest can retrieve all the outgoing edges from that method that are annotated with . This process can be repeated until the fault-sensitive operation itself is reached.
Figure 7 is the expanded view of the last e-c link shown in Fig. 5(b). Only one witness path was discovered by the analysis, which precisely reflects the analysis result shown in Fig. 6.

However, we are not always so lucky in bigger programs; paths in these programs can get very complicated, especially inside the JDK library classes that make heavy use of polymorphism. Figure 8 shows part of the Triggers view displayed when browsing e-c links in one of the testing benchmarks used in [2] — a FTP server written in Java [10]. Witness paths of one e-c link are partly expanded in the figure, with the fault-sensitive operation `SocketInputStream.read()` highlighted.

As can be seen from the figure, the fan out of some of the nodes along the paths is large (e.g., `InputStreamReader.close()`). Furthermore, many of the methods appear more than once, which indicates the possibility of recursion introducing a path with unbounded length. Since these paths are extracted out of a call graph, expanding the second appearance of a method on a path brings exactly the same set of children in the tree view. This is wasteful and introduces unnecessary complexity into the view. Manually identifying the recursion in a complex view like this is not trivial. Therefore we have automated recursion detection in ExTest. As shown in Fig. 8, many methods are annotated with ... and they are not expandable, which shows that the method has been called recursively and further expansion is not necessary.

If we only show only one witness path of the e-c link the natural selection would be to show the shortest one — the view can be greatly simplified, but the real complexity of the problem would be hidden from the user. With only one path shown, the user is not helped in identifying infeasible paths; however, expanding and highlighting the shortest path automatically among all paths may help in understanding the overall program structure quickly. We are now working on implementing this feature in ExTest.

4 Related Work

This paper presents a tool to help understand and maintain the exception handling feature of Java programs, based on exception-catch link analysis. There is much previous research work in both exception handling analysis and its usage in various software engineering tools. Here we will discuss only the works that are most closely related to the tool discussed in this paper.

There are tools built to improve exception handling in programs, for example avoiding exception handling through subsumption, or handling unhandled exceptions for a given method. Jo et. al [11] presented an interprocedural set-based [12] exception o-w analysis for checked exceptions. A tool [13] was built based on this analysis which shows, for a selected method, uncaught exceptions and their propagation paths. It is unclear from the paper whether a certain path for each exception is selected and displayed, or if all of the paths are displayed. Experiments show that this is more accurate than an intraprocedural JDK-style analysis on a set of benchmarks of which contain more than 1000 methods. Robillard et. al [14] described a data o-w analysis that propagates both checked and unchecked exception types interprocedurally. Their tool Jex illustrate

---

4 More discussion about research results in fault injection, exception analysis and optimizations, infeasible paths detection and program testing can be found in the related work section of [2].
Fig. 8. Exception Propagation Path
exception handling structure in application code. It analyzes exception control ow and thus identi es exception subsumption. These analysis mentioned above are less precise than ours. Their call graph is constructed using class hierarchy analysis, which yields very imprecise results [15, 16]. Even if a fairly precise call graph were provided for their analysis, the precision of the results would be resemble those of our Exception-ow analysis using the same call graph. So they are not capable of identifying that the catch clause in Fig. 4 can never be triggered.

Reimer and Srinivasan [17] introduced SABER, part of which targets at a wide range of exception usage issues in order to improve exception handling code in large J2EE applications and ultimately to increase robustness of the program. These issues include swallowed exceptions, single catch for multiple exceptions, a handler too far away from the source of the exception and costly handlers. Warnings are given to the programmer upon recognizing one of these problems. Unfortunately, analysis used to identify these potentially problematic code areas are not introduced. Their work, if combined with our analysis, might prune some of the warning messages to reduce the workload of user, and produce more valuable information at the same time. For instance, suppose the catch clause in Fig. 4 is empty, since our analysis shows that it can never be triggered, warnings relative to this catch clause can be suppressed. Moreover, when facing a try block with many call sites that potentially throw the same type of exception, our analysis can show whether these exceptions are of the same origin and with similar propagation paths, which in turn helps deciding whether the try block needs to be split.

5 Conclusions and Future Work

We present a program visualization tool that facilitates navigating code related to the exception handling feature of Java programs, based on the program analysis used in exception def-use analysis [2]. We want to reveal all information needed to the user, while carefully organizing the data to help human browsing and reasoning.

Despite of our current efforts, Fig. 8 shows us that exploring program code based on conservative static program analysis results can be difficult. One way to alleviate the situation is to use more precise analysis (but possibly more expensive) to reduce the size of the result data.

The current implementation of ExTest only displays the results of the static analysis in [2]. Our next step is to display visually both the static and dynamic analysis results, because we believe that it will give user more intuition and hints when browsing. We are working on combining two kinds of dynamic information into the current views: testing coverage data of e-c links and dynamic call graphs/trees. Once we incorporate these data, we can highlight e-c links not covered during the test in the e-c link views (shown in Fig. 5) to draw the user’s attention to these untested parts of the program. Furthermore, when a user chooses to explore the paths of a certain e-c link, with the dynamic call graph/tree, we can graphically show which of the edges are actually executed during the test. On this view, the users might want to concentrate on the □ fringe□

5 For example, a call graph constructed using Spark, a eld sensitive context insensitive points-to analysis module in Soot.
of the already executed edges, to see why the program is taking the wrong branch. Thus Extest can help the user either to compose another test case or decide that edge is actually infeasible.

References