Testing of Java Web Services for Robustness[*]

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

Ana Milanova
Department of Computer Science
Rensselaer Polytechnic Institute
Troy, NY 12180
milanova@cs.rpi.edu

David Wonnacott
Department of Computer Science
Haverford College
Haverford, PA 19041
davew@cs.haverford.edu

ABSTRACT

This paper presents a new compile-time analysis that enables a testing methodology for white-box coverage testing of error recovery code (i.e., exception handlers) in Java web services using compiler-directed fault injection. The analysis allows compiler-generated instrumentation to guide the fault injection and to record the recovery code exercised. (An injected fault is experienced as a Java exception.) The analysis (i) identifies the *exception-flow 'def-uses'* to be tested in this manner, (ii) determines the kind of fault to be requested at a program point, and (iii) finds appropriate locations for code instrumentation. The analysis incorporates refinements that establish sufficient context sensitivity to ensure relatively precise def-use links and to eliminate some spurious def-uses due to demonstrably infeasible control flow. A runtime test harness calculates test coverage of these links using an *exception def-catch* metric. Experiments with the methodology demonstrate the utility of the increased precision in obtaining good test coverage on a set of moderately-sized Java web services benchmarks.

Categories and Subject Descriptors

D.1.5 [**Object-oriented programming**]: Exception Handling; D.2.5 [**Testing and Debugging**]: Error Handling and Recovery; D.2.5 [**Testing and Debugging**]: Testing Tools; D.4.5 [**Reliability**]: Fault-tolerance; F.3.2 [**Semantics of Programing Languages**]: Program Analysis

General Terms

Reliability

Keywords

Def-Use Testing, Java, Exceptions, Test Coverage Metrics

1. INTRODUCTION

The emergence of the Internet as a ubiquitous computing infrastructure means that a wide range of applications – such as on-line auctions, instant messaging, grid weather prediction programs – are being designed as web services. These services must meet the challenges of maintaining performance and availability, while supporting large numbers of users, who demand reliability from these codes that are becoming more and more commonplace. A good analogy is to the telephone system, a technology that one expects to be 'always working'; the phone company demands only minutes of down time per year from its software. New testing technologies are needed to address the issue of reliability in this environment. Besides the traditional testing of functionality, there is a need to ensure reasonable application response to system/resources problems, in order to have performance gracefully degrade rather than experience application crashes. The robustness testing research in this paper addresses the problem of how to test the reliability of Java web services in the face of infrequent, but anticipatable system problems, which are responded to using Java's exception handling mechanism.

Traditional fault-injection testing of software in the operating system community is conducted in a black-box manner, using a probabilistic analysis to determine whether or not a software component will work properly when subjected to specific fault loads and workloads [2]. Testing is accomplished by simulating faults caused by environmental errors during test through *fault injection* [10, 12, 18, 21, 39]. Testers assume that applications run under specific workloads, and then inject faults randomly into the running code, selecting faults according to distribution functions derived from observation of real systems. After observing application reaction to the fault load, the testers derive data describing the likelihood that the application will deliver correct service (i.e., not crash) under the given fault loads and workloads [2].

Unfortunately, this approach does not ensure that the error recovery code in an application is ever exercised nor that the program takes an appropriate action in the presence of faults. In addition, given the probabilistic nature of the approach, it is hard to force application execution into the untested parts of error recovery code during further testing. Because many web services are written using components with unknown internal structure, testers need to identify vulnerabilities to system problems automatically (i.e., with the help of software tools). The testing of error recovery code in web services is necessary for ensuring the high reliability required of these systems.

Our methodology uses the tools of white-box def-use testing to aid a tester of web services in this task. There is a large body of existing work on *white-box* testing methodologies [5, 29, 17], aimed at exercising as much application code as possible during testing, and measuring code coverage using various program constructs such as control-flow edges, branches and basic blocks. However, error recovery code — code which handles errors that occur with small probability, especially due to interactions with the comput-

*This work was supported in part by NSF grants EIA-0103722 and CCR-9900988

ing environment (e.g., disk crashes, network congestion, operating system bugs) — is almost always left unexecuted in traditional white-box testing, because it may not be executable by merely manipulating program inputs.

Our analysis techniques identify program points vulnerable to certain faults and the corresponding error recovery code for these specific system faults. The techniques provided allow compiler-inserted instrumentation to inject appropriate faults as needed and to gather recovery code coverage information. This enables a tester to systematically exercise the error recovery code, by causing execution to exercise the vulnerable operations. Thus the methodology provides a means to obtain validation of application robustness in the presence of system faults. Although our experiments are based on web applications, the technique is not limited in that area and can be applied on general Java applications.

In our approach, it is important to be able to identify as precisely as possible where an exception, thrown in response to an experienced fault (i.e., a def), is handled (i.e., a use). A key concern in general for def-use testing is how to minimize the number of spurious def-uses reported by the analysis. Since these def-uses cannot be exercised by any test, a human being has to examine them, among the uncovered def-use links after testing, and determine (if she can) that they are spurious. This is a time-consuming, difficult job, especially for large object-oriented applications that use polymorphism heavily. Therefore, it is crucial to use a very precise analysis that, while practical in cost, can eliminate many of these spurious def-uses. This is a key goal of our new *exception-catch link analysis*.

Our target applications are Java web services because these programs are widely used to build large-scale distributed cooperative systems. Java is used increasingly to build components for these services. Furthermore, the exception construct and mandatory exception handling mechanism facilitates both construction and analysis of error recovery in a Java program, thus providing a good basis for validating our methodology for automatic identification and testing of error recovery code.

In a previous paper [15], we gave a general overview of our methodology for testing of error recovery code, and defined appropriate coverage metrics. We presented a proof-of-concept case study in which a proxy server application was instrumented by hand, and then fault injection was performed and recorded by executing the instrumentation. In this paper we have defined and implemented a compile-time exception-catch link analysis, fully automated the program instrumentation process, and experimented with several versions of analysis on a data set of moderately-sized web service applications.

The specific contributions of this paper are:

- Design of a new compile-time exception-catch link analysis to identify error recovery code in relation to certain resource usage program points (i.e., a def-use analysis for potential exceptions involving resource usage). This analysis essentially is an interprocedural def-use dataflow analysis calculation with two new refinements: (i) performing a points-to analysis using limited context sensitivity by inlining constructors that set object fields (in order to avoid conflating objects, especially in libraries with long call chains) and (ii) using the absence of data reachability through object references to confirm the *infeasibility* of some links, by showing the corresponding interprocedural paths to be infeasible.
- Demonstration of *automatic* program instrumentation directed by our analysis, that effectively constructs a compiler-directed fault injection engine from *Mendosus* [24], an existing fault injection framework.
- Empirical validation of our methodology using several moderately-sized Java web service applications, including comparison of our new analysis with less precise, less costly class-based analysis adapted to find exception-flow def-uses. These studies demonstrate the appropriateness of the precision of our analysis for this task, in that on average, 84% of all exception-flow def-use links are covered by the testing.

Overview. The rest of this paper is organized as follows. In Section 2 we describe our coverage metric, which is a slight variant of the original metric described in [15], and give an overview of the compiler-directed fault injection methodology. In Section 3, we discuss our compile-time analysis for exception-flow def-uses and its precision increasing refinements. In Section 4 we report our empirical results on moderate-sized Java applications, describing the impact on the exception-flow def-uses obtained, of varying the compile-time analysis used. In Section 5 we describe related work. Finally, we present our conclusions.

2. MEASURING COVERAGE OF FAULT-HANDLING CODE

We take advantage of the Java exception handling mechanism to help identify error recovery code. *Exceptions* in Java are used to respond to error conditions [3]. Each `catch` block is potentially the starting point of error recovery code for a matching error/exception raised during the lifetime of the corresponding `try` block.

Faults, Exceptions, Coverage Metric. A *fault* is some environmental error that being manifested. We begin with a set of faults that are of interest to the tester — for example, some testing may focus on disk and network errors. A fault-sensitive operation, which is either an explicit `throw` statement or a call to unknown method, is *affected* by a fault in that an exception is produced when the operation occurs and experiences a fault as a run-time error. Often these operations are calls to C library functions within the Java JDK libraries. We denote P to be the set of all fault-sensitive operations that may be affected by any element in the specific set of faults of interest. We assume P is known, because it can be precalculated once from the Java libraries and reused for all the programs subject to fault-injection testing with this same set of faults. In this paper we focus on faults related to Java *IOExceptions*.

In any given program execution, each element of P could possibly produce an exception that reaches some subset of the program's `catch` blocks. [1] By viewing fault-sensitive operations as the definition points of exceptions, and `catch` blocks as uses of exceptions, we can define a coverage metric in terms of *exception-catch (e-c) links*. This definition is analogous to the *all-uses* metric [33] of traditional def-use analysis:

Definition (*e-c link*): Given a set P of fault-sensitive operations that may produce exceptions in response to the faults of interest, and a set C of `catch` blocks in a program to be tested, we say there is a *possible e-c link* (p, c) between $p \in P$ and $c \in C$ if p could possibly 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.

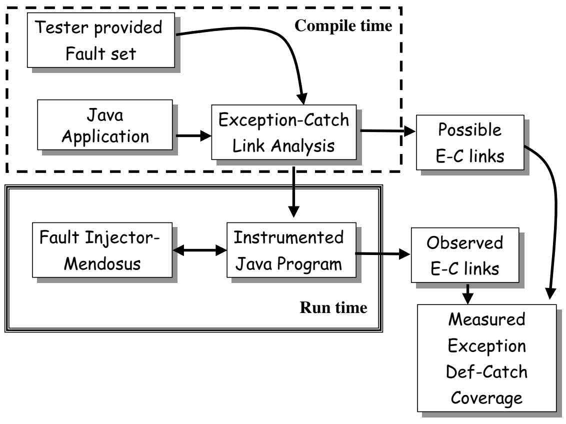
Definition (*Overall Exception Def-catch Coverage*): Given a set F of the possible *e-c links* of a program, and a set E of the *e-c links* experienced in a set of test runs T, we say the *overall exception def-catch coverage* of the program by T is $\frac{|E|}{|F|}$.

[1]There is a many-to-many relationship between system faults and Java exceptions [15]. For this paper we assume that the tester merely has to choose one or more exceptions of interest. For more details, see [15].

A high overall exception def-catch coverage indicates a thorough test, but a low coverage may result from either insufficient testing (i.e., a small E) or an overly conservative estimate of F, the set of *possible e-c links*. As in other forms of coverage testing, it is unacceptable for F to omit any *e-c links* possible at runtime, so our analysis must be conservative, producing a superset of F in the presence of imprecision. This is a common problem in software testing; it is addressed by using an analysis that is *as precise as possible* to eliminate many infeasible paths and by human tester examination. As we will see in Section 4, the precision of our analysis has a significant impact on the coverage results for the benchmarks.

Fault Injection Framework. Once we have calculated the possible *e-c links* for a program with the analysis in Section 3, then for a specific fault-sensitive operation, we have identified the `catch` blocks that may handle the resulting exception, if it occurs. Given the semantics of Java, there must be a *vulnerable* statement executed during the corresponding `try` block, that resulted in the execution of the fault-sensitive operation. The tester must try to have execution exercise both this vulnerable statement, often a call, and the fault-sensitive operation, so that the recovery code is reached. Obtaining test data to accomplish this task is the same test case generation problem presented by any def-use coverage metric.

The compiler uses the set of *e-c links* found to identify where to place the instrumentation that will communicate with *Mendosus* [24], the fault injection engine, during execution. This communication will request the injection of a particular fault when execution reaches the `try` block containing the vulnerable operation and will result in the recording of the execution of the corresponding `catch` block.



Tester provided Fault set
Compile time
Java Application
Exception-Catch Link Analysis
Possible E-C links
Fault Injector-Mendosus
Instrumented Java Program
Observed E-C links
Run time
Measured Exception Def-Catch Coverage

Figure 1: Compiler-directed fault injection framework

Figure 1 shows the organization of our fault-injection system. The box labeled *compile time* shows that for a chosen set of faults, corresponding to some set of exceptions and their fault-sensitive operations, the analysis presented in Section 3 calculates the possible *e-c links* and the vulnerable statements that are susceptible to them. The compiler inserts the instrumentation calling on Mendosus to insert a fault during execution of the corresponding `try` block and the recording instrumentation for recovery code in the `catch` block. Then, the tester runs the program and gathers the *observed e-c links* from that run. The tester then may have to try to make the program execute other vulnerable statements (i.e., by varying the inputs) in order to cover more of the possible *e-c links*. Finally, the test harness calculates the overall exception def-catch coverage for this test suite.

3. COMPILE-TIME ANALYSIS

Figure 2 illustrates the high level structure of the two-phased compile-time exception-catch link analysis which we designed to calculate *e-c links* in Java programs. **Exception-flow** analysis takes a static representation (i.e., AST) of a Java program as well as its call graph, and produces the *e-c link* set of the given program. Unlike previous exception-flow analysis [34, 20, 44] which relied on interprocedural propagation of exception types, our analysis is object-based, distinguishing between exception objects created by different `new()` statements. The **DataReach** analysis serves as a postpass filter which uses the reference points-to graph [35, 37] of the program to discard as many infeasible *e-c links* in the set produced by exception-flow analysis as possible, so as to increase the precision of the entire analysis. Intuitively, both of these analysis phases can vary in their precision, because they effectively are parameterized by the points-to and call graph construction analysis used as their inputs. Various analysis choices are available for call graph construction [13, 4, 16] which differ in their cost and the precision of the resulting graph. The empirical results discussed in Section 4 show that the precision of the call graph and points-to graph has significant impact on the precision of the final *e-c link* set obtained.

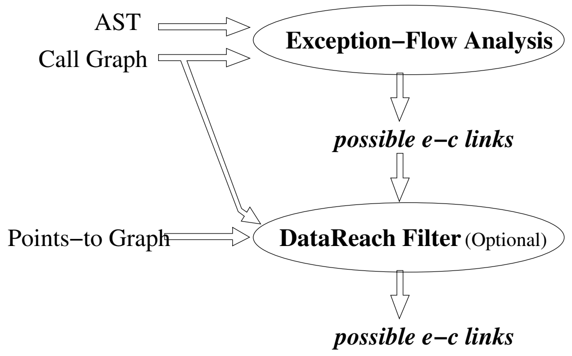


Figure 2: Two phases of exception-catch link analysis

3.1 Exception-flow analysis

In Java, if code in some method throws an exception[2] either the exception is handled within the method by defining a `catch` block for it, or the method declares in its signature that it might throw this kind of exception when called. In the latter case, its callers must either handle the exception or declare that they throw it as well [3]. We want to find the relationship between `catch` blocks and fault-sensitive operations. We use "`throw` statement" to represent all fault-sensitive operations in our discussions for simplicity; we actually mean all instructions or calls that may throw some exception, if a fault occurs.

A naive analysis that relies only on examination of user declared exception types in `catch` blocks and method signatures is too inaccurate to yield information of practical use. In part this is because the declared exception can be a supertype, subsuming many exception types that actually cannot be thrown in this context. Moreover, a method may declare that some exception may be thrown, when actually no exceptions can ever be raised; this can occur when the implementation of some method has changed, but the method declaration is not updated. Dynamic dispatch can add to the imprecision of the declared exception information. Suppose class `A` is the superclass of `B` and method `bar()` is declared in both of them, but only `A.bar()` may throw an exception of class `E` when called. If

[2]We are only considering **checked** exceptions, since exceptions related to I/O faults are checked.

some other method `foo()` contains a call `a.bar()` for `a` of static type `A`, then `foo()` must define a handler for exception `E` or declare that it throws this exception. However if at runtime reference `a` always points to a `B` object, no exception can ever be thrown at the call site.

Our exception-flow analysis is an interprocedural dataflow analysis that calculates for each `catch` block, all the `throw` statements whose exceptions could potentially be handled by that `catch`. This is a form of *def-use* analysis as shown in the following section.

Exception-flow as a dataflow analysis. We define *exception-flow* as the flow of each exception object thrown per `throw` statement along the exception handing path [31] — from the `throw` statement to the `catch` block where it is handled.

According to the semantics of exception handling in Java [3], we can assume there exists a variable for each executing Java thread that refers to the currently active exception object. During execution, any `throw` and `catch` operations are definitions and uses of that variable, respectively. Thus, we can apply a variant of the traditional Reaching-Definition [1] dataflow analysis to this problem, but there are some unique aspects of exception-flow that require special handling:

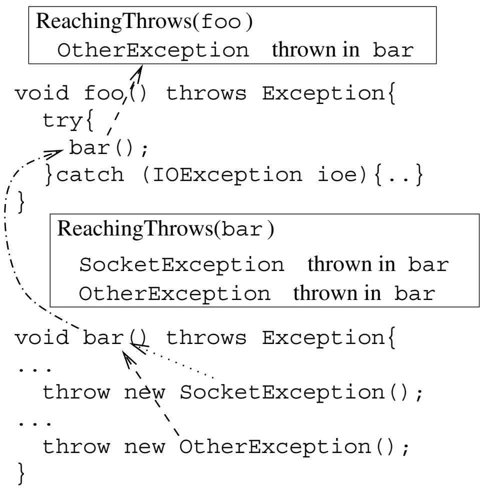
1. Types are associated with each use and definition. A use (i.e., a `catch`) *kills* all the reaching definitions whose type is a subtype of the type of the use.
2. The dataflow is in the reverse direction to execution flow; thus exception-flow is a backward dataflow problem.
3. The key control-flow statements in a method are `try` and `catch` blocks, `throw` statements and method calls. All other statements do not affect the exception-flow solution (given that the call graph is an input to this problem). The order of these statements within a method is of no consequence. What is important is whether or not a `throw` or method call is contained in a `try` block nest[3]. Therefore, within a method, we are only interested in paths from the method entry to each `try-catch` block or to a `throw` or a method call not contained in any `try-catch` block.

The analysis is interprocedural because of the nature of exception handling: an exception propagates along the dynamic call stack until a proper handler is reached. Our analysis is performed on a call graph whose edge annotations record the corresponding call sites, since call sites may occur within different `try-catch` blocks, which clearly affects the solution[4]. Within each method, the analysis calculates those exceptions which reach the entry to that method, by considering `throws` and method calls not contained within any `try-catch` block and those `try-catch` blocks within the method. The former statements yield some of the exceptions possibly raised and not handled in the method. Statements within the `try-catch` blocks may also yield unhandled exceptions, depending on the types of the respective `catch` blocks. Thus, the program representation used is a variant of a call graph, where each method node has an inner structure consisting of an edge from the entry node to each uncovered `throw` or method call, and an edge to each outermost `try-catch` block.

We define for each method the set of reaching exception objects that can reach its entry:

Definition (*ReachingThrows(method M)*): The set of all `throw` statements for which there exists an exception handling path [31] from the throw statement to method M, and the exceptions are not handled in method M.

Figure 3 gives an example illustrating the definition of *ReachingThrows*. We can see that the call site `bar()` inside method `foo()` is inside the `try` block, so that `SocketException` thrown in `bar()` will be handled (i.e., killed) in `foo()`. However, exception `OtherException`, also thrown by `bar()`, will not be handled and thus appears in *ReachingThrows(foo)*. If the call to `bar()` had not been placed within a `try-catch` block in `foo()`, both exceptions (i.e., `SocketException, OtherException`) would appear in *ReachingThrows(foo)*. Therefore, our analysis can be considered to have some *flow-sensitive* aspects, in that it captures the relation of `try-catch` blocks to the call sites and `throw` statements within them.



```
ReachingThrows(foo)
  OtherException   thrown in bar

void foo() throws Exception{
  try{
    bar();
  }catch (IOException ioe){..}
}

ReachingThrows(bar)
  SocketException   thrown in bar
  OtherException   thrown in bar

void bar() throws Exception{
...
  throw new SocketException();
...
  throw new OtherException();
}
```

Figure 3: Example of ReachingThrows

The dataflow equations for the *ReachingThrows* problem are defined on the annotated call graph of the program.[5] We define *RT(m)*, the ReachingThrows at the entry to method m, as

$$\begin{aligned} &RT(m) = \\ &\quad \{t \in T | type(gen(t)) - kill(trynest(t)) \neq \emptyset\} \\ &\cup \bigcup_{cs \in CS} \bigcup_{m' \in targets(cs)} \\ &\qquad \{t \in RT(m') | type(gen(t)) - kill(trynest(t)) \neq \emptyset\} \end{aligned}$$

where T is the set of `throw` statements in m; *gen(t)* is set of the exception objects thrown by *t*; *type(gen(t))* is the set of types of the objects in *gen(t)*; *trynest(k)* is the (possibly empty) nest of `try-catch` blocks containing statement *k*; *kill(trynest(k))* is the set of exception types handled by the `catch` blocks that correspond to *trynest(k)*, or $\emptyset$ if *trynest(k)* is empty; CS is the set of call sites in m; and *targets(cs)* is the set of all run-time target methods that can be reached by call site *cs* (there can be more than one target of a polymorphic call). Note also that the set difference operation must respect the exception inheritance hierarchy; subtraction of a kill set including exception type *et* must remove any exceptions of subtypes of *et* as well as *et* itself.

These dataflow equations are consistent with the definition of a monotone dataflow analysis framework [25] and therefore, amenable to fixed-point iteration.[6]

[3]In Java, `try` blocks can be nested within each other. Handlers are associated with exceptions in inner to outer order [3].

[4]Adding these annotations is not difficult for any call graph construction algorithm.

[5]Under certain conditions[3], `finallys` behave like `catches` and/or `throws`. Our algorithm handles these situations correctly, but we omit the details involving `finallys` for brevity.

[6]The iteration is only necessary here to handle interprocedural loops. Our implementation uses a prioritized worklist algorithm; nodes in the worklist are kept in postorder order.

Worst case complexity. The dataflow problem so defined is distributive and 2-bounded [25]; therefore, the complexity of the analysis is $O(n^2)$ where n is the number of methods. Given our program representation, the time cost of processing each method to find the constant terms in these equations is linear in the number of `try-catch` blocks, call sites and `throw` statements in the method, which is bounded above by k, the maximum number of statements in a method; this adds a kn term to the above complexity. Therefore, the overall worst case complexity is $O(n^2 + kn)$.

Analogous to classical dataflow use-def/def-use chains, our analysis produces *e-c links* between each of the `throw` statements and their corresponding `catch` blocks. By performing exception-flow analysis, we can find all the *e-c links* (t_i, h_j) where `throw` t_i can potentially trigger `catch` block h_j. Furthermore, by recording the interprocedural propagation path of t_i, we can provide the call chains from h_j to t_i to help the human tester understand why a specific *e-c link* is not covered in some test.

Selective constructor inlining. The exception-flow analysis described previously relies on having an annotated call graph for the program. In order to increase precision, we added selective context sensitivity to the points-to analysis that we use to build the call graph. Rather than building a full and costly context-sensitive points-to analysis, we performed *selective constructor inlining*; that is, we inlined each constructor at its call sites, when that constructor contained a *this* reference field initialization using one of its parameters. Without this transformation, a context-insensitive analysis would make it seem that the same-named fields of all objects initialized in this constructor could point to all the parameters so used [27, 26]. If we run a context-insensitive points-to analysis after this transformation, we obtain some degree of context sensitivity for constructors, eliminating some imprecision and obtaining a more precise call graph and points-to graph for both our exception-flow and DataReach analysis phases.

3.2 Data reachability analysis (DataReach)

We want to use a fairly precise program analysis to eliminate as many infeasible interprocedural paths as possible, to reduce the work that otherwise must be done by human testers when *e-c links* based on these paths cannot be covered. Using a more precise analysis for call graph construction such as points-to analysis [35, 37] helps to reduce the number of infeasible *e-c links* found. However, in practice even a very precise call graph building algorithm introduces many infeasible *e-c links*. Figure 4 is an example of typical use of the Java network-disk I/O packages. Figure 5 illustrates how infeasible *e-c links* are introduced even given a fairly precise call graph for the code. As we can see, the `try` block in `readFile` is only sensitive to disk faults and the `try` block in `readNet` is only sensitive to network faults. But exception-flow information is merged in `BufferedInputStream.fill()` and propagated to both `readFile` and `readNet`; thus, two infeasible *e-c links* are introduced reducing the possible runtime coverage to less than 50%.

This can be solved by using a different program representation such as a call tree [38] instead of a call graph. However, constructing a call tree by compile-time analysis is too expensive and once constructed, this representation is too large to scale appropriately. For example, to remove the infeasible *e-c links* in Figure 5, the call tree algorithm must be able to find that there are only 2 feasible call chains which share a middle segment of length 3. Separating these 2 chains would require a context-sensitive points-to analysis analogous to 4-CFA [40, 41], an expensive analysis. In many cases the length of the shared segment is even longer (e.g., when you need to wrap the basic InputStream with more than one filter class, such as `BufferedInputStream` & `DataInputStream`).

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

void readNet(Socket s){
 byte[] buffer = new byte[256];
 try{
  InputStream n =s.getInputStream();
  InputStream in=new BufferedInputStream(n);
  for (...)
   c = in.read(buffer);
 }catch (IOException e){ ...}
}
```

Figure 4: Code Example for Java I/O Usage

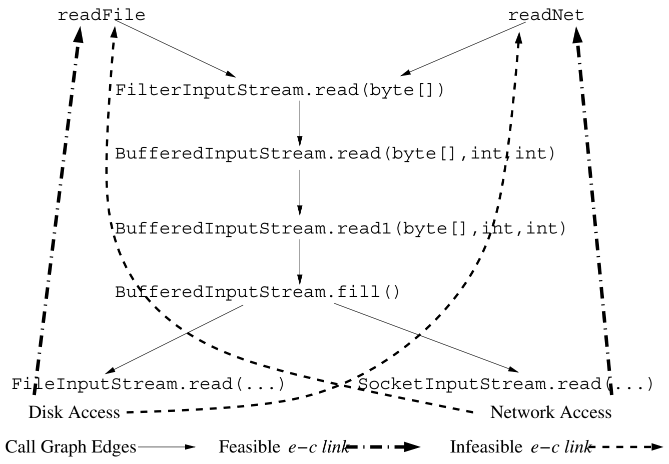

readFile
readNet
FilterInputStream.read(byte[])
BufferedInputStream.read(byte[],int,int)
BufferedInputStream.read1(byte[],int,int)
BufferedInputStream.fill()
FileInputStream.read(...)
SocketInputStream.read(...)
Disk Access
Network Access
Call Graph Edges
Feasible e–c link
Infeasible e–c link

Figure 5: Call Graph for Java I/O Usage

The intuitive idea of our approach is to use data reachability to confirm control-flow reachability, in that interprocedural paths requiring receiver objects of a specific type can be shown to be infeasible if those type of objects are not reachable through dereferences at the relevant call site. Continuing with Figure 4, consider the call site `in.read()` in method `readFile`. We want to know whether `SocketInputStream.read()` can be called during the lifetime of `in.read()`. In the explanation below, we refer to `in.read()` as the *original call* and to `SocketInputStream.read()` as the *target call site*. The argument about data reachability relies on the following intuition: if `SocketInputStream.read()` is called, some object of `SocketInputStream` must have been created previously to serve as the receiver. There are only three ways this can occur:

1. The object is created **during the lifetime** of the original call and passed to the target call site by assignments between method return values and local variables.
2. The object is associated with `in` by field dereferences of one of the global variables (i.e., Java static fields), that occur **during the lifetime** of the original call.
3. The object is associated with `in` by field dereferences of one of the arguments of the original call (including the receiver), that occur **during the lifetime** of the original call.

Therefore given an original call site, we can express the feasibility of a particular call path in terms of whether some data reachability is possible according to these conditions. For example, to show that *e-c link* referred to above is infeasible, we verify that there is no object in the points-to set of the receiver of the target call site with type `SocketInputStream` that can either be created in one of the methods reachable from the original call, or reachable by transitive field loads from the receiver or the arguments of the original call site or static fields. This means that the exception-flow def-use path is infeasible. Note, we only consider object fields and static fields loaded in *methods reachable from the original call*. Clearly, we need reasonably precise points-to information [23, 35] to obtain the high-quality data reachability information.

DataReach Algorithm. The DataReach algorithm requires that we have the points-to graph and call graph of the program [23, 35]. First, we calculate *universe*: the set of all methods that are reachable from the given original call (according to the call graph). This set contains all the instructions that can be executed during the lifetime of the original call. Second, we collect all the `new` statements in *universe* from which we can derive N: the set of all objects created during the lifetime of the original call. Third, we collect all the static field loads in *universe*, and calculate S: the union of the points-to sets of static fields loaded during the lifetime of the original call. Fourth, we calculate P: the union of points-to sets of arguments (including receivers) of the original call site, and set $U = N \cup S \cup P$. Fifth, we collect all the instance field loads in *universe* and calculate U^*: the closure of U under the instance field dereferences that may occur during the lifetime of the original call. Finally, we intersect U^* with the points-to set of the receiver of the target call site. If we are trying to prove the infeasibility of a particular library call for example, we merely need to show that there are no objects in the intersection with type appropriate for the call to have occurred.

The process described is used to judge the feasibility of a particular call edge. It can be applied repeatedly to examine each call edge downstream from the given call site. The algorithm in Figure 6 is based on this idea. It starts from the given call site, does a breadth-first search on the call graph and judges the feasibility of each encountered call edge before actually following it. This algorithm outputs *reachable_methods*, the set of all methods reachable through data reachability from the given original call.

In summary, if a fault occurs during the original call, then an exception may be handled by a `catch` block associated with the `try` in which the original call is nested. In this case, there is a corresponding *e-c link* resulting from an excepting call to some method f or `throw` in method f during the lifetime of the original call. If the *reachable_methods* set does not contain f, then the *e-c link* is spurious (i.e., corresponds to an infeasible control-flow path).

Worst case complexity. The *while* loop iterates at most n (number of methods) times. At first glance, method *reachable()* may be called $n * E$ times where E is the number of call graph edges. But whenever *reachable()* returns false, the call edge can be added into a map indexed by the objects needed to make the call edge "reachable". And when more objects are added into U, the map can be checked to instantiate some of the call edges. We have the references to the objects and we can implement the "map" by adding annotations on object nodes in the points-to graph, thus both of these operations are constant time. So *reachable()* only needs to be called E times. The cost of running *reachable()* will not exceed the number of objects pointed to by the receiver, which is bounded by, but often much smaller than, r: number of objects in the points-to graph (i.e., the total number of object creation sites in the program). For calculating U, remember that the algorithm collects an object along an edge in the points-to graph at most once. Assume that the maximum number of fields in an object is t. Then, over the entire algorithm we explore at most $O(r^2t)$ edges in the points-to graph. Thus, the worst case complexity of DataReach is dominated by $O(E * r + r^2t)$ (i.e., at most cubic in terms of the program size, where both E and r are proportional to $O(n)$ in practice).

```
Boolean reachable(U, receiver, method)
{
  if method is private or static, return true;
  intersection = U ∩ receiver's points-to set
  if there are objects in intersection
        with type that resolve to method
    return true;
  else return false;
}

Set closure(U, fieldset)
{
  for each object in U
    for each field of object
      if (field in fieldset)
        U += points_to_set(object.field);
}

Main Algorithm:
reachable_methods = empty
fieldset = empty
pending_arcs = call edges from the original call site
U = points-to sets of arguments of the original call site
while reachable_methods changed
  for each arc in pending_arcs
    if reachable(U, arc.receiver, arc.target_method)
      remove arc from pending_arcs
      reachable_methods += arc.target_method
      pending_arcs += call edges from arc.target_method
      fieldset += instance field references in arc.target_method
      U += new objects created in arc.target_method
      U += points-to sets of static fields in arc.target_method
      U = closure(U, fieldset)
    end if
  end for
end while
```

Figure 6: DataReach Algorithm

4. EMPIRICAL RESULTS

In this section we discuss the instrumentation used in our methodology and report our experimental findings.

4.1 Instrumentation

The methodology described in Section 2 requires that the Java program be instrumented to report coverage of the *e-c links* exercised and to communicate with *Mendosus* to request specific faults. A detailed description of the methodology was described in our previous paper [15]; we briefly summarize it here.

The instrumentation is accomplished through method calls. For each *e-c link* (p, c), we first locate the `catch` block c, and the corresponding `try` block. At the entry of the `try` block, a special method call is inserted to direct *Mendosus* to inject the fault selected at static instrumentation time. At the entry of the `catch` block another method call is inserted to query and record the call stack encapsulated in the caught exception. The instrumentation methods called are designed so that each instrumentation point can be turned on and off by a command line option. Note that the fault must be selected so that exactly one fault-sensitive operation will fail and throw an exception. In addition, we record the I/O objects created by the user code during execution, in order to limit the scope of the injected faults to this set.

4.2 Experimental setup & benchmarks

We implemented exception-flow analysis and DataReach analysis as two separate modules in the Java analysis and transformation framework Soot [37] version 2.0.1, using a 2.8GHz P-IV PC with Linux 2.4.20-13.9 and the SUN JVM 1.3.1_08 for Linux. By separating the two phases of our analysis, we were able to show the gains from adding the DataReach postpass. Soot provides a call graph builder using *Class Hierarchy Analysis* (CHA) [13]. We implemented another call graph builder using *Rapid Type Analysis* (RTA) [4]. Soot also provides *Spark*, a field-sensitive, flow-insensitive and context-insensitive points-to analysis (a form of 0-CFA) [41, 36, 35, 23]. The instrumentation phase is also implemented as a separate module in Soot.

We experimented with the following six different analysis configurations:[7]

1. CHA — Build call graph with Class Hierarchy Analysis.
2. RTA — Build call graph with Rapid Type Analysis.
3. PTA — Build call graph using Spark.
4. InPTA — Build call graph with Spark plus selective constructor inlining.
5. PTA-DR — Use Spark to provide the points-to graph and call graph plus use DataReach as a postpass filter.
6. InPTA-DR — Use Spark plus selective constructor inlining to provide the points-to graph and the call graph, and use DataReach as a postpass filter.

We used four Java web service applications of moderate size as our benchmarks.

- FTPD, a Ftp Server in Java by Peter Sorotokin v0.6
- JNFS, a server application that runs on top of a native file system and listens to and handles requests for both read and write accesses to files. The server communicates with various clients via RMI [32]
- Haboob, a simple web server based on SEDA, a staged event-driven architecture [48]
- Muffin, a web filtering proxy server [28]

Name	*Classes*	*Methods*	*LOC*
FTPD	11(1407)	128(7479)	2783
JNFS	56(1664)	447(9603)	10478
Haboob	338(1403)	1323(7432)	39948
Muffin	278(1365)	2080(7677)	32892

Table 1: Benchmarks

Column 2 of Table 1 is the number of user classes, with those in parenthesis comprising the JDK library classes reachable from each application. The data in column 3 are the number of user methods and those in parenthesis are the JDK library methods reachable from each application. Column 4 gives the number of lines of code in user code source files. The method reachability information is calculated by Spark, with lines of code calculated using the UNIX *wc* utility. JNFS is the only multi-node application.[8]

[7]Selective constructor inlining and DataReach were only used where stated explicitly.

[8]Currently, we assume the network supporting RMI is reliable; i.e., we ignore faults that affect RMI transportation.

As shown in Figure 1, dynamic testing is conducted by running the instrumented code with various workloads to exercise different vulnerable points in the applications. Experienced *e-c links* are recorded in a log file during the test. By processing the *e-c link* information file and log file after the test we obtain the coverage data. The dynamic tests were performed on a cluster of 800MHz PIII PCs using Linux 2.2.14-5.0; we used IBM Java 2.13 Virtual Machine for Linux for all of our benchmarks. *Mendosus* was running as a daemon process on each of these machines.

In this testing we made the usual assumptions that (i) faults are independent of each other, and (ii) faults occur rarely. We only injected one fault per run, resulting in at most one *e-c link* covered per test run; therefore, we needed to run each benchmark several times, each time targeting one *e-c link*. Because we lack a model for faults that tend to happen together, systematically testing more than one fault at a time is difficult. A testing harness was constructed, which iterated over the *e-c links* information file, repeatedly running one benchmark program as necessary. As usual it was the tester's responsibility to find proper inputs and program configurations, so that designated vulnerable statement (and fault-sensitive operation) were executed.

4.3 Empirical data

Table 2 lists the number of *e-c links* reported for each benchmark in each analysis configuration. The last column shows the number of *e-c links* actually covered for each benchmark in the fault injection test. Table 3 is the overall exception def-catch coverage for all the benchmarks derived from the data in Table 2. We can see from the tables that the use of points-to analysis for call graph construction dramatically reduced the number of *e-c links* reported in all of the benchmarks. With RTA or CHA, the number of false *e-c links* reported is 2 to 6 times more than the actual *e-c links* that we can cover in the testing.[9]

Program	CHA	RTA	PTA	InPTA	PTA-DR	InPTA-DR	Covered
FTPD	34	34	16	16	16	13	**11**
JNFS	104	104	39	39	22	19	**16**
Haboob	96	73	12	12	12	12	**10**
Muffin	480	258	112	112	87	42	**35**

Table 2: Number of *e-c links*

Program	CHA	RTA	PTA	InPTA	PTA-DR	InPTA-DR
FTPD	32%	32%	69%	69%	69%	85%
JNFS	15%	15%	41%	41%	72%	84%
Haboob	10%	14%	83%	83%	83%	83%
Muffin	7%	14%	31%	31%	40%	83%

Table 3: Overall Exception Def-catch Coverage, in percentage

The context sensitivity obtained by adding selective constructor inlining before performing points-to analysis had no effect on any of the benchmarks, when we only considered call graph construction. However, when combined with the DataReach postpass, the additional precision provided, reduced the number of reported *e-c links* in three out of four benchmarks (i.e., compare columns PTA and InPTA-DR in Table 2). For the *e-c links* reported by InPTA-DR, the coverage percentage of all four benchmarks was stabilized at approximately 84% with small variance. In Muffin, the additional precision helps cut the number of reported *e-c links* by more

[9]Recall that all of these analyses are *safe* meaning that if one analysis fails to report a given *e-c link* that another analysis reports, then this *e-c link* is spurious.

than half (see Table 2). Thus, DataReach is a client of precise points-to analysis, where added precision can make a difference.

Haboob is special in that it is the only benchmark that uses a self-constructed non-blocking network library, which does not have as much polymorphism as the standard JDK library. This is why the simple PTA analysis is sufficient to analyze Haboob, as shown in Table 2.

Figure 7 shows the running times of each part of the static analysis on each benchmark using configurations PTA-DR and InPTA-DR. Running times of the instrumentation phase are too small to be shown, under 2 seconds for all the benchmarks. Our current analysis always finished in less than an hour. In the worst case for the InPTA-DR configuration, the time our analysis took to find one *e-c link* in a program is less than 3 minutes. DataReach is time consuming, but it is effective in reducing spurious *e-c links* (i.e., comparing the columns for PTA and PTA-DR, inPTA and inPTA-DR in Table 2). For two of the benchmarks, Muffin and JNFS, where our analysis took much longer to finish, DataReach used about 90% of the total running time; for the other two benchmarks, it used about 50% of the total running time. We believe that an optimized implementation of DataReach will improve overall analysis performance significantly.

4.4 Uncovered *e-c links* in Muffin

Using the InPTA-DR analysis we were able to identify and cover `catch` blocks related to I/O fault recovery, leaving only a small portion, 16% to 17%, as needing human inspection. We examined more closely some of these uncovered *e-c links* to find out why they remained uncovered by our testing. As a case study we used our Muffin benchmark, examining the seven uncovered *e-c links* produced by InPTA-DR. They can be partitioned into 2 categories according to the reasons why they were not covered.

The first category consists of *e-c links* not covered for subtle reasons which are really hard to discover through static analysis, but not so difficult for programmers to reason about. There are 4 *e-c links* in this category. One of them involves a `try-catch` block which handles exceptions thrown because of faults in a TCP connection. By examining the code we found that it is part of a resolver which translates machine names (i.e., ASCII strings) to IP addresses by communication (coded in another method with separate `try-catch` block) with a given DNS server. However, TCP is only used when a message is large enough, which will not occur since the messages are just domain names and IP addresses. Although this *e-c link* was not covered, the input data required to cover this *e-c link* would need to include extremely long URL names to force use of TCP; this is information that a human tester can determine but is very difficult for an automatic analysis to ascertain. By not being able to cover this *e-c link* easily, our methodology focuses the attention of the tester on this part of the code.

Consider the other 3 *e-c links* in the first category. In Muffin, the user can specify configuration files using URLs, which may be either remote (network access) or local (disk access). These 3 *e-c links* involve handling of network exceptions thrown when trying to modify some configuration file. While error messages will be given when a remote file is to be modified, no remote file would ever be written and these 3 *e-c links* are left uncovered.

The second category is composed of 3 *e-c links* which are difficult to confirm as feasible or not by human inspection. As mentioned in Section 3.1 our analysis provides the call chains that start from c_j and end with p_i for any *e-c link* (p_i, c_j). But even with these call chains given, the job of deciding whether an uncovered *e-c link* in this category is actually infeasible is hard, since these call paths are prohibitively long and confusing to trace. The example given below is one of the possible call chains found for one of these *e-c links*.[10] There are several hundred call paths given for this single *e-c link*.

```
org.doit.muffin.Handler.processRequest()
org.doit.muffin.Https.recvReply()
org.doit.muffin.Reply.read()
org.doit.muffin.Reply.read()
java.io.SequenceInputStream.read()
java.util.zip.GZIPInputStream.read()
java.util.zip.InflaterInputStream.read()
java.util.zip.InflaterInputStream.fill()
java.io.BufferedInputStream.read()
java.io.BufferedInputStream.read1()
java.io.BufferedInputStream.fill()
java.util.jar.JarInputStream.read()
java.util.zip.ZipInputStream.read()
java.util.zip.ZipInputStream.readEnd()
java.util.zip.ZipInputStream.readFully()
java.io.PushbackInputStream.read()
java.io.FilterInputStream.read()
java.io.FileInputStream.read()
```

We inspected these call chains and found all of the call chains for this particular *e-c link* share the same prefix, but after `SequenceInputStream.read()` they begin to vary by selecting `read()` methods from different subclasses of `InputStream` and following different permutations of calls. After reading the source code of `SequenceInputStream` we found that this class uses an `Enumeration` class to keep track of subsequent `InputStreams`. Although no object of `GZIPInputStream` has ever been assigned to the subsequent input stream of `SequenceInputStream`, the usage of the container class confuses the points-to analysis into producing the current result: `read()` in `SequenceInputStream` may call `read()` in `GZIPinputStream` and also almost every subclass of `InputStream`.

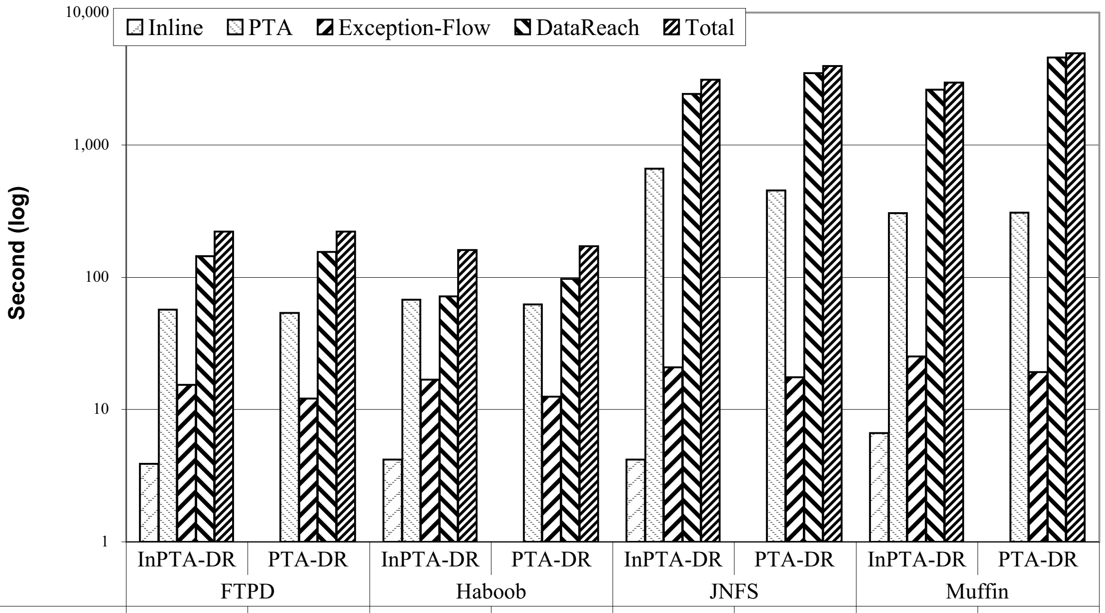
Call chains for all 3 *e-c links* share the same characteristics described here; they all involve the use of containers. This phenomenon is caused by context-insensitive points-to analysis, in a manner similar to the analysis imprecision for constructors discussed previously. More precise points-to analysis [26] addresses this problem by distinguishing calls by their receiver object when analyzing methods, thus producing a less connected points-to graph; this should reduce the call chains for a *e-c link*, or maybe even make possible for DataReach to judge that the *e-c link* is actually infeasible. We believe that additional context sensitivity added to the points-to analysis would further improve the precision of our *e-c links*, but further experimentation is needed.

5. RELATED WORK

This paper presents exception-catch link analysis and its use in def-use testing of Java program recovery code. There is much previous research relevant to this work, but due to limitations of space, we will discuss only the most closely related results.

Dataflow testing and coverage metrics. There is a large body of work that explores def-use or *dataflow testing* in different programming language paradigms. The seminal papers established a set of related dataflow test coverage metrics and explained their interrelations [33, 14]. The contribution of our work is to define and implement a def-use analysis of appropriate precision that fairly accurately matches exceptions (i.e., representative exception objects created at specific creation sites) to their handlers. This is espe-

[10]Parameters are omitted for readability.



	FTPD		Haboob		JNFS		Muffin	
	InPTA-DR	PTA-DR	InPTA-DR	PTA-DR	InPTA-DR	PTA-DR	InPTA-DR	PTA-DR
Inline	3.9		4.2		4.2		6.7	
PTA	56.7	53.7	67.7	62.2	659.4	453.1	304.5	307.2
Exception-Flow	15.4	12.1	16.8	12.5	20.9	17.5	25.1	19.2
DataReach	144.6	155.8	71.7	97.2	2,419.2	3,490.4	2,618.0	4,581.3
Total	220.6	221.6	160.4	171.9	3,103.7	3,961.0	2,954.3	4,907.7

Benchmarks

Figure 7: Time Cost Break-down of Static Program Analysis

cially important to ensure the dependability of the web applications that are our focus [15].

Sinha et. al defined an interesting and novel set of coverage metrics for testing exception constructs and gave their subsumption relations [42]. The metrics were defined for checked exceptions explicitly thrown in user code, however they seem easily extensible to both implicit and explicit checked exceptions. Our overall exception def-catch coverage metric seems equivalent to an extended version of their *all-e-deacts* criteria defined for both implicit and explicit exceptions. Because we are most interested in recovery code that deals with problems due to system interactions, we focus on implicit checked exceptions that are thrown in JDK libraries, whereas they deal with user-thrown exceptions, that are probably user-defined as well. No exception analysis or implementation experience with their metrics is presented.

The overall exception def-catch coverage metric for *e-c links*, that relates resource-usage faults to specific exception objects, differs slightly from our previous *overall fault-catch* coverage metric [15]. Our original metric required the injection of each kind of fault that could trigger a particular exception for a fault-sensitive instruction, rather than trying to cause a specific exception to occur. Both metrics are analogous to the *all-uses* metric in traditional def-use testing [33], with fault-sensitive operations corresponding to definitions of exceptions and `catch` blocks corresponding to uses. Overall fault-catch coverage requires the application of the complete range of faults during testing, consistent with existing operating systems fault-injection technology. In this paper, because we are injecting faults at the interface between JDK I/O methods and native methods rather than at the device-level [15], we cannot differentiate between some device-level faults that result in the same exception; thus we inject only one fault to trigger each exception.

As stated in Section 1, traditional fault-injection testing is performed by treating the application as a black box. Success is judged by how often the application does not crash in response to an injected fault. Other white-box, control-flow coverage metrics have been proposed by some groups for use with fault-injection testing; these correspond to previous metrics (e.g., branch, edge and basic block coverage) and have been summarized previously [15].

Analysis of exception handling. Two previous exception-flow analyses were aimed at improving exception handling in programs, for example avoiding exception handling through subsumption [34, 20]. These differ from our exception-catch link analysis in significant ways. First, their call graph is constructed using class hierarchy analysis, which yields a very imprecise call graph [13, 4]. Second, these analyses trace exception types through the call graph of the program to the relevant `catch` clauses that might handle them. Conceptually, these analyses use one abstract object per class. An operation that can throw a particular exception is treated as a source of an abstract object that is then propagated along reverse control-flow paths to possible handlers (i.e., `catch` blocks).

Jo et. al [20] present an interprocedural set-based [19] exception-flow analysis; only checked exceptions are analyzed. 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 [34] describe a dataflow analysis that propagates both checked and unchecked exception types interprocedurally. Neither approach analyzes Java libraries unless source code is available (not the case for the JDK). They each handle a large subset of the Java language, but make the choice to omit or approximate some constructs (e.g., *static initializers, finally*s). Both of these analyses are more imprecise than ours, especially in their approximation of interprocedural control-flow; neither of them trace definitions of specific exception objects to their appropriate `catch` blocks[11].

Another analysis of programs containing exception handling constructs [43] calculates control dependences in the presence of implicit checked exceptions in Java. This analysis focuses on defining a new interprocedural program representation that exposes exceptional control-flow in user code. In a more recent technical report [44], Sinha et. al present an interprocedural program representation which more accurately embeds the possible intraprocedural control-flow through exception constructs (i.e., `trys`,`catchs` and `finallys`). Class hierarchy analysis is used to construct the call edges in this representation. An exception-flow analysis is defined by propagation of exception types on this representation to calculate links between explicitly thrown checked exceptions in user code and their possible handlers. It seems clear that this analysis could be extended to include implicit checked exceptions as well, assuming that the program representation could be constructed from the bytecodes of the JDK library methods, and that the fault-sensitive operations could be identified. The CHA version of our analysis seems the most similar to the analysis presented in [44]; this version is shown on our benchmarks to be too imprecise for obtaining coverage of *e-c links* corresponding to implicit checked exceptions, the focus of our work.

Choi et. al [8] designed a new intraprocedural control-flow representation, that accounted for operations that might generate unchecked exceptions called *PEIs, potentially excepting instructions*; they used this representation as a basis for safe dataflow analyses for an optimizing compiler. It is difficult to compare their representation with the others described here, because they capture different sorts of exceptions, such as *NullPointerException*, that correspond to different possibly excepting instructions.

Exceptions and compilation. Dynamic analyses have been developed to enable optimization of exception handling in programs that use exceptions to direct control-flow between methods, such as some of the Java Spec compiler benchmarks [47]). The IBM Tokyo JIT compiler [31], successfully uses a feedback-directed optimization to inline exception handling paths and eliminate `throws` in order to optimize exception-intensive programs whose performance can be improved up to 18% without affecting performance of non-intensive codes. In *LaTTe* [22], exception handlers are predicted from profiles of previous executions and exception handling code is only translated in the JIT on demand, so as to avoid the cost when it is not necessary. The *MRL VM* [9] performs lazy exception throwing, in that it avoids creating exception objects, where possible, unless they are live on entry to their handler.

Points-to analysis. There is a wide variety of reference and points-to analyses for Java which differ in terms of cost and precision. The information computed by these analyses can be used as input to our exception-flow and data reachability analyses; clearly, the precision of the underlying analysis affects the quality of the computed coverage requirements. A detailed discussion of points-to and reference analyses and the dimensions of precision in their design spectrum appears in [36]. Our partially context-sensitive points-to analysis is most closely related to the context-sensitive analyses in our previous work [27, 26]. These approaches avoid the cost of non-discriminatory context sensitivity, which seems to be impractical; they rely on techniques which preserve the practicality of the underlying context-insensitive analysis while improving precision substantially. This is achieved by effectively selecting parts of the program for which the analysis computes more precise information, either by using parameterization mechanisms as in [27, 26], or partial constructor inlining as in our current algorithm. Other context-sensitive points-to analyses that seem to be substantially more costly than ours, are presented in [11, 16, 30, 7]; these analysis algorithms implement non-discriminatorily context sensitivity.

[11]Note, in our analysis we use the usual approximation of one representative exception object for each creation site, these two algorithms do not distinguish between exceptions of the same type created by two different sites.

Infeasible paths. Bodik et al. present an algorithm for static detection of infeasible paths using branch correlation analysis, for the purposes of refining the computation of def-use coverage requirements in C programs [6]. Our data reachability analysis focuses on the detection of infeasible paths in Java which arise due to object-oriented features and idioms such as polymorphism; this is not addressed in [6]. Souter and Pollock present a methodology (without empirical investigation) for demand-driven analysis for the detection of type infeasible call chains [45, 46]. Similarly to their work, our analysis is demand-driven as we analyze the program starting from the original call. However, our data reachability analysis propagates information in terms of objects instead of classes which will result in more precise analysis results. In addition, our work proposes a technique for summarizing the effects of callees; this problem is not addressed in [45] and [46]. Our simple RTA-like technique for collecting potential receiver objects proves suitable for the problem of eliminating infeasible *e-c* links; the empirical results demonstrate that it can eliminate substantial number of infeasible links.

6. CONCLUSIONS

We have defined a fairly precise exception-catch link analysis which has been shown useful on our benchmarks for testing error recovery code of Java programs. Our full analysis algorithm outperforms other (less precise) versions of the analyses that we investigated on our benchmarks, and exhibits significant precision gains in the set of *e-c links* calculated. Our use of data unreachability to infer control-flow unreachability shows promise in allowing us to prune spurious *e-c links*.

Our automatic compiler-directed fault injection methodology applied to our benchmarks leaves, on average, approximately 16% of the links uncovered and therefore needing to be examined by a human tester. This is an upper bound on the *false positive e-c links* that are reported for these benchmarks. Given that testing is by its nature an interactive activity, the uncovered *e-c links* can be seen as drawing a tester's attention to recovery code that requires human reasoning as part of the normal testing process.

Our future plans include testing application uses of other Java JDK libraries, such as *java.rmi*, and expanding our analysis to handle multi-node programs and middleware that use configuration files for dynamic loading of classes.

7. REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers – Principles, Techniques and Tools*. Addison Wesley, 1988.

[2] J. Arlat, A. Costes, Y. Crouzet, J.-C. Laprie, and D. Powell. Fault injection and dependability evaluation of fault-tolerant systems. *IEEE Transactions on Computers*, 42(8):913–923, Aug. 1993.

[3] K. Arnold and J. Gosling. *The Java Programming Language, Second Edition*. Addison-Wesley, 1997.

[4] 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)*, pages 324–341, Oct. 1996.

[5] R. V. Binder. *Testing Object-oriented Systems*. Addison Wesley, 1999.

[6] R. Bodik, R. Gupta, and M. L. Soffa. Refining data flow information using infeasible paths. In M. Jazayeri and H. Schauer, editors, *Proceedings of the Sixth European Software Engineering Conference (ESEC/FSE 97)*, pages 361–377. Springer–Verlag, 1997.

[7] R. Chatterjee, B. G. Ryder, and W. A. Landi. Relevant context inference. In *Proceedings of the ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, Jan. 1999.

[8] J.-D. Choi, D. Grove, M. Hind, and V. Sarkar. Efficient and precise modeling of exceptions for analysis of Java programs. In Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, pages 21–31, September 1999.

[9] M. Cierniak, G.-Y. Lueh, and J. M. Stichnoth. Practicing judo: Java under dynamic optimzations. In Proceeedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 13–26, 2000.

[10] M. Cukier, R. Chandra, D. Henke, J. Pistole, and W. H. Sanders. Fault injection based on a partial view of the global state of a distributed system. In *Symposium on Reliable Distributed Systems*, pages 168–177, 1999.

[11] J. D. David Grove, Greg DeFouw and C. Chambers. Call graph construction in object-oriented languages. In *Proceedings of ACM SIGPLAN Conference on Object-oriented Programing Systems, Languages and Applications (OOPSLA'97)*, pages 108–124, Oct. 1997.

[12] S. Dawson, F. Jahanian, and T. Mitton. ORCHESTRA: A Fault Injection Environment for Distributed Systems. In *Proc. 26th Int. Symp. on Fault Tolerant Computing(FTCS-26)*, pages 404–414, Sendai, Japan, June 1996.

[13] 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.

[14] P. Frankl and E. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, Oct. 1988.

[15] C. Fu, R. P. Martin, K. Nagaraja, T. D. Nguyen, B. G. Ryder, and D. Wonnacott. Compiler-directed program-fault coverage for highly available Java internet services. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2003)*, June 2003.

[16] D. Grove and C. Chambers. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(6), 2001.

[17] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, 3(4):279–290, July 1977.

[18] S. Han, K. Shin, and H. Rosenberg. DOCTOR: An Integrated Software Fault Injection Environment for Distributed Real-Time Systems. In *Int. Computer Performance and Dependability Symp. (IPDS'95)*, pages 204–213, Erlangen, Germany, Apr. 1995.

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

[20] 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.

[21] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham. FERRARI: A Tool for the Validation of System Dependability Properties. In *Proc. 22nd Int. Symp. on Fault Tolerant Computing(FTCS-22)*, pages 336–344, Boston, Massachusetts, 1992. IEEE Computer Society Press.

[22] S. Lee, B.-S. Yang, S. Kim, S. Park, S.-M. Moon, K. Ebcioglu, and E. Altman. Efficient Java exception handling in just-in-time compilation. In Proceedings of the ACM SIGPLAN Java Grande Conference, 2000.

[23] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In *International Conference on Compiler Construction*, LNCS 2622, pages 153–169, 2003.

[24] X. Li, R. P. Martin, K. Nagaraja, T. D. Nguyen, and B. Zhang. Mendosus: A SAN-Based Fault-Injection Test-Bed for the Construction of Highly Available Network Services. In *Proceedings of the 1st Workshop on Novel Uses of System Area Networks (SAN-1)*, Cambridge, MA, Jan. 2002.

[25] T. J. Marlowe and B. G. Ryder. Properties of data flow frameworks: A unified model. In *Acta Informatica, Vol. 28*, pages 121–163, 1990.

[26] A. Milanova. *Precise and Practical Flow Analsis of Object-oriented Software*. PhD thesis, Rutgers University, 2003. Also available as DCS-TR-539.

[27] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to and side-effect analysis. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 1–11, 2002.

[28] The Muffin world wide web filtering system. See `http://muffin.doit.org/`.

[29] G. J. Myers. *The Art of Software Testing*. John Wiley and Sons, 1979.

[30] R. O'Callahan. *The Generalized Aliasing as a Basis for Software Tools*. PhD thesis, Carnegie Mellon University, 2000.

[31] T. Ogasawara, H. Komatsu, and T. Nakatani. A study of exception handling and its dynamic optimization in java. In *Proceedings of ACM SIGPLAN Conference on Object-oriented Programing Systems, Languages and Applications (OOPSLA'01)*, pages 83–95, 2001.

[32] M. J. Radwin. The java network file system. See `http://www.radwin.org/michael/projects/jnfs/`.

[33] 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.

[34] 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.

[35] A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for java using annotated constraints. In *Proceedings of the Conference on Object-oriented Programming, Languages, Systems and Applications*, pages 43–55, 2001.

[36] B. G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In Proceedings of the Twelveth International Conference on Compiler Construction, pages 126–137, April 2003. invited paper.

[37] M. Sable. Soot: a java optimization framework. See `http://www.sable.mcgill.ca/soot/`.

[38] M. L. Scott. *Programming Language Pragmatics*. Morgan Kaufmann, 2000.

[39] Z. Segall, D. Vrsalovic, D. Siewiorek, D. Yaskin, J. Kownacki, J. Barton, D. Rancey, A. Robinson, and T. Lin. FIAT — Fault Injection based Automated Testing environment. In *Proc. 18th Int. Symp. on Fault-Tolerant Computing (FTCS-18)*, pages 102–107, Tokyo, Japan, 1988. IEEE Computer Society Press.

[40] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–234. Prentice Hall, 1981.

[41] O. Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, 1991.

[42] S. Sinha and M. J. Harrold. Criteria for testing exception-handling constructs in Java programs. In Proceedings of the International Conference on Software Maintenance, 1999.

[43] 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.

[44] S. Sinha, A. Orso, and M. J. Harrold. Automated support for development, maintenance, and testing in the presence of implicit control flow. Technical Report GIT-CC-03-48, College of Computing, Georgia Institute of Technology, September 2003.

[45] A. L. Souter and L. L. Pollock. Type infeasible call chains. In *Proceedings of the IEEE International Workshop on Source Code Analysis and Manipulation*, 2001.

[46] A. L. Souter and L. L. Pollock. Characterization and automatic identification of type infeasible call chains. *Information and Software Technology*, 44(13):721–732, October 2002.

[47] Specbench.org. Java client/server benchmarks.

[48] M. Welsh, D. E. Culler, and E. A. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Symposium on Operating Systems Principles*, pages 230–243, 2001.