

# Exception-chain Analysis: Revealing Exception Handling Architecture in Java Server Applications

Chen Fu                      Barbara G. Ryder

Division of Computer and Information Sciences, Rutgers University  
Piscataway, NJ 08854  
{chenfu,ryder}@cs.rutgers.edu

## Abstract

Although it is common in large Java programs to rethrow exceptions, existing exception-flow analyses find only single exception-flow links, thus are unable to identify multiple-link exception propagation paths. This paper presents a new static analysis that, when combined with previous exception-flow analyses, computes chains of semantically-related exception-flow links, and thus reports entire exception propagation paths, instead of just discrete segments of them. These chains can be used 1) to show the error handling architecture of a system, 2) to assess the vulnerability of a single component and the whole system, 3) to support better testing of error recovery code, and 4) to facilitate the tracing of the root cause of a logged problem. Empirical findings and a case history for Tomcat show that a significant portion of the chains found in our benchmarks span multiple components, and thus are hard to find manually.

## 1 Introduction

Current developments in languages and software engineering make it easier to reuse existing pieces of software to build large systems or to add functionality. However, the pervasive usage of COTS components complicates the task of achieving high *availability* for the entire system for the following reasons: First, since COTS components are separately developed and often poorly documented, if at all, understanding the behavior of the final integrated system under error conditions is hard; mastering the system's error recovery architecture is even harder. Second, an error may travel through several components before (if at all) being logged for future investigation. This makes it very difficult for a programmer to locate the root cause of an observed problem, if the knowledge of the error recovery behavior of the components and their interactions in the system is not available. Last but not least, error recovery codes are often least tested. Bugs in the error recovery code may exaggerate a small local fault, allowing it to stall the whole system, or silently let a critical problem go by without logging.

The Java programming language provides a program-level exception handling mechanism for responding to anticipated error conditions that occur during program execution. This mechanism helps separate exception handling

code from code that implements normal system behavior. Exception handling code might seem to provide a good starting point for code inspection to ensure system availability. However in our benchmarks, exception handling code that deals with certain kinds of faults is widely scattered over the whole program, and is mixed with other exception handling code, or even irrelevant code, making it hard to understand the behavior of the program under certain error conditions.

There are several compile-time program analyses of varying precision [5, 6, 8] that can be used to find the exception flow in a Java program (i.e., program paths from a `throw` statement to its corresponding `catch` clause). With the results of these analyses, a programmer can ask: What are the kinds of exceptions and/or the set of `throw` statements that can reach a given program point?

But in component-based systems, exception flow spanning different components often manifests as *chains* of exception `throws` and `catches`, instead of a single exception-flow link. Although individual exception-flow links can be obtained with relatively high precision, each link is only a discrete segment of the entire exception propagation path. Therefore, its utility in the discovery of the exception handling structure of the whole system, or in tracing back to the root cause of a logged problem of interest, is limited.

In this paper we propose a new compile-time analysis built on top of previous exception flow analyses [5]. It combines exception flow links into chains of exception handling paths, and thus reveals complete exception propagation paths. The contributions of this paper are:

1. Design of a new compile-time *Exception-chain* analysis to construct chains of exception-flow links whose corresponding exception objects are semantically-related. This analysis relies on a new intraprocedural *Handler-inspection* analysis that identifies `catch` clauses that *rethrow* either the incoming exception object and/or a newly constructed exception object based on the incoming one<sup>1</sup>. The results of *Handler-inspection* can be used to identify related exception-

---

<sup>1</sup>In this paper, the term *rethrow* refers to a `throw` within the `catch` clause (i) of the incoming exception object or (ii) of a new exception object containing semantic information from the incoming exception object.

flow links and combine them into chains and also can be used to rank the quality of catch clauses and to support better testing of error handling code.

2. Definition of a *service dependence graph*, a graphical depiction of exception flow between components. This graph is obtained from the exception-flow chains by abstraction; only inter-component edges of the chains are shown.

3. Empirical study of our methodology using several Java server applications, including a case history for Tomcat, demonstrating the uses of the analysis results: (i) to reveal the high-level architecture of the error handling code, (ii) to construct a non-trivial *service dependence graph* of components, and (iii) to assess the vulnerability of certain components as well as the whole system under different conditions.

With exception flow chains visualized in a service dependence graph, a tester or a system maintainer can see the exception-handling interaction between system components, and the root cause of either a testing failure or a system crash, without consulting the system source code.

**Overview.** The rest of this paper is organized as follows. Section 2 reviews the analysis introduced in previous work [5], on which our new analysis is based, and then motivates our new analysis. In Sections 3 and 4 respectively, we introduce our *Handler-inspection* analysis, and then discuss findings from our experiments. In Section 5 we describe existing static and dynamic analyses for finding exception-flow information, as well as other related work. Finally, we present our conclusions.

## 2 Background

Here we briefly review our previous work [5], on which the new analysis is based. Then, we discuss why this approach is not sufficient to reveal the exception handling architecture of a component-based system.

### 2.1 Exception Analysis Framework

In a Java program, each fault-sensitive operation (e.g., a call to a native method from the JDK to read from disk) may produce an exception that reaches some subset of the program's catch blocks. An *exception-catch (e-c) link* is defined as follows:

**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  $c$ .

The two pass static analysis algorithm in [5], comprised of *Exception-flow* and *DataReach* analysis, finds the possible *e-c links* in a Java program. *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 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.

It is obvious that the precision of *Exception-flow* analysis is affected by the precision of the call graph. But in practice even use of a very precise call graph may introduce

```

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){ ... }
    }
}

```

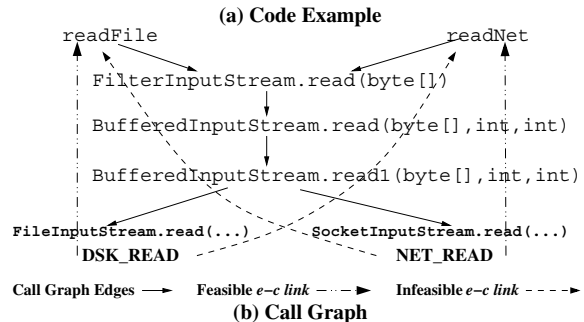


Figure 1. Java I/O Usage Example

many infeasible *e-c links*. Part (a) of Figure 1 is an example of typical uses of the Java I/O package. Part (b) illustrates the results of *Exception-flow* analysis based on a fairly precise call graph of the code: 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 inspecting the code manually, we can see that two of the reported *e-c links* (DSK\_READ, catch in readFile) and (NET\_READ, catch in readNet) are infeasible.

A second pass filtering analysis, *DataReach*, reduces the number of infeasible *e-c links* produced. The intuition is to use data reachability, obtained using points-to analysis, to confirm control-flow reachability. For example, continuing with Figure 1, if the goal is to prove SocketInputStream.read() is **not** reachable from the call site fsrc.read() in method readFile, we only need to prove that no object of type SocketInputStream is reachable during the lifetime of the method call fsrc.read(). Thus the following evidence is sufficient: 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, nor may be created by a new statement. Thus, 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 cannot be proved infeasible.

Figure 2 shows the organization of the automatic exception-flow testing system in [5]. The two pass static analysis described above calculates the possible *e-c links* for

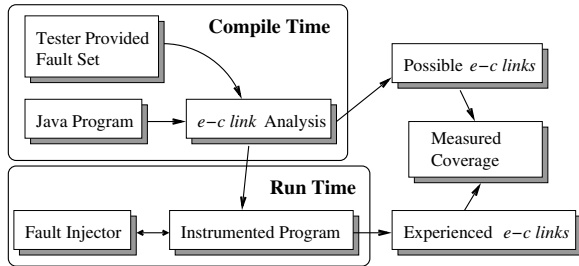


Figure 2. Exception Handler Testing Framework

a program. The dynamic 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 identify 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` 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*. In the current system,  $P$  contains all the native methods in JDK library that do network or disk I/O because i.) I/O exceptions are the most frequent and most important exceptions in web services, and ii.) the current implementation of the fault injection framework is limited. 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*.

## 2.2 Rethrow of a Caught Exception

The analysis in Section 2.1 can be used to reveal the exception propagation paths in a Java program (i.e., `throw`, `catch` pairs with chains of calls between them) with relatively high precision. Our first attempt was to build a graph out of these paths to review the overall exception handling structure of the whole system. But we found that the previous analysis cannot capture the behavior of one of the common practices in exception handling – rethrow of caught exceptions, usually in the `catch` clause.

Shenoy mentions the following as “some of the generally accepted principles of exception handling” in [11]:

1. If you can’t handle an exception, don’t catch it.
2. If you catch an exception, don’t swallow it (i.e., make no use of the exception object).
3. Catch an exception as close as possible to its source.
4. Log an exception where you catch it, unless you plan to rethrow it.

However point 1 is obviously in conflict with point 3; therefore sometimes it is better to catch an exception, add more contextual information (e.g., maybe by encapsulating the existing exception object within a new exception object)

and rethrow. Additionally, as stated in the Java JDK Library API Specification, in multi-layered systems if an operation on the upper layer fails due to a failure in the lower layer, letting the exception from the lower layer propagate outward could expose the implementation detail between layers. Doing so breaks encapsulation as well as ties the API of the upper layer to this implementation. So it is necessary to wrap the exception with a new one (i.e., in an instance of a new exception type providing a higher level of abstraction) and rethrow.

```
catch (Exception ex)
{
    throw new java.sql.SQLException(
        "Cannot connect to MySQL server: " +
        ex.getClass().getName(), "08S01");
}
```

Figure 3. Caught Exception Rethrow Example

Figure 3 shows a `catch` clause that is slightly simplified from a real one found in MySQL Connector/J 2.0.14. This `catch` clause extracts some information from the caught exception (i.e., the exception class name), constructs a new exception based on that information and rethrows it.

An exception rethrow, although desirable for various reasons, divides the exception flow from the original `throw` to the final handler into multiple segments. Existing exception-flow analyses cannot connect these closely related *e-c links* into a chain, which makes it difficult to master the exception propagation paths and handling structures of the system. A programmer trying to diagnose and repair a system degradation (or crash) may have very limited information to aid in tracing back to the root cause of a logged exception, and determining the source of the problem. If the actual exception flow is a chain spanning many software layers in the system, the testing framework in [5] is limited to exploring only individual segments of this chain.

## 3 E-c Chain Analysis

In this section we present an analysis that automatically identifies cases of exception rethrow. With this analysis, we can reconstruct the exception-flow segments into *e-c chains*, chains starting from the original `throw` and ending in the final `catch`.

**Handler-inspection analysis.** We have argued that exception rethrow is a desirable design for recovery code in modular systems. Nevertheless it adds difficulty to problem diagnosis and to the automatic inference of the exception handling structure. Because most rethrows happen inside a `catch` clause, we can design a local (i.e., intraprocedural) program analysis that examines the code inside the `catch` clause automatically, to determine whether or not the caught exception is rethrown, or a new related exception is rethrown within the `catch` clause. The basic idea is to determine how the caught exception object is used.

When the Java code shown in Figure 3 is translated to bytecode, each statement in the source code will be broken down into multiple simple bytecodes. A Java bytecode analysis tool (e.g., Soot [10]) can translated these bytecodes

```

1  r1 := @caughtexception;
2  r2 = new java.sql.SQLException;
3  r3 = new java.lang.StringBuffer;
4  r3.<init>();
5  r4 = r3.append("Cannot connect...");
6  r5 = r1.getClass();
7  r6 = r5.getName();
8  r7 = r3.append(r6);
9  r8 = r7.toString();
10 r2.<init>(r8, "08S01");
11 throw r2;

```

**Figure 4. Exception Rethrow Bytecode Representation**

into the sequence of expression statements shown in Figure 4 to facilitate further analysis and optimization. Here `@caughtexception` represents the reference to the caught exception in the `catch` clause and `<init>` signals a call to a constructor.

Each arrow shown in Figure 4 goes from a statement that defines a variable to a statement where that variable is used, that is a *def-use link*. Intraprocedural reaching-definitions [1] is a classic dataflow analysis that can produce def-use links for all the variables in a given method. By following these def-use links we can see that the statements 6 and 7 extract a string (`r6`) from the caught exception (`r1`). Then another string (`r8`) is constructed from `r6` and some other text. Finally in statement 10, `r8` is used as an argument of the constructor of another exception object (`r2`) that is rethrown in statement 11.

```

1  Initialize worklist to be empty;
2  add (ref_to_caught, pseudo_def_statement) to worklist;
3  mark (ref_to_caught, pseudo_def_statement) processed;
4  while worklist not empty
5    (ref, stmt) = worklist.remove_first();
6    use_statements = find_all_uses(ref, stmt);
7    for each statement in use_statements
8      for each def_ref in statement
9        if (def_ref is local variable)
10         if ((def_ref, statement) is not processed)
11           add statement into worklist;
12           mark (def_ref, statement) processed;
13         end for
14         if statement includes call to other method
15           and ref is used as parameter or receiver
16           label statement "Call Other Method";
17         switch kind of statement:
18         case assign statement:
19           if (assign destination is field or array reference)
20             label statement "Store into Field/Array"
21         case return statement:
22           label statement "Exception Object Returned"
23         case throw statement:
24           label statement "Rethrow"
25         end switch
26       end for
27     end while

```

**Figure 5. Handler-inspection Analysis Algorithm**

This process of variable usage tracing can be automated.

Figure 5 shows the algorithm that traces the usage of caught exceptions intraprocedurally. The algorithm takes a `catch` block, and attaches labels to some of the statements. If some statement in a `catch` block is labeled “Rethrow”, this block is considered a *interconnecting point* where two *e-c links* can be connected. The algorithm makes the following assumptions: First, the first statement of a `catch` clause is considered to be a pseudo-definition statement that initializes the reference variable pointing to the caught exception object. Second, a function `find_all_uses` is implemented that takes two parameters: a variable and a statement that defines the variable, and returns a set of statements that use that variable.<sup>2</sup> As a consequence of choosing to do a local analysis, we make conservative assumptions at method calls: the receiver and all the actual parameters are considered to be *defined* by the call statement. However, receiver and actual parameters of string and exception manipulation methods, e.g., `StringBuffer.append()`, are not considered to be defined, but only used.

In Figure 5, the loop from line 4 to 27 tries to find statements where the reference to the caught exception is used. Lines 8 to 13 say if the reference variable is used in a statement that defines another variable, keep tracing usage of the latter variable. This makes sure that we keep tracing the usage of information extracted or constructed from the caught exception, such as `r5`, `r6`, `r7` and `r8` in Figure 4. Lines 10 to 12 ensure that a statement only will be processed once, so that the main loop terminates. Lines 14 to 25 contain labeling for different kinds of statement types referring to the reference variable. For example, the algorithm reports that this handler rethrows the exception, if any of the processed statements is a `throw` statement (Line 23). Note that to keep our analysis local, the algorithm does not trace exception chains involving the reference variable being passed into another method (Line 14), or being stored into some field or array (Line 19), or being returned to the caller (Line 21). This algorithm design choice means that the analysis may miss some actual rethrows (i.e., allow false negatives).

***E-c chain construction.*** Both *Handler-inspection* analysis and the *Exception-flow* analysis in [5] are implemented in Soot, but they are not dependent on each other. *Exception-flow* analysis produces a set of *e-c links* ( $p, c$ ). At the same time the *Handler-inspection* analysis can parse all the `catch` clauses to find all the *interconnecting points* ( $c, p$ ) where  $p$  is a `throw` statement in `catch` clause  $c$  that rethrows an exception.

After obtaining both *e-c links* and *interconnecting points*, it is easy to construct *e-c chains* ( $p, c, p, c, p, c, \dots$ ) representing the propagation path of a set of exceptions resulting from single error condition. An *e-c chains* constructor is implemented that builds *e-c chains* automatically by matching

<sup>2</sup>These assumptions are satisfied by the way Java bytecode is defined and the features provided by Soot.

catch clauses and throw statements from *e-c links* and *interconnecting points*.

In our experiments we found that many of these *e-c chains* span multiple components. Thus, this analysis information can be used to illustrate exception flow between components. These can be helpful for programmers who need to understand the overall fault-handling behavior of component-based programs. During system diagnosis, more detailed information, (e.g., *e-c links*, their interconnections, the corresponding call chains) can be provided to the programmer to aid in problem localization. Since all this information is obtained using static analysis, *no run-time overhead* is imposed on the system. In addition, by extending the fault-injection testing approach in [5], the quality of the recovery code can be tested in advance of installing the web service application.

**Testing of E-c chains.** An *e-c chain* ( $p, c, p, c \dots p, c$ ) is composed of a sequence of *e-c links*. When tested, we want to make sure that at runtime, the given *e-c chain* is the propagation path of the exception thrown from the original fault-sensitive operation (i.e., the first  $p$ ). To do that, each try block corresponding to some  $c$  in any *e-c chain* is given an ID and instrumented at the entry and exit points. A thread local stack is used the keep track of try blocks that are currently *in range*. If the original fault-sensitive operation is executed, the fault injection engine can examine the stack and trigger the exception if and only if the sequence of try blocks in the stack matches the sequence of catch clauses in the given *e-c chain*. This testing framework is currently being developed.

## 4 Empirical Results

In this section we report our empirical findings and discuss a case history of Tomcat, whose goal was to demonstrate the effectiveness of our methodology. The case history demonstrates the complexity and the inter-component nature of the *e-c chains* determined by our analysis.

### 4.1 Experimental setup & benchmarks

We implemented the analysis in the Java analysis and transformation framework Soot [10] version 2.0.1, using a 2.8 GHz P-IV PC with Linux 2.6.12 and the SUN JVM 1.3.1\_08. We used five Java applications as our benchmarks:

1. Muffin, a web proxy server (<http://muffin.doit.org/>).
2. SpecJVM (<http://www.spec.org/jvm98>).
3. VMark, a Java server side performance benchmark based on *VolanoChat* – a web based chat server (<http://www.volano.com/benchmarks.html>).
4. Tomcat 3.3.1, a Java servlet server from the *Apache Software Foundation* (<http://tomcat.apache.org/>). The servlets application running on top of Tomcat is an online auction service modeled after EBay (<http://www.cs.rice.edu/CS/Systems/DynaServer/>). This application communicates with MySQL database using MySQL Connector/J.
5. HttpClient, an HTTP utility package from the *Apache Jakarta Project* (<http://jakarta.apache.org/commons/>). We

collected its unit tests to form a whole program to serve as a benchmark.

Table 1 shows the sizes of the benchmarks. Spark, a points-to analysis based call graph constructor provided with Soot[10], was used to compute the call graph of each benchmark so as to estimate the code that is *reachable* from the main function. Column 2 shows the number of user (i.e., non-JDK library) classes, with those in parentheses comprising the JDK library classes reachable from each application. The data in column 3 shows the number of reachable user methods and those in parenthesis are the JDK library methods reachable from each application. Column 4 gives the number of catch clauses in reachable user methods. The last column shows the size of the .class files (in bytes) of each benchmark, excluding the Java JDK library code.

**Table 1. Benchmarks**

Name	#Classes	Methods	Handlers	.class Size
Muffin	278(1365)	2080(7677)	270	727,118
SpecJVM	484(2161)	2489(4592)	289	2,817,687
VMark	307(2266)	1565(5029)	502	2,902,947
Tomcat	470(1869)	2964(8197)	502	4,362,246
HttpClient	252(2210)	1334(4741)	536	1,049,784

VMark, Tomcat and HttpClient are composed of many components, identified by multiple jar files in the distribution.<sup>3</sup> We have Java source code for all the benchmarks except SpecJVM and VMark. Only part of the source code for SpecJVM is provided and there is no source code for VMark. Although we can conduct our experiments using only bytecode, the unavailability of source code hindered the process of interpreting our experimental results.

On each benchmark, the *Handler-inspection* analysis finished in under 2 minutes and *e-c chain* construction took even less time. This total analysis cost is negligible comparing to the running time of the *Exception-flow* analysis we are using – about 1 hour for most benchmarks used in [5]. (Recall this analysis does not execute at runtime.)

### 4.2 Empirical Data

As mentioned before, the *Handler-inspection* analysis automatically examines all the catch clauses to find out how the caught exceptions and information derived from them are used. We can categorize each exception handler based on the information obtained, partitioning them into the following categories: the caught exception (or information derived from it) is (i) rethrown, (ii) stored into a field/array, (iii) returned to caller, (iv) ignored, or (v) the catch clause is completely empty, or (vi) other cases.

Figure 6 shows the percentage breakdown of the reachable handlers in each of the benchmarks according to the above categorization. As we can see from the chart, in 4 out of 5

<sup>3</sup>We assume that the user of our analysis knows the boundary of components. Here we assume one component per jar file of each benchmark; this assumption can be overridden by specifying the component membership of classes according to a provided XML schema. There is no jar file defined in Muffin or SpecJVM.

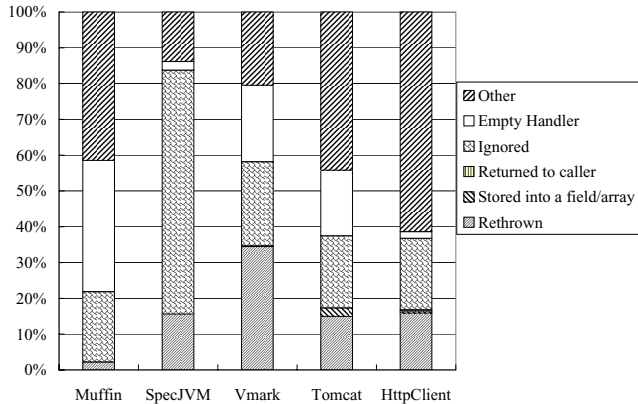


Figure 6. Usage of Caught Exceptions in catch Clauses

benchmarks, the percentage of handlers that rethrow exceptions ranges from 15% to 35%, something that we *cannot* ignore. Empty catch clauses occur significantly often in all of the benchmarks. There is also a significant percentage of non-empty catch clauses in which caught exception objects are ignored. It is very rare that exception objects are stored into some field/array or returned to the caller.

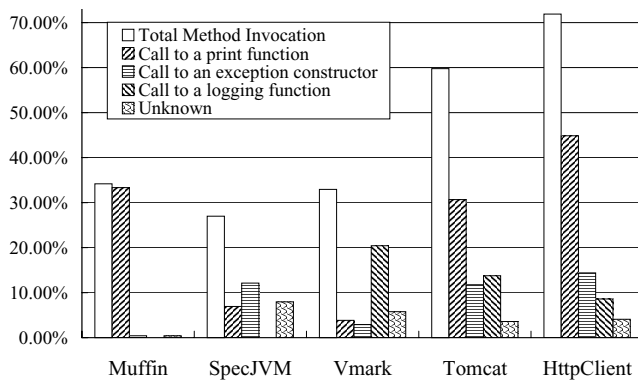


Figure 7. Methods Calls Related to Caught Exception

All of the handlers in category (vi) contain invocations of other methods with information from the original exception used as the receiver or a parameter. Some handlers in category (i), (ii) and (iii) also may make such method calls. Figure 7 shows the kinds of method calls that appear in these handlers. The height of each bar represents the number of catch clauses in each category, normalized by the total number of reachable handlers in the benchmark. Most of the time our analysis can automatically identify the call targets as either a constructor of another exception, a printing function in the Java library, or an application-specific logging function, (in order to discover the last case, information for each benchmark must be manually specified before the analysis). Only a relatively small number of them are some other exception handling method in the application.

From the data presented above we can see that *Handler-inspection* analysis can summarize the behavior of the catch clauses. This information, when combined with *e-c chains* discovered in the system, can help a programmer pay more attention to the important catch clauses with undesirable

behaviors (e.g., swallowing exceptions). We believe that catch clauses that can be reached by many different exception sources are more important, because defects in them may be more harmful.

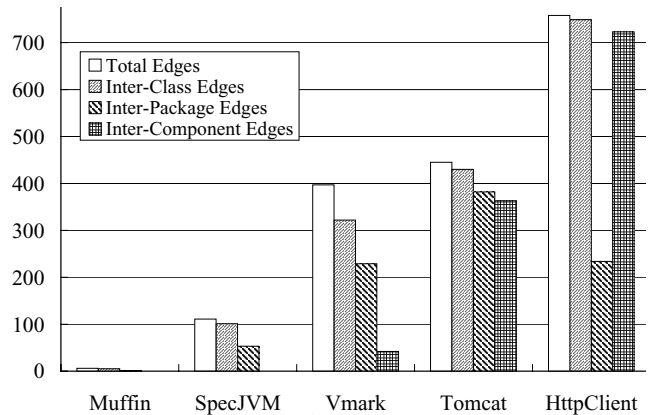


Figure 8. Number of e-c links Starting from a Rethrow

After *Handler-inspection* analysis, *interconnecting points* can be identified among the catch clauses (i.e., a catch clause  $c$  containing one statement labeled “rethrow”  $p$ ). We would like to know the possible destinations of the rethrown exceptions in these handlers. So we examine all the *e-c links*  $(p, c)$  that start from one of the *interconnecting points*  $(c, p)$ . Figure 8 shows numbers of these *e-c links* in which the source and target of the *e-c link* belong to different classes, packages or components. In all the benchmarks (except Muffin), as expected the majority of these *e-c links* propagate across components or package boundaries. This information is of great value in discovering and understanding the interaction between components, and revealing the high-level recovery structure of the system. In systems of this complexity, it is hard to determine this merely by manual inspection.

In *HttpClient* there are many more *e-c links* across components than across packages. The reason is that we are using its unit tests to form a whole program (necessary for our analysis). Unit tests are packed in a different component (i.e., jar file) from the main implementation, but both are included in the same package; in all the other benchmarks, each component consists of one or more packages not vice-versa. The large number of *e-c links* between the implementation and the test components shows that the methods under test often pass along exceptions back instead of handling them locally.

Table 2. Number of Chains of Difference Length

Length	1	2	3	4	5	6	Total
<i>Muffin</i>	6						6
<i>SpecJVM</i>	69	46					115
<i>VMark</i>	300	81	12				393
<i>Tomcat</i>	312	365	31	3	2	10	723
<i>HtpClient</i>	583	547	275				1405

Finally, the *e-c chain* constructor can connect the *e-c links* gathered with their identified *interconnecting points* to form

*e-c chains*. Table 2 lists the distribution of *e-c chains* of different lengths in each of the benchmarks. Note that since these *e-c chains* are constructed from *e-c links* that start from some *interconnecting point*, each one shows an exception propagation path with the first segment missing. The reason we are showing the data this way – instead of starting from the original *throw* – is that some of the *interconnecting catch* clauses are *protective* handlers that usually can only be reached by *unchecked* exceptions (e.g., `NullPointerException` or `ThreadDeath`). These handlers are used to prevent the malfunctioning of some component that may bring down the system, but the *e-c links* reaching them are either very hard to find or do not exist explicitly in the code. So we ignore the first segment of each *e-c chain* in order to gather and report uniform data. Of course, the *e-c chain* constructor provides the whole path for examination, when the first segment involves a checked exception.

As can be seen from Table 2, 4 out of 5 benchmarks show a significant portion of the *e-c chains* have length greater than 1. Since these are *e-c chains* with the first segment missing, we can see that in many cases, one exception can go as far as 2 “hops” before reaching its final handler. There are surprisingly long *e-c chains* found in Tomcat, which shows the complex exception handling of the system. Clearly, this data is sensitive to the way in which we count *e-c chains* that share *intersecting points*. Here, we count all possible combinations of incoming *e-c links* with outgoing *e-c links*. For example, suppose a single *interconnecting point* has two incoming *e-c links* and two outgoing ones, forming an X shape; the number of *e-c chains* will be 4.

In all the benchmarks except Muffin, exception rethrow is common and with the *Handler-inspection* analysis, we can automatically identify semantic relations between individual *e-c links* caused by this phenomenon. As often these paths go across different components, a programmer can better understand the interactions between components caused by the application recovery code, with the help of this information. Next, we will show how to use this information to create a higher level view of exception-handling architecture in the *e-c chain* graph.

### 4.3 E-C Chains in Tomcat

The data presented above, especially the long *e-c chains* found in Table 2, drew our attention to Tomcat. So we manually inspected its *e-c chains* and source code, hoping to find answers to the following questions: *How precisely does the analysis identify interconnection points? What can these e-c chains tell us about the overall exception-handling behavior of the system?*

**Precision.** We are primarily interested the precision of recognizing *interconnection points* in all the *catch* clauses. As mentioned in Section 5, the *Handler-inspection* analysis can report false positives because it is approximate. Also, the analysis does not examine called methods in a *catch* clause, even if the exception is passed into them. There may be cases

where the callee takes some exception and throws it or constructs a new exception from it and throws that exception. In such cases, the exception thrown in the callee is directly or indirectly related to the caught exception in the caller. The corresponding *catch* clause should be recognized as an *interconnecting point*, but the analysis does not do so; this case is a *false negative*.

To check the number of false positive and false negative cases, we manually inspected all the *catch* clauses in Tomcat to verify the results of the automatic *Handler-inspection* analysis. Surprisingly, *we did not find any false positives*; that is, all the *interconnecting points* found, actually throw some exception that is either directly or indirectly related to the original caught exception! Unfortunately, we did identify 3 cases of false negatives. There are 2 *catch* clauses in the Apache Crimson package, which call the same method that constructs a new exception out of the caught one and then throws it. Another *catch* clause in the Tomcat Facade package calls a method which throws its parameter directly.

We also may introduce false positives as we form *e-c chains* from the results of the *Handler-inspection* analysis. When we connect multiple *e-c links* into a *e-c chain*, the call path associated with the chain maybe infeasible, although the call paths associated with each *e-c link* are feasible. This may occur, for example, if two exception objects are handled in one *interconnecting point* and the rethrow target is determined by the object thrown. Thus, there may appear to be two possible handler targets, but only one corresponds to each incoming exception object. We were unable to verify that this problem did not occur in Tomcat, since to manually figure out call chain feasibility in a large object-oriented system is not straight-forward. However, the situation, should it occur, can be partially alleviated by applying the *DataReach* analysis from [5] to remove *e-c chains* only associated with infeasible call paths.

The existence of some false negatives in our analysis is not unexpected. To avoid false negatives would require a much more precise interprocedural analysis that would be very costly, and itself might introduce additional false positives. Thus, we chose to implement an analysis of practical cost, which identifies, we believe, the bulk of the *e-c chains* of interest. Given the complexity of exception handling in Tomcat and the results of our manual inspection, we feel this decision is justified.

**E-c chain Graph.** The *e-c chains* can be depicted in a graph and shown in differing granularity to help in different tasks. When shown at the component level, *e-c chains* illustrate the interactions between components of the system, which helps a user understand the high level error recovery architecture of the system. Critical components (i.e., those either handle or issue many exceptions with different source/sink) can be located for testing or inspection.

In system diagnosis tasks, first the programmer can obtain the immediate cause of the symptom from the system

log. Displaying *e-c chains* may help the programmer decide which of the components are involved and what are the possible root causes. Then, detailed information such as the position of `throws` and `catchs` in the code and call paths between them, can be shown to help with detailed reasoning.

We manually collected all the *e-c chains* with length greater than 2 and display them in Figure 9, which shows the exception-flow architecture of the system. This process can be automated using graph drawing packages such as Graphviz (<http://www.graphviz.com>).

By looking at the *e-c chain* graph in Figure 9, we can easily make two observations. First, on the left-hand-side of the graph, MySQL Connector/J, which needs to do I/O in order to communicate with the MySQL database, propagates exceptions first to DynaServer, then to the Tomcat Facade component. So if the network connection to the database goes down when the system is running, it may cause problems in the servlet application, but components in Tomcat, other than the Tomcat Facade, are very likely not to be affected. Second, according to the right-hand-side of the graph, the Tomcat Util component is the only one that does I/O operations directly, including parsing XML by calling the Crimson component. In this sense, the Tomcat Facade and the Tomcat Util components serve as firewall between other parts of Tomcat and outside environment, thus are identified as critical components in error recovery. If these components are well tested to handle/log exceptions properly, the whole system will likely to handle/log errors properly.

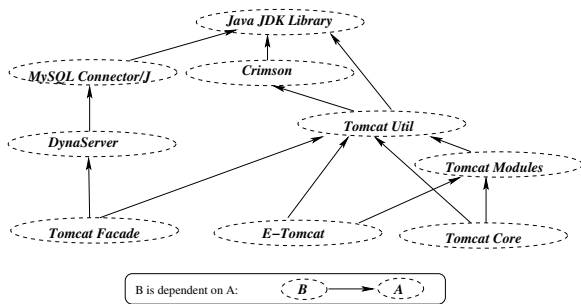


Figure 10. Service Dependence Graph of Tomcat

The *e-c chain* graph can also be presented in a coarser granularity to reveal dependences between components, and thus forms a *service dependence graph*: When an exception flows from component A to component B, we can see that an operation failure in A may cause an operation failure in B. In another words, some operation in B is dependent on the service in A to complete its functionality. Figure 10 is the service dependence graph of Tomcat.

The *e-c chains*, when depicted in the graphs in Figure 9 and 10, can show the exception-handling architecture of Tomcat in a compact form. By inspecting the graphs, a programmer can understand the exception-handling interaction between major components, and at the same time, estimate the vulnerability of certain components as well as that of the whole system. A person trying to gain knowledge about

possible root causes of a particular problem can browse the exception propagation path and participating components on these graphs. All this knowledge can be obtained by examining the graphs showed above without consulting the source code of the system.

## 5 Related Work

There has been much previous research in static and dynamic analyses to discover exception flows in programs and to categorize and evaluate exception handlers. In this section, we will discuss only the most relevant research results in each of these areas.

### 5.1 Static Exception-Flow Analysis

There are several existing static exception-flow analyses for Java that vary in their precision. Their basic idea is similar: An operation that can throw a particular exception is treated as a source of an abstract object that is propagated along reverse control-flow paths to possible handlers (i.e., `catch` blocks), and thus exception-flow links are discovered. Due to the common interprocedural nature of exception handling, much of this propagation happens along call graph edges, in the reverse direction of execution flow. Thus, how interprocedural control-flow is approximated determines the precision of these techniques.

Jo *et. al* [6] present an interprocedural set-based exception-flow analysis; only checked exceptions are analyzed. Experiments show that this is more accurate than an intraprocedural *javac*-style analysis on a set of benchmarks five of which contain more than 1000 methods. Robillard *et. al* [8] describe a dataflow analysis that propagates both checked and unchecked exception types interprocedurally. Each of these techniques handles a large subset of the Java language, but makes the choice to omit or approximate some constructs (e.g., *static initializers*, *finallys*). These analyses use class hierarchy analysis to construct call graphs that are therefore very imprecise [4, 2].

Sinha *et. al* [12] present an interprocedural program representation which more accurately embeds the possible intraprocedural control flow through exception constructs (i.e., `try`s, `catch`s and `finally`s). Class hierarchy analysis (CHA) 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.

Fu *et. al* [5] build their exception-flow analysis parameterized by the choice of call graph constructor: CHA, rapid type analysis (RTA) [2], or points-to analysis (PTA) [9]. Experiments show that more than 85% of the false positive exception-flow links found in the relatively large benchmarks when CHA is used can be removed by simply switching to the PTA call graph constructor. Fu *et. al* also proposed a schema of filtering algorithms that further reduces the number of false positives by around 50% in their benchmarks. A



framework for def-use testing of exception handling is defined based on the above analysis.

**Limitations.** Unfortunately, although all of these static analysis identify individual exception-flow links, none of them discover the possible semantic relations between these links, induced by shared exception objects or exception data. These semantic relations are the focus of our analysis presented here.

## 5.2 Dynamic Exception-Flow Analysis

A dynamic analysis of exception-flow is presented by Candea *et. al* [3]. This approach discovers exceptions propagated across the boundaries of components (i.e., bean/servlet/JSP). For each method of a newly loaded component, the analysis parses the `throws` clause in the method declaration to obtain the set of all the exception types that may be thrown by that method, plus possible unchecked exception types. Each time the method is invoked, a new exception type from the set is picked and thrown. If that exception causes failure of some other component, an edge from the exception throwing component to the failed component is added to a graph known as a *failure map* that tracks inter-component exception-flow.

**Limitations.** Often the exception types listed in the `throws` clause of a Java method are actually supertypes (or supersets) of what can be thrown (e.g., due to subsumption). Moreover, a method declaring that it throws some type of exception is very likely to be just a propagator of the exception, rather than the origin of the `throw`. Exception-flow links derived using this technique may be imprecise, despite of the analysis' dynamic nature, and also incomplete (e.g., missing the chain origin).

## 5.3 Catch Clause Categorization

Reimer and Srinivasan [7] present a list of actual exception usage issues observed in large J2EE applications that that have hindered the maintainability and serviceability of these applications. These issues include swallowed exceptions, using a single `catch` for multiple exceptions, and placing a handler too far away from the source of the exception. Unfortunately, the underlying analysis is not discussed in the paper. Data tables show that they did not find any handler with exception rethrows, a finding in conflict with our empirical data (see Section 4).

## 6 Conclusion and Future Works

We have defined a static *Handler-inspection* analysis that examines reachable `catch` clauses to identify `catch` clauses that rethrow exceptions. Our *Exception-chain* analysis combines this information with *e-c links* found by an existing static analysis, forming *e-c chains* at compile time without any runtime overhead. A graph of these *e-c chains* depicts the architecture of system recovery code at several levels of granularity: component, package, class. We believe that this graph and its related service dependence graph that highlights exception flow between components, are valuable for system problem diagnosis and program understanding tasks.

Our future plans include building a GUI to display the *e-c chains* to allow interactive browsing on many levels of granularity. We are extending the instrumentation algorithm of the testing framework in [5] to accommodate both *e-c links* and *e-c chains* for better error recovery code testing.

## References

- [1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers – Principles, Techniques and Tools*. Addison Wesley, 1988.
- [2] D. Bacon and P. Sweeney. Fast static analysis of C++ virtual functions calls. In *Proceedings of ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages and Applications (OOPSLA'96)*, Oct. 1996.
- [3] G. Candea, M. Delgado, M. Chen, and A. Fox. Automatic failure-path inference: A generic introspection technique for internet applications. In *WIAPP '03: Proceedings of the The Third IEEE Workshop on Internet Applications*, page 132, Washington, DC, USA, 2003. IEEE Computer Society.
- [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):292–311, Apr. 2005.
- [6] J.-W. Jo, B.-M. Chang, K. Yi, and K.-M. Choe. An uncaught exception analysis for java. *Journal of Systems and Software*, 72(1):59–69, 2004.
- [7] 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.
- [8] 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.
- [9] A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for java using annotated constraints. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 43–55, Tampa Bay, FL, USA, 2001.
- [10] Sable, McGill. Soot: a java optimization framework, 2003. Available at <http://www.sable.mcgill.ca/soot>.
- [11] S. Shenoy. Best practices in EJB exception handling. IBM developerWorks Artical, May 2002. Available at <http://www-128.ibm.com/developerworks/library/j-ejbexcept.html>.
- [12] 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.