Navigating Error Recovery Code in Java Applications*

Chen Fu Barbara G. Ryder
Department of Computer Science, Rutgers University
Piscataway, NJ 08854, USA
{chenfu,ryder}@cs.rutgers.edu

ABSTRACT

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 quite 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`) by the Java Virtual Machine[10]. Proper handling of these exceptions in program code is 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, because they can not be triggered by just tuning input data of the program. In our previous work[5], we proposed a white box testing metric for the exception handling behavior of the program. The underlying program analysis together with a testing framework using fault injection[11] were presented and empirically evaluated. Although very precise analysis is used in identifying exceptions and their handlers (i.e. 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[7]. Based on the program analysis introduced in [5], it groups together handlers that handle exceptions triggered by a set of fault-sensitive operations[1]. Thus *ExTest* 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 some call site in the `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 that was introduced in [5]. Sec. 3 shows the structure and the functionality of *ExTest*. In Sec. 4 we discuss the related work in the area of understanding and improving exception handling code in Java programs. Future research work is discussed in Sec. 5.

2. BACKGROUND

In [5], we proposed a def-use testing methodology for exception handling code, which is analogous to the *all-uses* metric of traditional def-use testing [12]. We repeat some of the key concepts here: In a Java program, each fault-sensitive operation may produce an exception that reaches some subset of the program's `catch` blocks. We define *exception-catch (e-c) links*:

Definition ((*e-c link*):): Given a set P of fault-sensitive operations that may produce exceptions, and a set C of `catch` blocks in a program, we say there is an *e-c link*(p, c)[5] between $p \in P$ and $c \in C$ if p may trigger c; we say that a given *e-c link* is **experienced** in a set of test runs T, if p actually transfers control to c by throwing an exception during a test in T.

For an *e-c link*(p, c), p can be seen as a *def* of an exception object and c as the corresponding *use*. In the experiments conducted, we selected P 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. Note that the testing framework is not dependent on the P selected. Currently only checked exceptions are considered in the system.

Figure 1 shows the organization of our exception def-use testing system. The static (compile-time) analysis, which calculates the possible *e-c links* for a program, will be introduced shortly. The dynamic (run-time) analysis monitors program execution, calls for the fault injector to trigger an exception at an appropriate time, and records test coverage. The compiler uses the set of *e-c links* to iden-

*This work was supported in part by IBM Eclipse Innovation Grant

[1]A fault-sensitive operation is a `throw` statement, or a native method that may be affected by some fault – a hardware or OS failure – and produce an exception.

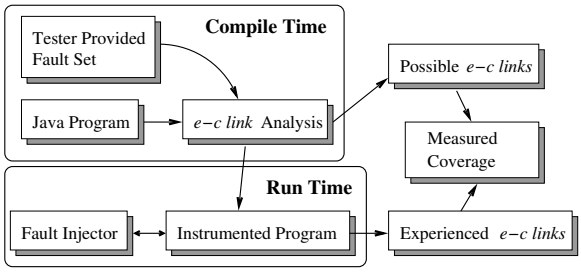


Figure 1: Exception def-use Testing Framework

tify where to place instrumentation that will communicate with the fault injection engine during execution. This communication will request the 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 the fault-sensitive operation of the *e-c link*. The compiler also instruments the code to record the execution of the corresponding `catch` block. The tester runs the program and gathers the *experienced e-c links* from each run. The testing goal is to drive the program into different parts 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: *experienced e-c links* vs. possible *e-c links*.

Next we introduce the static analysis that finds the possible *e-c links* in a Java program. Two algorithms are developed: ***Exception-flow*** and ***DataReach*** analysis [5].

Exception-flow is a dataflow analysis defined on the program call graph. Each $p \in P$ is propagated along the call edges in the reverse direction until some `try-catch` block c is met that encloses the call site and catches the exception thrown by p; thus an *e-c link* (p, c) is recorded.

```
void readFile(FileInputStream f){
  byte[] buffer = new byte[256];
  try{
    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 ssrc=new BufferedInputStream(n);
    for (...)
      c = ssrc.read(buffer);
  }catch (IOException e){ ...}
}
```

Figure 2: Code Example for Java I/O Usage

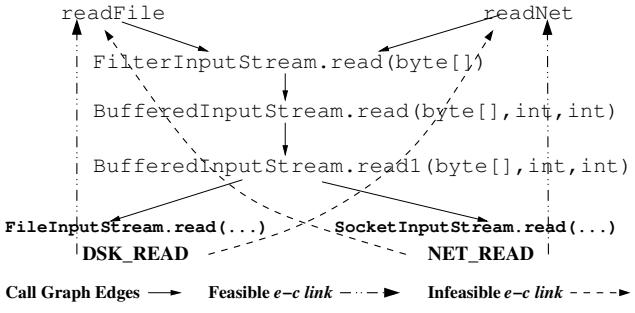


Figure 3: Call Graph for Java I/O Usage

It is obvious that the precision of *Exception-flow* analysis is affected by the precision of the call graph. But 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 the result of *Exception-flow* analysis based on a fairly precise call graph of code in Fig.2: both fault-sensitive operations DSK_READ and NET_READ can be propagated to the `try` blocks in `readFile` and `readNet`, resulting in 4 *e-c links*. But by reading the code we can see that (DSK_READ, `catch` in `readFile`) and (NET_READ, `catch` in `readNet`) are infeasible.

We developed ***DataReach*** analysis to reduce the number of infeasible *e-c links* produced. The intuition was to use data reachability obtained using points-to analysis, to confirm control-flow reachability. For example, continuing with Fig. 2, we can prove `SocketInputStream.read()` is **not** reachable from the call site `fsrc.read()` in method `readFile`, by showing that during the lifetime of the call `fsrc.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-flow paths associated with this *e-c link* are not feasible, so that the infeasibility of the *e-c link* from `SocketInputStream.read()` to the `catch` block in `readFile` is proved. In general, *DataReach* tries to prove the infeasibility of each *e-c link* output by *Exception-flow* analysis, and only outputs those that it cannot prove to be infeasible. Our experiments showed that *DataReach* improved the precision of the system significantly; it reduced the number of possible *e-c links* by 41% on average (compared to the most precise analysis without DataReach) in 6 benchmarks used in [5].

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 for this purpose.

Motivation

During the study, we found that exception handlers that deal with certain kinds of faults are often scattered in the program and mixed with handlers that handle other kinds of error conditions. 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. DSK_READ or NET_READ). 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[2]. The `catch` block in this method handling I/O exception will **never** be triggered, because the code in the `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 has 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. types) is available. Then she can eas-

[2]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 the example in the following discussions.

```
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){ ...}
}
```

Figure 4: Unreachable `catch` Block

ily locate all the `catch` clauses that handle `IOException` or more general types of exceptions, but she still has to manually inspect at least all three `try-catch` blocks in both Fig. 2 and Fig. 4, instead of just the one in method `readFile` that actually handles the exception result from disk failure. The problem becomes much more severe in real 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* (p, c) 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 p 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 these operations to 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* (p, c), the exception thrown at p never reaches c. 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.

Tool Structure

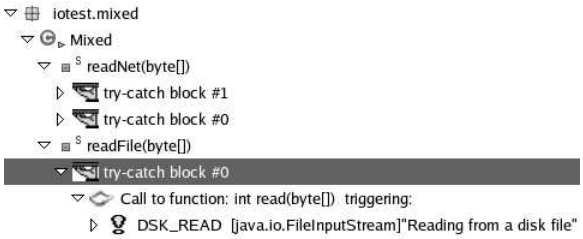
Our program analysis is implemented as modules in the Soot Java Analysis and Transformation Framework [9] version 2.0.1. Upon user request, *ExTest* starts another process running Soot with our modules enabled, and reads the output data of the Soot modules after the process finishes.

In the Eclipse IDE, we want the users to be able to explore the *e-c links* (e.g. browsing 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.

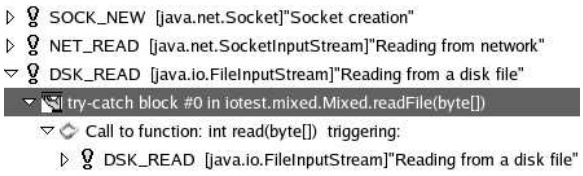
Browsing *e-c links*

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

Figure 5(a) shows the *Handlers* view, 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: "DSK_READ".



(a) *Handlers* View



(b) *Triggers* View

Figure 5: Tree Views of *e-c links*

Figure 5(b) shows the *Triggers* view, where the *e-c links* are grouped by the fault-sensitive operations. By expanding the "DSK_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 `try` block that may lead to the fault-sensitive operation, etc., can be opened and highlighted in the Java source file editors, upon double click 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.

Displaying All Paths for an *e-c link*

We also want to display the paths that show how p in an *e-c link* (p, c) can be reached from the `try` block that corresponds to c. Selecting and displaying only one (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 p are actually infeasible. So it is necessary for *ExTest* to display **all** these paths to be practically useful. But the total number of paths may be exponential to the size of the program [2]! Clearly, the approach of gathering and dumping all these paths into an output file after the analysis finishes will not scale.

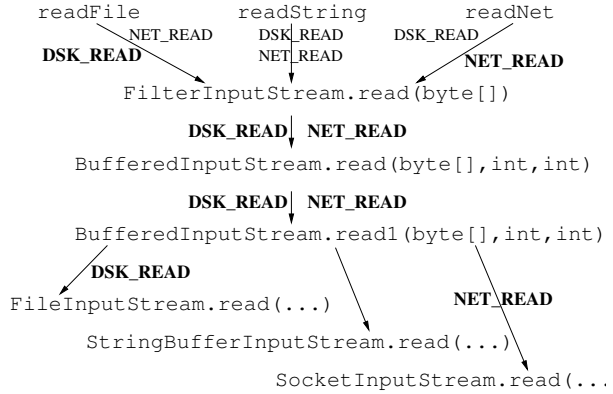


Figure 6: Annotated Call Graph

Exception-flow analysis can record the propagation paths of each $p \in P$ by annotating call edges in the call graph. Figure 6 shows the annotated call graph for the code in Figs. 2 & 4. Since the set of fault-sensitive operations P is pre-selected according to the

fault set provided by 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.

Recall that *DataReach* proves that some of the *e-c links* are infeasible by showing the infeasibility of the 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* (p, c), the annotations of p on all the call edges associated with (p, c) are *confirmed* only if we *cannot* prove the infeasibility of (p, c). During the output of the call graph, only the *confirmed* annotations are written with the graph. In Fig. 6 confirmed annotations are shown in bold face.

With the annotated call graph, the paths can be generated on demand in *ExTest*. Suppose one user chooses to trace the paths of some *e-c link* (p, c). *ExTest* can retrieve from the graph all the outgoing edges departing from the `try` block that are annotated with p, and the target methods can be displayed to the user. Then the user can choose to trace one of these methods, *ExTest* can retrieve all the outgoing edges from that method that are annotated with p and display the target methods of these edges. This process can be repeated until the fault-sensitive operation p itself is reached.

Figure 7: Exception Propagation Path

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[3] shows part of the *Triggers* view displayed when browsing *e-c links* in one of the testing benchmarks used in [5] – a FTP server written in Java [18]. 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 the shortest one – the view can be greatly simplified, but the real complexity of the problem would be hidden from the user: the user is not helped in identifying infeasible

[3]JDK 1.3.1_08 is used in the figure.

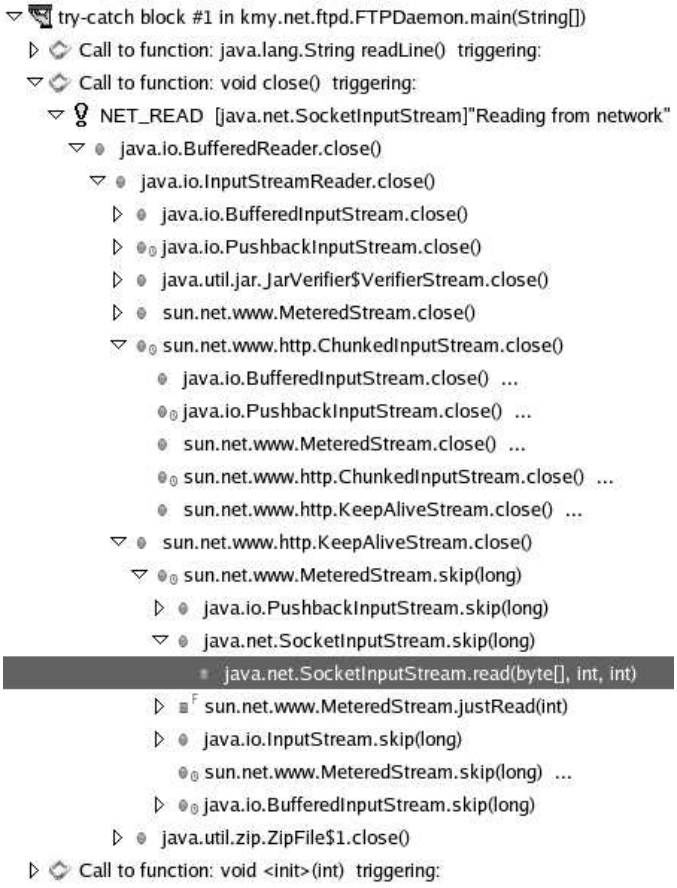


Figure 8: Exception Propagation Path

paths; however, expanding and highlighting the shortest path automatically among all paths may help in understanding the overall program structure. 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[4].

There are tools built to improve exception handling in programs, for example avoiding exception handling through subsumption, or finding unhandled exceptions for a given method. Jo et. al [8] presented an interprocedural set-based [6] exception-flow analysis for checked exceptions. A tool [3] 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 five of which contain more than 1000 methods. Robillard et. al [14] described a dataflow analysis that propagates both checked and unchecked exception types interprocedurally. Their tool *Jex* illustrates the exception handling structure in application code. It

[4]More extensive discussion of related research results can be found in the related work section of [5].

analyzes exception control-flow and thus identifies exception subsumption. These analysis are less precise than ours. Their call graph is constructed using Class Hierarchy Analysis (CHA), which yields very imprecise results [4, 1]. Even if a fairly precise call graph[5] were provided for their analysis, the precision of the results would be resemble those of *Exception-flow* 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 [13] 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. 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, the underlying analysis is not discussed.

Sinha and colleagues [17] proposed a tool (i) to visualize exception anomalies similar to those defined in [13] by using the static analysis of [16], and (ii) to display exception-based test coverage requirements related to those in [15]. The static analysis used for call graph building for both of these tasks is based on CHA. Our experiments in [5] on testing interprocedural exception handling in moderately large benchmarks (e.g., 2080 methods, 278 classes) showed that more than 97% of the e-c links found using CHA were false positives. Thus, the analysis in [17] has been shown to be too imprecise for practical use on real programs. In addition, it is not clear how exceptions thrown within the Java JDK libraries are accounted for in [17]; the empirical data reported for checked exceptions shows their usage is sparse and does not seem to include exceptions thrown by the Java libraries and caught by the application. These factors raise serious questions about the practicality and scalability of the analysis in [17] and thus, the utility of the proposed tool.

5. CONCLUSIONS AND FUTURE WORK

We present a tool that facilitates navigating code related to the exception handling feature of Java programs, based on exception def-use analysis [5]. We want to reveal all information needed to the user, while carefully organizing the data to help browsing and reasoning.

Despite of our current efforts, exploring program code based on conservative static program analysis results can be difficult (see Fig. 8). 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 [5]. Our next step is to add dynamic analysis results in to the views: testing coverage data of *e-c links* and dynamic call graphs/trees. With 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" of the already executed edges, to see why the program is taking the wrong "branch". In this manner *Extest* will help the user either to compose another test case or decide that edge is actually infeasible.

6. REFERENCES

[5]For example, a call graph constructed using *Spark*, a field-sensitive context-insensitive points-to analysis module in Soot.

[1] D. Bacon and P. Sweeney. Fast static analysis of C++ virtual functions calls. In *Proceedings of ACM SIGPLAN Conference on Object-oriented Programing Systems, Languages and Applications (OOPSLA'96)*, Oct. 1996.

[2] T. Ball and J. R. Larus. Efficient path profiling. In *MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 46–57, Washington, DC, USA, 1996. IEEE Computer Society.

[3] B.-M. Chang, J.-W. Jo, and S. H. Her. Visualization of exception propagation for java using static analysis. In *SCAM'02: Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation*, October 2002.

[4] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy. In *Proceedings of 9th European Conference on Object-oriented Programming (ECOOP'95)*, pages 77–101, 1995.

[5] C. Fu, A. Milanova, B. G. Ryder, and D. G. Wonnacott. Robustness Testing of Java Server Applications. *IEEE Transactions on Software Engineering*, 31(4), Apr. 2005.

[6] N. Heintze. Set-based analysis of ml programs. In *Proceedings of the ACM Conference on Lisp and Functional Programmig*, pages 306–317, 1994.

[7] `http://www.Eclipse.org`. The Eclipse IDE, 2005.

[8] J.-W. Jo, B.-M. Chang, K. Yi, and K.-M. Cho. An uncaught exception analysis for Java. *Journal of Systems and Software*, 2004. in press.

[9] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In G. Hedin, editor, *Compiler Construction, 12th International Conference*, volume 2622 of *LNCS*, pages 153–169, Warsaw, Poland, April 2003. Springer.

[10] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification. Second Edition*. Addison Wesley, 1999.

[11] K. Nagaraja, X. Li, B. Zhang, R. Bianchini, R. P. Martin, and T. D. Nguyen. Using fault injection to evaluate the performability of cluster-based services. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS 2003)*, Seattle, WA, Mar. 2003.

[12] S. Rapps and E. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, SE-11(4):367–375, Apr. 1985.

[13] D. Reimer and H. Srinivasan. Analyzing exception usage in large java applications. In *EHOOS'03: ECOOP2003 - Workshop on Exception Handling in Object Oriented Systems*, July 2003.

[14] M. P. Robillard and G. C. Murphy. Static analysis to support the evolution of exception structure in object-oriented systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 12(2):191–221, 2003.

[15] S. Sinha and M. J. Harrold. Criteria for testing exception handling constructs in java programs. In *Proc. Int'l Conf. Software Maintenance*, Sep. 1999.

[16] S. Sinha and M. J. Harrold. Analysis and testing of programs with exception-handling constructs. *IEEE Transactions on Software Engineering*, 26(9):849–871, September 2000.

[17] S. Sinha, A. Orso, and M. J. Harrold. Automated support for development, maintenance, and testing in the presence of implicit control flow. In *Proc. Int'l Conf. Software Engineering (ICSE'04)*, 2004.

[18] P. Sortokin. Ftp server in java, 2003.