IMPROVING SOFTWARE RELIABILITY USING EXCEPTION ANALYSIS OF OBJECT ORIENTED PROGRAMS

BY CHEN FU

A dissertation submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Graduate Program in Computer Science

Written under the direction of

Barbara Gershon Ryder

and approved by

_____

_____

_____

_____

New Brunswick, New Jersey

January, 2008

ABSTRACT OF THE DISSERTATION

Improving Software Reliability Using Exception Analysis of Object Oriented Programs

by Chen Fu

Dissertation Director: Barbara Gershon Ryder

More applications are designed as server programs, many of which are expected to run 24x7. Ensuring the quality of error handling code is vital to the high availability that are expected from them. However, error handling code is hard to explore, review and test, for the reason that 1) it is scattered all over the system, often not at all organized; 2) it is impossible to trigger during runtime by simply manipulating the program inputs or configurations.

The goal of our research is to provide tools that helps programmers explore, review and test error handling code in a structured way to boost the system availability and maintainability.

The contributions of this thesis are the following:

- Definition of the problem of white box robustness testing for Java-based server applications, including an exception *def-catch* coverage metric and testing framework.

- A new program analysis that enables the above mentioned testing methodology, which 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.

- Empirical studies of several variants of the analysis algorithms, which demonstrate increased precision in obtaining good test coverage on a set of server benchmarks. These studies include aggregate accuracy and timing information, with discussions of cases in which static analysis is difficult.

- A program understanding tool that visualizes discovered *exception-flow 'def-use'* links.

- A novel program analysis that discovers semantic relations between the *exception-flow 'def-uses'* links and combines them into chains, in order to reveal the propagation path of an exception from its original to its final handler.

- An initial case study of testing exception propagation chains.

Acknowledgements

I have been fortunate enough to have had the support of so many talented people, without which this thesis would not have been possible.

First and foremost, I have to thank Dr. Barbara G. Ryder for being a teacher, colleague as well as friend of mine. It is impossible to overstate my thankfulness to her for her continuous encouragement, patient guidance and sharp criticism.

I wish to express my sincere thanks to Dr. Xiaoxia Ren, for being a caring wife and also a helping colleague, for giving ideas, suggestions and support in those struggling moments.

I owe my gratitude to Dr. Thu Nguyen, Dr. Richard Martin, Dr. David G. Wonnacott, Dr. Ana Milanova, and Dr. Kiran Nagaraja, for their cooperation and support during the course of my Ph.D. study.

I must also thank current and previous members of PROLANGS Lab, including Weilei Zhang, Ophelia Chesley and Bruno Dufour, for being my friends and providing the most inspiring and enjoyable working environment.

I am also grateful to Dr. Phyllis G. Frankl and other committee members for reviewing my thesis draft and constructive suggestions.

Many thanks to professors and staff members in the Computer Science Department of Rutgers University, for their inspiring teaching and professional support during my study in Rutgers.

A great Thanks to all!

Dedication

To Sophia and Arianna, two best girls that anyone could possibly want.

Table of Contents

List of Tables

List of Figures

Chapter 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 server applications (typically accessible over the web). These applications must meet the challenges of maintaining performance and availability, while supporting large numbers of users, who demand reliability from these programs that are becoming more and more commonplace.

However these servers often fail to meet such high availability expectations due to the following reasons: 1) Hardware platforms on which these services often run are heterogeneous hardware clusters, which themselves are complex. 2) Software used in these services suffers from short life cycle caused by market pressure, which emphasizes increasing the number of features instead of building more robust systems.

Adding redundancy works great in improving system availability when there are only non-deterministic and low probability problems present. For deterministic or high probability software bugs, it does not necessarily solve the problem. On June 4th 1996, the Ariane 5 launcher veered off its flight path, broke up and exploded less than a minute into its first flight. Later investigation revealed that a software bug caused the guidance computers to shut down despite the fact that there were two independent replicas. This bug finally resulted in the starting of the launcher's self-destruction system, causing the loss of more than US$370 million [4].

The quality of exception handling code in these complex systems is vital to their overall availability because that the exception handling code actually defines the system's behavior under sub-system or infrastructure errors. It should be responsible for recovering from transient problems, automatically selecting alternative resources if

available, and logging enough error information for future investigation. The first two tasks prolong system *up* time and the third one reduces system *down* time. More importantly, exception handling code should be correctly placed to prevent small problems or sub-system errors from catastrophically bringing down the whole system, resulting rather in performance degradation or partially losing of functionality instead of system crash. In the Ariane 5 failure, an integer overflow in the guidance computers caused an exception to be thrown because the speed of Ariane 5 launcher was faster than that of the Ariane 4, for which the software was originally developed. This exception was not specifically handled and the default exception-handling mechanism was so defined just to shut-down the guidance computer[4]. Either a correctly placed exception handler for this operation, or a more sophisticated default exception-handling function could have helped in avoiding the failure.

This thesis discuss a series of analysis algorithms and tools to facilitate testing as well as understanding, inspection of the exception handling code, so as to increase system availability. Program analyses are designed to locate exception propagation paths in a given Java program. Our testing framework and tools can use this information to direct fault injection testing and help in exception handling code navigation. We devoted much effort in improving the precision of the analysis to reduce the time spent on inspecting and testing false paths[1] in both testing and code inspection.

1.1 Testing of Exception Handling Code

The robustness testing research in this thesis addresses the problem of how to test the reliability of server applications written in Java, in the face of infrequent but anticipatable system problems that the program usually responds to via Java's exception handling mechanism.

Traditional reliability testing of software in the dependability 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

[1]Infeasible paths reported by the analysis as feasible because of analysis imprecision.

workloads [5]. Testing is accomplished by simulating faults caused by environmental errors during test through *fault-injection* [18, 19, 30, 34, 61]. 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 reliable service (i.e., not crash) under the given fault loads and workloads [5].

Unfortunately, this approach does not ensure that the exception handling 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 exception handling code during further testing.

There is also a large body of existing work on *white-box* testing methodologies [10, 45, 29], 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, exception handling code – code which handles errors that occur with small probability, especially due to interactions with the computing 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 executed by merely manipulating program inputs or even configurations.

We developed *exception-catch link* analysis that can identify exception propagation paths: from where an exception may be thrown (i.e. a *def*) to where it may be handled (i.e. a *use*) in a given Java program. This allows compiler-inserted instrumentation 1) to inject appropriate faults to trigger the exception, and 2) to gather recovery code coverage information. Now a tester can systematically exercise the error recovery code, by causing normal program execution into these paths (normal execution is in the reverse direction of the exception propagation paths[39]). Thus the methodology provides a means to obtain validation of application robustness in the presence of system faults. Although our experiments are based on server applications, the technique can be applied to general Java applications.

1.2 Exception Handling Code Understanding

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

With the exception propagation paths of a given program provided by *exception-catch link* analysis, the following questions can be answered: What are the kinds of exceptions and/or the set of `throw` statements that can reach a given program point? Where are all the handlers for this particular (kind of) exception(s)? We developed a program visualization tool that carefully presents *exception-catch link* analysis results to facilitate answering of these two questions, which can help a programmer navigate exception handling code that related to certain kind of problems.

Furthermore, in component-based systems, exception flow spanning different components often manifests as *chains* of exception `throw`s and `catch`s, instead of a single exception-flow link. Although individual exception-flow links can be obtained with relatively high precision using *exception-catch link* analysis, 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.

Our *Exception-chain* analysis captures this behavior and provides complete exception flow information for interesting exceptions. The result greatly facilitates understanding exception handling architecture of a given program.

1.3 Contributions

We present our advances over the current state-of-the-art below; some previous disclosure of these contributions have appeared in our publications [23, 24, 25, 26].

- Definition of a *white box* coverage metric for testing exception handling code in Java applications. Design of a testing framework for automatic measurement

of the coverage metric. Demonstration of *automatic* program instrumentation directed by compile-time analysis, that effectively constructs a compiler-directed fault injection engine from *Mendosus* [38], an existing fault injection framework.

- Design of a set of new compile-time analyses to identify exception propagation paths in a given program with high precision, including:

  1. The *exception-flow* analysis that identifies exception handling code in relation to certain resource usage program points (i.e., a def-use analysis for potential exceptions involving resource usage).
  2. The *DataReach* analysis which uses the absence of data reachability through object references to confirm the *infeasibility* of some interprocedural program paths.
  3. We also reformulate the *DataReach* analysis as a general schema that can be instantiated to yield different algorithms by varying the number of distinct sets of visible objects (as in the work of Tip and Palsberg [75]). Several new variants of the *DataReach* analysis schema (which we call *C-DataReach*, *M-DataReach*, and *V-DataReach*) have been defined and explored. This exploration compares the relative accuracies and computational complexities of these four variants of our analysis. Experiments show that *DataReach* analyses can effectively reduce the number of false positives produces by the above mentioned *exception-catch link* analysis.

- Experimentation with coverage testing of exception handling code on a set of Java server applications, using several variants of our DataReach algorithm, and several variants of the earlier stages of our analysis. These studies include aggregate accuracy and timing information, as well as specific discussions of the cases in which static analysis is difficult.

- Design and development of a program visualization tool. It groups together handlers that handle exceptions triggered by a set of fault-sensitive operations[2]; and

[2]A fault-sensitive operation is a `throw` statement, or a native method that may be affected by some

thus facilitates navigation of the program code that relates to exceptions triggered by certain operations of interest. It also shows all program paths via which these operations can be reached from some call site in the `try` block, helping a user to understand the exception handling structure, and to identify spurious exception def-uses.

- A new `catch` clause inspection algorithm that traces the usage of the caught exception so as to categorize `catch` clauses to help programmers concentrate on potentially problematic ones. It is extended to form a new exception *chaining* analysis that can identify semantic relations between *exception-catch links* and combine them into chains to reveal entire propagation paths of exceptions.

- Empirical studies of the use of exception *chaining* analysis in exception handling code inspection and testing. An initial case study shows that the compiler directed fault-injection testing framework can be effectively extended to test entire exception propagation paths. Also a graphical representation of these chains at different *zooming* levels helps a programmer gain knowledge of the exception handling structure of the program without diving into the source code.

1.4 Thesis Outline

The rest of this thesis is organized as follows. In Chapter 2 we introduce our compiler directed fault-injection testing framework, including background knowledge and definition of terms that will be used later. Then *Exception-flow* and *DataReach* analyses will be discussed in Chapter 3, followed by the empirical results of the testing framework on a set of benchmarks in Chapter 4. Chapter 5 presents the program visualization tool that helps navigating exception handling code. Chapter 6 reveals how semantically related *exception-catch links* are connected into chains. Chapter 7 shows experimental results for exception chain analyses. Chapter 8 and Chapter 9 talk about related work and conclusion, respectively.

---

fault – a hardware or OS failure – and produce an exception.

Chapter 2

Compiler Directed Fault Injection Testing

In this chapter we first introduce the problem of testing for robustness. Then we present our compiler directed fault-injection testing framework for white box def-use testing of exception handling code in Java programs. Some of these discussions appeared earlier in publications [23, 24].

2.1 Background

Before giving our methodology for coverage testing exception handling code in Java programs, we first review prior uses of the term coverage and discuss the relation between operating system/hardware faults, Java exceptions, and exception handlers in the application.

2.1.1 Why Java

Our system is implemented to work for Java-based server systems. We choose Java-based systems for the following reasons: First, unlike C or C++ where the programming convention often overloads the return mechanism to describe errors, Java contains well-defined program-level constructs, exceptions, that respond to error conditions [6]. This facilitates both the construction and analysis of error recovery. Second, Java is used increasingly in building large-scale server applications. Third, the platform independence of Java, its portable program representation (i.e., bytecode), and its standardized JDK libraries all facilitate software reuse via COTS components.

The exception handling mechanisms provided by Java and C# are similar. The differences include different syntax rules and also the fact that C# does not support *checked* exceptions while Java does [41]. We believe that the techniques discussed in

this thesis are also applicable in C# with minor modifications in implementation detail, because our analysis algorithms will not be affected by these differences.

The difference concerning *checked* exception essentially means the following: A Java program which may experience fault-induced[1] errors cannot be written without inclusion of some exception handling code that will be triggered upon these errors, but it is possible to do so in C#. Thus it is more likely in C# than in Java that some potentially exceptional operation was left unprotected which may result in system failure. For this reason we believe that our techniques would be at least equally, if not more valuable for C# programs.

2.1.2 Exception Handling in Java

The Java programming language provides a program-level exception handling mechanism in response to error conditions that happen during program execution.

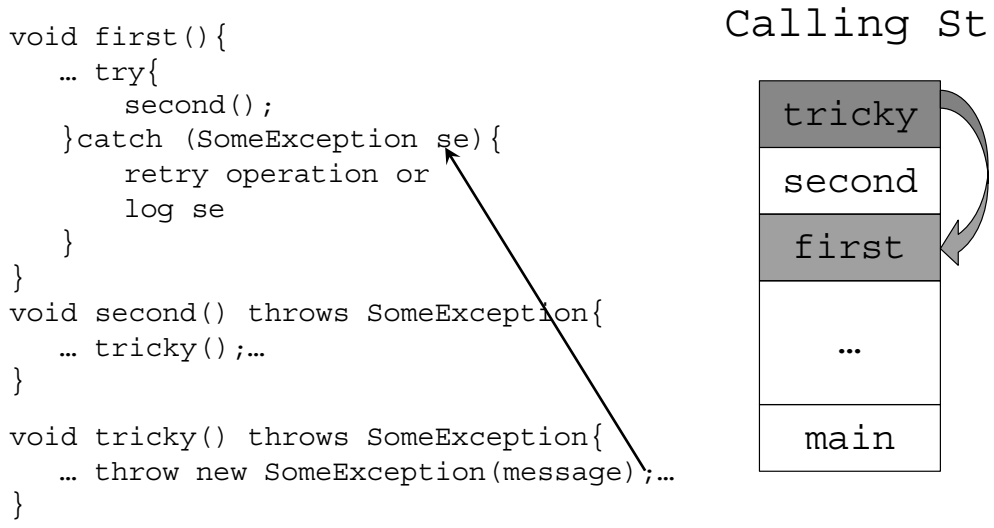


Figure 2.1: Exception Handling In Java

The programmer can signal the transition from *normal* state to *exceptional* state by throwing an exception, as the example code shown in Figure 2.1, in method `tricky`. The method can choose to handle the exception locally, or rely on its caller to handle it. To handle an exception, as shown in method `first`, the potentially exception-throwing call

[1]In this thesis "fault" means environmental errors including infrastructure or sub-system problems, e.g. disk crash, network partition, OS bug, instead of a software bug in the program under investigation.

site needs to be enclosed in a `try` block followed by some `catch` clauses, each specifying the type of the exception that it is intended to handle together with code that actually handles the exception.

At runtime, as shown in the right half of Figure 2.1, when an exception is thrown, the Java VM will inspect the call stack from top to bottom looking for an exception handler with the right exception type. Once an exception handler is located, a transfer of control happens in the direction of the arrow in the figure, and top portion of the run-time stack (those method invocation above method `first`) is discarded.

There are two kinds of exceptions in Java, *checked* and *unchecked* exceptions. If a method itself or its callee can potentially throw an *checked* exception but does not handle the exception, it must explicitly specify this fact by a `throws` clause in the method signature with the types of exceptions that could be thrown by calling this method (see methods `second` and `tricky` in Figure 2.1). Most user-defined exceptions as well as exceptions caused by the Java runtime system are *checked* exceptions. If a method can potentially throw an *unchecked* exception, it does not need to specify this fact in its method signature. Examples of *unchecked* exceptions include `NullPointerException`, `RuntimeException`, etc.. Note that although exception handling mechanisms in Java and C# are similar, one major difference is that all the exceptions in C# are *unchecked*.

There is another way to categorize exceptions in Java. They can be divided into *synchronized* exceptions and *unsynchronized* ones. *Synchronized* exceptions have explicit throwing points, either a explicit `throw` statement, or potential exceptional operations that upon execution, may throw certain types of exceptions (e.g., a call to a native method that could throw an exception, or dereference of a null reference, as in `NullPointerException`). Thus a call stack can be saved for later inspection when the exception is created, usually at the place where it is thrown. But *unsynchronized* exceptions may not have an explicit throwing point at runtime. One example of an *unsynchronized* exception is `OutOfMemoryException`.

The techniques discussed in this thesis try to locate an exception propagation path from its origin (i.e., where it is thrown) to its destination (i.e., where it is handled). Thus we can not deal with *unsynchronized* exceptions for which the origins are not

clearly defined. So in later part of this thesis, we use the term *exception* to refer only to *synchronized exception*. However, since our techniques do not rely on the `throws` clause in method signatures, we can trace both *checked* and *unchecked* exceptions.

2.1.3 Definitions of Coverage

Both the dependability and software engineering communities have precise definitions for the term *coverage*; however, they use this term in very different ways. In the dependability context, coverage is defined as the conditional probability that the system properly processes a fault, given that the specific fault occurs [12]. Later work included the assumption that the fault was *activated* in the probabilistic definition [5]. A number of modeling and analysis strategies naturally arise from this definition. First, coverage can be mathematically represented as *probability density* and *cumulative density functions* (PDF and CDFs). Second, these functions can be transformed into probability density over time and cumulative density over time, leading to a range of analyses using stochastic process models (e.g., [21]). These models can describe the impact that coverage has on the expected time to enter a failure state under a given fault load, and the amount of redundancy necessary to achieve targeted levels of availability and performance.

By contrast, the software engineering community uses a fundamentally different definition of coverage. In this context, coverage is defined as the fraction of the application that has been exercised by a given test in terms of specific programming constructs including statements and branches. For example, *all-branch* coverage ensures that every branch in a program (e.g., exits from an `if` statement) is traversed at least once during testing. Similarly, *all-statement* coverage guarantees that every statement in the program has been executed at least once during testing. Another set of constructs, based on dataflow, traces values from their definition point to their subsequent usage, that is, *def-use* coverage [52]. A hierarchy of def-use coverage metrics has been defined, among which, the *all-defs* coverage metric requires that tests cover one path between each value-setting operation and a use of that value [52]; this is to ensure that errors due to incorrect flow of data values are handled properly.

For the remainder of this thesis, we call the definition based on conditional probability *fault coverage* and the software engineering definition, *program coverage*. One of our primary goals in this work is to define a metric for program coverage as it relates to exception handling code that describes the coverage of combinations of recovery-code blocks and fault types, not the fraction of actual faults that were handled.

2.1.4 Fault Injection

Also note that just as the term *coverage*, *fault* and *fault injection* are also used by both the dependability and software engineering communities, with different definitions. In the software engineering community, *fault* is usually used to refer to a software bug. *Fault Injection* means "planting" bugs into the existing *correct* programs by code mutation, with the hope that these code errors are very similar to those unintentionally added by programmers, in order to demonstrate the effectiveness of fault localization and/or testing techniques.

In the dependability community, however, a *fault* indicates an environmental problem or a sub-system error that manifests itself. *Fault Injection* is the effort to introduce or simulate faults into an otherwise healthy system. This latter definition of *fault injection* is used in the remainder of this thesis.

Experimental evaluation by fault injection has long been used for estimation of fault tolerant system measures such as *fault coverage* [18, 19, 30, 34, 61]. There are both hardware and software implementations of fault injection. Hardware implemented fault injection can vary from hardware being exposed to heavy-ion radiation, to pulling the power plug manually; thus these approaches either depend on complex and expensive special hardware, or are performed in an ad hoc manner.

Software implemented fault injection, sometime referred to as *fault emulation*, is preferable to other fault injection alternatives because it is easy to implement and modify when requirements change, and it is easy to control and observe the faults injected. Software implemented fault injection can take many forms. In the testing of operating systems, for example, fault injection is often performed by a driver (kernel-mode software) that intercepts system calls (i.e., calls into the kernel) and randomly

returning a failure for some of the calls. This type of fault injection is also useful for testing low level user mode software (e.g., an X server). In managed code, on the other hand, it is common to use instrumentation.

2.2 Exception Def-Use Coverage Testing

Here we present the definition of coverage for exception handling code of Java programs, and introduction of our testing framework. We also provide an in depth discussion about the program analysis algorithm needed in this framework in measuring and even help improving the coverage.

2.2.1 Faults, Exceptions, Coverage Metric

A *fault* is some environmental error that manifests itself. 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*, either an explicit `throw` statement or a call to unknown method, is *affected* by a fault if when the operation occurs, it experiences a fault as a run-time error and an exception is produced. 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 of 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 thesis 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. [2] 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 [52] of traditional def-use analysis:

Definition (*e-c link*): Given a set P of fault-sensitive operations that may produce

[2] There is a many-to-many relationship between system faults and Java exceptions. For this paper we assume that the tester merely has to choose one or more exceptions of interest. For more details, see [23].

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 $\frac{|E|}{|F|}$.

It is not hard to locate E in the above definition starting from a program execution trace (we only need to record the execution of operations in P and C). The set F for a given program does not vary from test to test, and can thus be computed statically. Calculating the elements in F precisely using static analysis, however, is challenging, as will be discussed in the following chapters.

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 Chapter 4, the precision of our analysis has a significant impact on the coverage results for the benchmarks.

2.2.2 Testing Framework

As defined previously, to measure *Exception Def-catch Coverage* we need to locate elements in both F (the possible *e-c links* of a program) and E (the *e-c links* experienced in a set of test runs).

The set F for a given program can be computed statically. Locating the elements in F precisely using static analysis, however, is challenging. An indepth discussion about this can be found in Chapter 3. In this section, we focus on measuring E after each test,

assuming we already have the information about F before any test. In later sections we will discuss techniques needed to meet this assumption.

Measuring Exception Def-catch Coverage

Since F contains all possible *e-c links* (p, c), it is not hard to instrument each `catch` clause c for which there exists an *e-c link* $(p', c') \in F$ and $c = c'$, in order to record them in the program trace if triggered. But as mentioned previously, these `catch` clauses can not be triggered by just manipulating input data or configurations of the program if they are dealing with environmental problems. A fault-injection engine must be used at runtime to help directing control into these `catch` clauses.

The first approach that came to our mind was just to use the traditional fault-injection method. That is, after instrumentation, we would run the program under test with the fault-injection engine enabled, and a fault will be injected into the system randomly. If the fault was injected while some *fault sensitive operation* occurs, an exception will be thrown and one of the `catch` clauses will be triggered, which will be logged in the trace.

Our intention is to log each executed *e-c link* (p, c) during the test instead of just the c end of the *e-c link*. However, at runtime, when some `catch` clause is triggered, we can inspect the call stack information stored in the caught exception using the method `printStackTrace`, to figure out the origin of the caught exception p.

It is not easy to achieve high coverage using this approach, because covering not so frequently executed *e-c links* is very challenging. For instance, a program can use a memory cache to reduce disk reads, and disk operations will become less frequent. Then it becomes less likely that a disk fault will be injected while a disk operation is underway, which is necessary to trigger the corresponding exception. Thus it will be hard to cover *e-c links* related to disk operations.

Improving Coverage

We now consider how the compiler can instrument application code to communicate with a fault-injection engine at runtime, in order to direct the fault-injection process

to obtain high program-fault coverage as measured by our metric. The intuition is that when the program runs, if we can foresee that a particular kind of fault-sensitive operation may happen in a way that has not been tested, we want to inform the fault injection engine to inject the fault at this time, hopefully to cause an exception to be thrown and to trigger the corresponding `catch` clause.

Specifically, we use *Mendosus* [38] as our fault-injection infrastructure, but our approach could, in principle, be applied using any fault-injection system that can inject the faults we study. For this work, we have extended Mendosus with an API for dynamic external direction as to when specific faults should be injected. Previously, Mendosus injected faults according to a pre-determined script comprised of traces and/or random distributions.

We currently inject only one fault per run of the program, using multiple single-fault runs to obtain high coverage. Our techniques could be used to inject multiple faults per run, but we have no way of measuring the interactions between faults, and thus have not explored this approach. Our choice of the "single-fault-per-run" approach could potentially prevent us from covering a catch clause in code that can only be reached after the recovery from a prior fault, but we do not expect this to be a problem in practice.

Once we have calculated the possible *e-c links* for a program, 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 in the corresponding `try` block, that resulted in the execution of the fault-sensitive operation. The tester must try to have the 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*, 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.

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 command line option or environment variables. Note that the fault must be selected so that one and only one fault-sensitive operation[3] will fail and throw an exception.

In addition, we need to control the scope of the fault. For instance, if we want to inject a disk fault, we want it only to affect disk read or write operations applied on the file related to the *e-c link* (p, c) we are targeting. If the file object is created previously or in the `try` block that corresponds to c, we can probably find it using static analysis. But that is not always the case. Also, we cannot let the fault affect all the disk read operations in the system because that will block some disk operations conducted by VM (e.g. class loading) which will fail the entire test. So we instrument the application to keep track of the set of network and file objects created by the application; this dynamically maintained set is stored in *Mendosus*, which can watch for file or socket *close* through the operating system, and remove that object from the set. Then, whenever a fault injected, *Mendosus* can always restrict its scope to the files and sockets in this set.

Figure 2.2 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 the following section 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

[3]e.g., network read/write, disk read/write, network accept, network connection, etc..

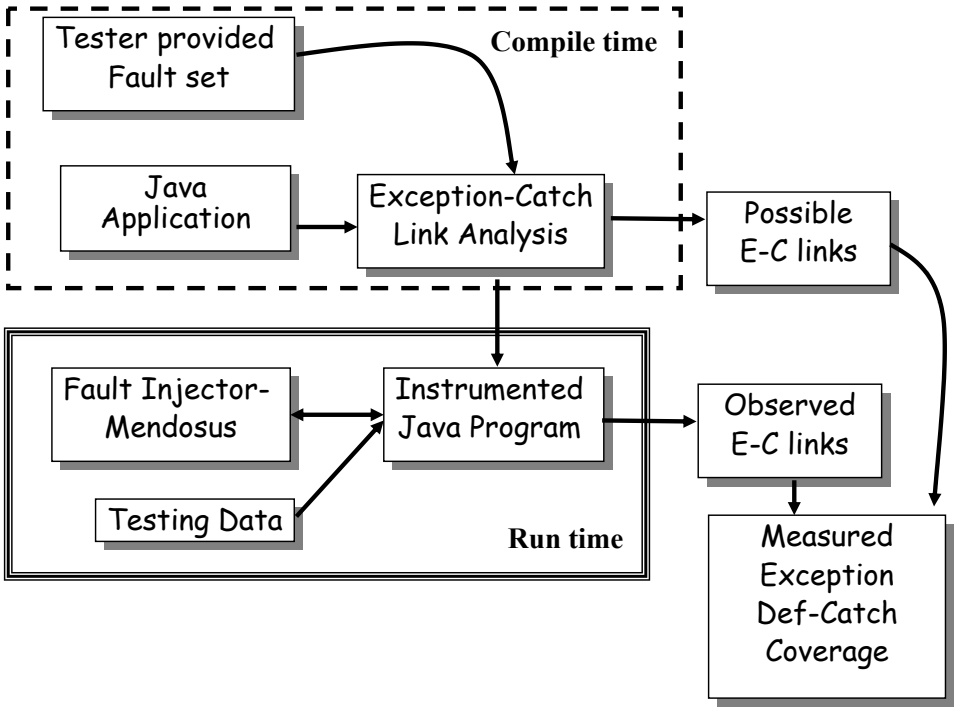


Figure 2.2: Compiler-directed fault injection framework

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.

Chapter 3

Program Analysis to Locate *e-c links*

To measure formerly defined coverage metrics, we need to know the total number of *e-c links* in the program and the number of *e-c links* that are exercised during testing. The former relies on compile-time analysis while the latter is recorded at runtime. This chapter presents in depth discussion about the compile-time analyses needed in the compiler directed fault-injection testing framework. Some of these discussions appeared earlier in publications [23, 24].

Figure 3.1 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 analyses [55, 33, 66] 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 [57, 59] 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 [20, 7, 27] which differ in their cost and the precision of the resulting graph. The empirical results discussed in Chapter 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.

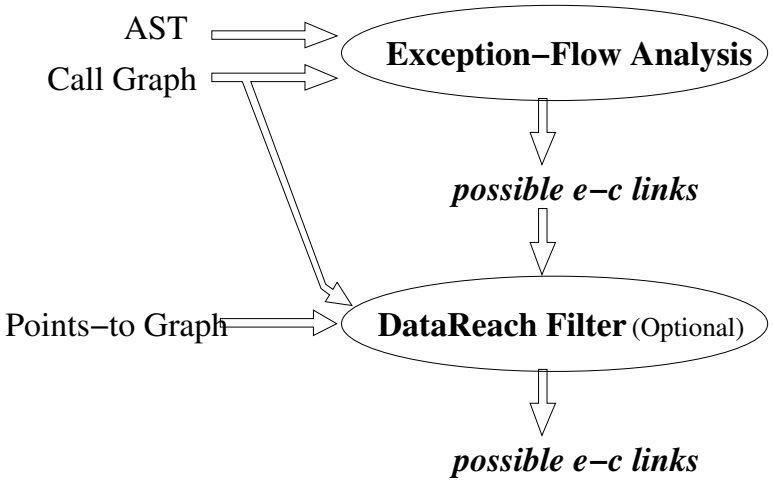


Figure 3.1: Two phases of exception-catch link analysis

3.1 Exception-flow analysis

In Java, if code in some method throws a checked exception[1], 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 [6]. We want to find the relationship between `catch` blocks and fault-sensitive operations. We use the term "`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

[1]Note that as discussed in 2.1.2, we are only considering *synchronized* exceptions in this thesis, which have definite origins.

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 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. What's more, declarations can not be used to gain any information about unchecked exceptions for which a declaration on the method signature is not required.

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. We define *exception-flow* as the flow of each exception thrown per `throw` statement along the exception handing path [50] — from the `throw` statement to the `catch` block where it is handled.

According to the semantics of exception handling in Java [6], 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 the same as or a subtype of the type of the use. Interfaces, when used as the parameter of `catch` clauses, have the same effect as abstract classes with their implementors as subclasses.

2. 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[2]. Therefore,

[2] In Java, `try` blocks can be nested within each other. Handlers are associated with exceptions in

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. The dataflow is in the reverse direction with respect to execution flow on the call graph; thus exception-flow is a backward dataflow problem. 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[3]. 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`, method call or outermost `try-catch` block.

We define for each method the set of `throw` statements that can reach its entry to be the set of exceptions that can either be generated within the method or propagated to the method (at a call) and then preserved through the method to method entry, as follows:

Definition (*ReachingThrows(method M)*): The set of all `throw` statements for which there exists an exception handling path [50] from the throw statement to method M, and the exceptions are not handled in method M.

Figure 3.2 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()`, because it is

---

inner to outer order [6].

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

a subclass of `IOException`. However, exception `OtherException`, also thrown by `bar()` while not a subclass of `IOException`, 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.

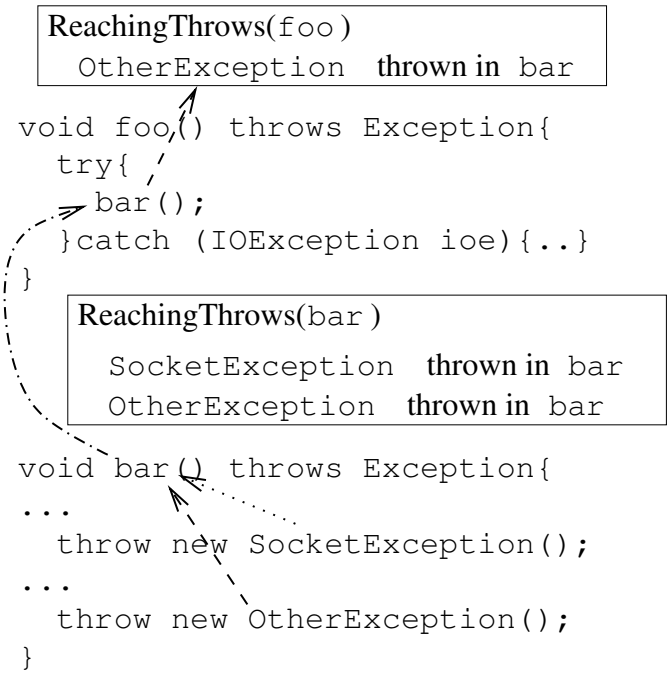


Figure 3.2: Example of ReachingThrows

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

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

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 `trycatch` 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 [40] and therefore, amenable to fixed-point iteration.[4]

In Java, `finally` blocks are used to ensure code is executed in both exceptional and normal circumstances. What's more, implicit `finally` blocks are also added whenever there are `synchronization` blocks to ensure monitor resources are properly released upon exceptional exit of these blocks. Explicit and implicit `finally` blocks are translated into special `catch` and rethrow in bytecode by `javac`. These special `catch` clauses are very easy to identify due to their special structure. We believe that code in `finally` blocks is not dedicated exception-handling code because it can be reached by normal execution and may be covered using traditional functionality testing techniques. Thus we choose to ignore `finally` blocks in our analysis. However, after minor modification, `finally` blocks can be treated as ordinary `catch` blocks in our analysis.

By performing exception-flow analysis, we can find all the *e-c links* (t_i, h_j) where a `throw` t_i can potentially trigger a `catch` block h_j. Furthermore, the interprocedural propagation path of t_i can be recorded by adding annotations onto elements of *ReachingThrow*. Thus call chains from h_j to t_i can be calculated on demand after the exception-flow analysis to help the human tester understand why a specific *e-c link* is not covered in some test.

Worst case complexity. The dataflow problem so defined is distributive and 2-bounded [40]; therefore, the complexity of the analysis is $O(n^2)$ where n is the number of

[4]The iteration is only necessary here to handle interprocedural loops. Our implementation uses a prioritized (postorder) worklist.

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 $O(n^2 + kn)$.

The exception-flow analysis described above relies on having an annotated call graph for the program. The accuracy of the analysis is affected by the accuracy of the call graph. We will show the impact of using different call graph building algorithms on the number of *e-c links* reported by exception-flow analysis in Chapter 4. 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 [43, 42]. We run a context-insensitive points-to analysis on the program after this transformation, and thus obtain some degree of context sensitivity for constructors. This eliminates some imprecision and obtains a more precise call graph and points-to graph for both our Exception-flow and DataReach analysis phases.

3.2 Data reachability analysis

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 [57, 59] 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 3.3 is an example of typical use of the Java network-disk I/O packages.

Figure 3.4 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 vulnerable to disk faults and the `try` block in `readNet` is only vulnerable to network faults. But exception-flow information is merged in `BufferedInputStream.fill()`[5] and propagated to both `readFile` and `readNet`; thus, two infeasible *e-c links* are introduced reducing the achievable run-time coverage to 50% or less.

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

Figure 3.3: Code Example for Java I/O Usage

This inaccuracy can be resolved by using a different program representation such as a call tree [60] 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 3.4, the call tree algorithm must be able to find that there are only 2 feasible call chains which

[5] We use a fully qualified naming convention in our examples; that is, we express all method names in a ClassName.MethodName format, even for instance methods.

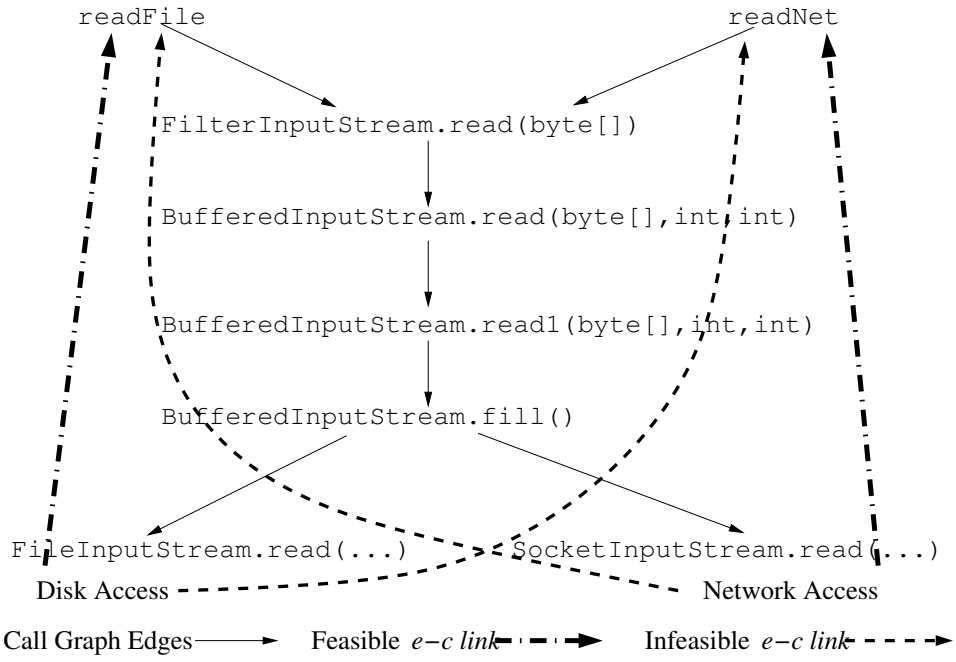


Figure 3.4: Call Graph for Java I/O Usage

share a middle segment of length 3. Separating these 2 chains would require a context-sensitive points-to analysis analogous to 4-CFA [62, 64], 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` and `DataInputStream`).

3.2.1 Overview of our Approach

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 3.3, consider the call site `fsrc.read()` in method `readFile`. We want to know whether `SocketInputStream.read()` can be called during the lifetime of `fsrc.read()`. In the explanation below, we refer to `fsrc.read()` as the *original call* and to the polymorphic call site in `BufferedInputStream.fill()` as the *target call site*, which may reach `SocketInputStream.read()`

according to the call graph. The receiver variable of the *target call site* is denoted as `rt`. The argument about data reachability relies on the following intuition: if `SocketInputStream.read()` is called, some object of type `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 `rt` by field dereferences of (i) one of the global variables (i.e., Java static fields) or (ii) one of the objects created during the lifetime of the original call, that occur **during the lifetime** of the original call.

3. The object is associated with `rt` by field dereferences of one of the arguments of the original call (including the receiver), that occur **during the lifetime** of the original call.

At the call site in method `readFile`, `fsrc` points to a `BufferedInputStream` object whose `in` field points to a `FileInputStream`. In `BufferedInputStream.fill()`, `this.in` is assigned to `rt` and a call to `rt.read(...)` is issued. According to the rules above, `FileInputStream.read(...)` is reachable because a `FileInputStream` object is reachable from `rt` by field dereference. However no `SocketInputStream` is reachable through transitive field dereferences via the fields accessed from either the arguments, the receiver of the original call, or any static field loaded, and no such object is created. Thus it is clear that during the lifetime of the original call site, `rt` cannot point to an object with type `SocketInputStream`. Therefore the polymorphic call cannot be dispatched to `SocketInputStream`, and the corresponding *e-c link* is infeasible.

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 the conditions above. 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 [36, 57] to obtain the high-quality data reachability information.

The rest of this section describes DataReach, the original data reachability algorithm and discusses sources of its imprecision. Section 3.3 presents a schema of successively more precise data reachability algorithms, inspired by the call graph construction algorithms in reference [75], in which types are propagated between different program elements (e.g., methods, classes and fields). Our algorithms propagate reachable objects instead of types. Nevertheless, our algorithms rely on the key intuitions of the algorithms in [75]: keeping separate sets associated with program constructs achieves better accuracy than keeping one set for the entire program, albeit for the collection of different data in a different problem.

3.2.2 Original DataReach Algorithm

Now we present the details of the original *DataReach* algorithm that requires a points-to graph as input. The nodes of the points-to graph are the reference variables in the program and the object names that represent the set of heap objects created during program execution. Our analysis assumes a common object naming scheme which assigns one object name per allocation site; other more precise object naming schemes are possible as well but they tend to be more expensive [43]. Let O denote the set of object names. Function Pt: $Ref \rightarrow \mathcal{P}(O)$ takes as an argument a reference variable or a reference object field and returns a subset of $\mathcal{P}(O)$, the powerset of O. DataReach is defined in terms of three sets: U, F and R. Set U is initialized to the set of objects passed as actual arguments at the original call; intuitively, it contains the universe of objects that may flow to the target call from the original call. Set F is the set of all instance fields that are read during the lifetime of the call. As the algorithm examines static and instance field accesses in the methods reachable during the lifetime of the original call, it adds to U those objects that thereby become reachable. In other words, the algorithm adds object o_j to U if and only if there is a path $o_i \xrightarrow{f_0} o_1 \ldots \xrightarrow{f_k} o_j$ in the points-to graph, where field identifiers $f_0, \ldots f_k \in F$ and $o_i \in U$ before this addition. Set R denotes the set of methods reachable during the lifetime of the original call.

The DataReach algorithm can be specified by the following constraints (using the constraint-based formalism from [75]). The statement of these constraints is followed

by a discussion of their meaning.

- **input:** Pt: $Ref \rightarrow \mathcal{P}(O)$
- **initialize:** $M \in R$ for each target M at original call
  $Pt(v) \subseteq U$ for each actual argument v at original call
  $F = \emptyset$

1. For each method M, each virtual call site $e.m(\ldots)$ in M, each object $o \in Pt(e)$ where $StaticLookup(o, m) = M'$:
   $(M \in R) \wedge (o \in U) \Rightarrow M' \in R$
2. For each method M and for each object creation statement s_i: $\ldots = new\ o_i$ in M:
   $(M \in R) \Rightarrow o_i \in U$
3. For each method M and for each static field read statement s_i: $\ldots = C.f$ in M:
   $(M \in R) \Rightarrow Pt(C.f) \subseteq U$
4. For each method M and for each instance field read statement s_i: $\ldots = r.f$ in M:
   $(M \in R) \Rightarrow f \in F$
5. $(o \in U) \wedge (f \in F) \Rightarrow Pt(o.f) \subseteq U$

The algorithm initializes the set of reachable methods R to the set of targets at the original call, U to the set of objects pointed to by the actual arguments at the original call (including all possible receivers), and the set of accessed fields F to the empty set. Auxiliary function *StaticLookup* returns the dynamic target of the call, based on the static type of the receiver object o and the compile-time target m. Constraint 1 specifies the addition of new methods to the set of reachable methods at virtual calls; a new method M' is added to R only if the receiver object that triggers the invocation of M' is in the set U. Static calls are trivially handled as their target method is known exactly. Constraint 2 specifies that an object is added to set U whenever there is an object creation statement in a reachable method. Similarly constraint 3 specifies that objects are added to U whenever a static field is accessed. Finally, constraint 4 collects the set of field identifiers accessed in reachable methods, and constraint 5 accounts for the computation of the transitive closure of U with respect to the set of accessed fields F.

The solution of these constraints can be used to judge whether or not an edge in the call graph downstream from the original call site, can be reached on a statically feasible path from that call site. The algorithm starts from the given call site and judges the feasibility of each encountered call edge using set U, before actually following the edge. The algorithm outputs R, the set of all methods reachable through data reachability from the given original call site. Recall the intended use of our DataReach algorithm. If there is no feasible path of calls to the target method during the lifetime of the original call, then the corresponding *e-c link* is proved spurious.

If a fault occurs during the lifetime of the original call, then an exception may be handled by a `catch` block associated with the `try` in which the original call site is nested. In this case, there is a corresponding *e-c link* resulting from an excepting call to some method m or `throw` in method m during the lifetime of the original call. If at the target call to m, the set of possible target methods does not contain m, then the *e-c link* is spurious (i.e., it corresponds to an infeasible control-flow path); thus, there is no need for this link to be exercised.

For example, if the original call is `in.read()` in method `readFile` and *reachable_methods* calculated by the above algorithm *does not* include method `SocketInputStream.read()`, then we know that the `catch` block in `readFile` will not be triggered by exceptions thrown in `SocketInputStream.read()` so the corresponding *e-c link* is infeasible.

3.2.3 Imprecision of DataReach

The original data reachability algorithm produced relatively precise results. However, examples from several new benchmark programs reveal that in many cases its conservative estimate is not sufficient. Therefore, there is a need to investigate more precise analysis.

Example. Consider the example in Figure 3.5. Assume we start DataReach analysis at original call c_1 in method `Read1`. Set U will contain objects o_1, o_2 and o_5 and every object reachable from them along fields accessed in the reachable methods `A.m`, `A.n` and `Hashtable.put`. Since context-insensitive points-to analysis and even some of

the practical context-sensitive ones (e.g., 1-CFA) do not distinguish between objects stored in different containers or maps, any object that is stored in a `Hashtable` object will be reachable from o_5 along a path of field accesses in F. Thus, the set of objects reachable from o_5 includes o_4 and we have $\{o_1, o_2, o_4, o_5\} \subseteq U$. As a result, both `Y.read` and `Z.read` are determined to be feasible targets at call `x.read()` and the analysis erroneously concludes that both the `throw` in `Y.read` and the `throw` in `Z.read` will be handled by the `catch` block in method `Read1`. Similarly, starting DataReach from original call c_2 in method `Read2`, the analysis determines that both the `throw` in `Y.read` and the `throw` in `Z.read` will be handled by the `catch` block in method `Read2`. It is easy to see that the only two feasible *e-c links* are (i) between `throw new SomeIOException` and the `catch` in `Read1`, and (ii) between `throw new OtherIOException` and the `catch` in `Read2`. Similar patterns in actual benchmark code led us to investigate a more precise analysis.

3.3 A Schema for Data Reachability Analysis

We propose a new general schema for data reachability analysis, that includes our original DataReach algorithm as an instantiation. Similarly to the call graph construction algorithms by Tip and Palsberg [75], our schema can be instantiated to yield different algorithms by varying the number of sets used to calculate the objects which are visible in methods reachable from the original call, (i.e., the set from which the possible receivers at the target call are drawn). DataReach keeps a single set U. The new data reachability algorithms in our schema keep separate sets for program entities such as classes, methods and reference variables. The major differences with Tip and Palsberg's algorithms are that (i) our algorithm propagates objects rather than class types, and (ii) our algorithm is formulated on a *partial* program rather than on a *complete* program. The algorithms in our schema keep specialized local information for program entities such as methods and reference variables, which results in increased precision for data reachability calculations. For example, consider the set of statements in Figure 3.5. Clearly, the `Hashtable` object o_5 created in method `A.n` does not flow to `A.m`; thus, the precision of the data reachability analysis will benefit if instead of keeping a single set

```
class X {
    void read() throws IOException {...}
}
class Y extends X {
    void read() throws IOException {
        ... if (...) throw new SomeIOException();
    }
}
class Z extends X {
    void read() throws IOException {
        ... if (...) throw new OtherIOException();
    }
}

class A {
    void m(X x) throws IOException {
        n(x);
        x.read();
    }
    void n(X x) {
s5:     Hashtable ht = new Hashtable(); //o5
        ... if (...) ht.put(...,x);
    }
}


    void Read1() {
        try {
s1:         A a = new A(); //o1
s2:         Y y = new Y(); //o2
c1:         a.m(y);
        }catch(IOException e) { ... }
    }

    void Read2() {
        try {
s3:         A a = new A(); //o3
s4:         Z z = new Z(); //o4
c2:         a.m(z);
        }catch (IOException e) { ... }
    }
```

Figure 3.5: Imprecision of DataReach algorithm

U throughout the analysis, a set U_M is kept for each method M.

The original DataReach algorithm is an instance of the schema. In this section we discuss two additional instances: one that uses separate sets U_M for each method M (this instantiation is referred to as *M-DataReach*), and one that uses separate sets U_V for each reference variable V (referred to as *V-DataReach*). Analogously to the algorithms in [75], M-DataReach is potentially more precise than DataReach, and V-DataReach is potentially more precise than M-DataReach. It is possible to define an algorithm, where there is a set per class by aggregating the method sets for all methods in that class into a single set U_C (referred to as *C-DataReach*); for brevity we omit a detailed discussion of this instantiation.

3.3.1 Separate sets for methods (M-DataReach)

The M-DataReach algorithm keeps distinct sets U_M and F_M for each method M; U_M is computed with respect to F_M from the points-to graph given as input to the algorithm. Analogously to [75], $ParamTypes(M)$ is used for the set of static types of the arguments of method M (excluding the implicit parameter `this`), and the notation $ReturnType(M)$ is used for the static return type of M. $MatchingObjects(t, U)$ denotes the set of objects in U of type t (or of a subtype of t). We extend the notation $MatchingObjects(.)$ to apply to a set of types as follows: $MatchingObjects(T, U) = \bigcup_{t \in T} MatchingObjects(t, U)$.

The following constraints define M-DataReach:

- **input:** Pt: $Ref \rightarrow \mathcal{P}(O)$
- **initialize:** $M \in R$ for each target M at original call
  $Pt(v) \subseteq U_M$ for each actual argument v at original call and for each target M
  $U_N = \emptyset$ for each non-target method N
  $F_M = \emptyset$ for each method M

1. For each method M, each virtual call site $e.m(\ldots)$ occurring in M, each object $o \in Pt(e)$ where $StaticLookup(o, m) = M'$:
   $(M \in R) \wedge (o \in U_M) \Rightarrow$

$$\begin{cases} M' \in R \wedge \\ MatchingObjects(ParamTypes(M'), U_M) \subseteq U_{M'} \wedge \\ MatchingObjects(ReturnType(M'), U_{M'}) \subseteq U_M \wedge \\ o \in U_{M'} \end{cases}$$

2. For each method M and for each object creation statement s_i: $\ldots = new\ o_i$ in M:
   $(M \in R) \Rightarrow o_i \in U_M$
3. For each method M and for each static field read statement s_i: $\ldots = C.f$ in M:
   $(M \in R) \Rightarrow Pt(C.f) \subseteq U_M$
4. For each method M and for each instance field read statement s_i: $\ldots = r.f$ in M:
   $(M \in R) \Rightarrow f \in F_M$
5. $(o \in U_M) \wedge (f \in F_M) \Rightarrow Pt(o.f) \subseteq U_M$

Intuitively, constraint 1 refines the analogous constraint from DataReach. First, the receiver object o at a virtual call in method M should be available in U_M. Second, set U'_M of the callee is updated with the objects from set U_M of M that match the parameter types of the callee. Third, set U_M of the caller M is updated with the objects from set $U_{M'}$ of the callee M' matching the return types of the callee. Constraints 2 and 3 respectively gather objects created in M, and objects that flow to M due to static field reads. Finally, constraint 4 gathers the set of instance fields that may be accessed in M and constraint 5 computes the transitive closure of U_M by only traversing points-to graph edges corresponding to fields in F_M.

Example. Consider the code in Figure 3.5. After initialization at original call c_1 we have $U_{A.m} = \{o_1, o_2\}$. Applying constraint 1 at call `n(x)` results in objects o_1 and o_2 being added to $U_{A.n}$; no objects flow back to $U_{A.m}$. Since no fields are accessed in `A.m`, the closure is $U_{A.m} = \{o_1, o_2\}$. Therefore, the only possible receiver at call `x.read()` is o_2 and the only possible exception that may be thrown back to the original call is `SomeIOException`.

3.3.2 Separate sets for variables (V-DataReach)

Additional precision over M-DataReach can be achieved by distinguishing the object sets for each reference variable. For this instantiation of the schema, called V-DataReach, the algorithm keeps distinct sets U_V for each reference variable V. This analysis takes advantage of a predicate $\mathit{MethodLocal}(o)$ which returns *true* if object o does not escape its creating method, and *false* otherwise. This information can be trivially computed from a points-to graph as shown in [57].

The following constraints define V-DataReach, in an analogous way to the two previous instantiations of the schema:

- **input:** Pt: $\mathit{Ref} \rightarrow \mathcal{P}(O)$
- **initialize:** $M \in R$ for each target M at original call
  $U_{a_i} \subseteq U_{M.f_i}$ for actuals a_i and formals $M.f_i$
  Initialize $U_{M.this}$ of targets M accordingly
  Initialize all other U_v, $U_{o.f}$ and Local to $\emptyset$

1. For each method M,
   each virtual call site $l = e.m(e_1, \ldots, e_n)$ occurring in M,
   each $o \in Pt(e)$ where $\mathit{StaticLookup}(o, m) = M'$:
   $(M \in R) \wedge (o \in U_e) \Rightarrow$
   $$\begin{cases} M' \in R \wedge \\ U_{e_i} \subseteq U_{M'.f_i} \text{where } f_i \text{ are the formal parameters of } M' \wedge \\ U_{M'.ret\_var} \subseteq U_l \wedge \\ o \in U_{M'.this} \end{cases}$$
2. For each method M and for each reference assignment statement s_i: $l = r$ in M:
   $(M \in R) \Rightarrow U_r \subseteq U_l$
3. For each method M and for each object creation statement s_i: $l = new\ o_i$ in M:
   $$\begin{cases} (M \in R) \Rightarrow o_i \in U_l \\ (M \in R) \wedge \mathit{MethodLocal}(o_i) \Rightarrow o_i \in \mathit{Local} \end{cases}$$

4. For each method M and for each static field read statement $l = C.f$ in M:
   $(M \in R) \Rightarrow Pt(C.f) \subseteq U_l$

5. For each method M, for each instance field write statement $l.f = r$ in M and each $o_i \in Pt(l)$ where $o_i \in Local$:
   $(M \in R) \wedge (o_i \in U_l) \Rightarrow U_r \subseteq U_{o_i.f}$

6. For each method M, for each instance field read statement $l = r.f$ in M and each $o_i \in Pt(r)$:
   $(M \in R) \wedge (o_i \in U_r) \Rightarrow$
   $$\begin{cases} o_i \in Local \Rightarrow U_{o_i.f} \subseteq U_l \wedge \\ o_i \notin Local \Rightarrow Pt(o_i.f) \subseteq U_l \end{cases}$$

Intuitively, constraints 1-4 refine the corresponding constraints from M-DataReach. V-DataReach keeps flow information per reference variable instead of per method; therefore it produces more precise results. The following example illustrates the benefits of these constraints.

Example. Consider the set of statements in Figure 3.6. Starting from original call c_1 in `Read1`, M-DataReach will compute $U_{A.m} = \{o_1, o_2, o_3\}$. At target call site `x1.read()` in `A.m` the two possible receivers according to the input points-to graph are o_1 and o_2. Since both o_1 and o_2 are in $U_{A.m}$, they are determined to be valid receivers; therefore, the `throw SomeIOException` and the `throw OtherIOException` statements flow to the `catch` in `Read1`. In contrast, V-DataReach is able to avoid this imprecision because it keeps separate sets U_{x1} and U_{x2} for $x1$ and $x2$ respectively.

Constraints 5 and 6 refine constraint 5 from M-DataReach. Note that constraint 3 collects set *Local*; this set contains objects o instantiated during the traversal of reachable methods that do not escape their creating method. Clearly, since the objects in *Local* do not escape their creating method, they do not escape the lifetime of the original call. The role of constraint 5 is to separate instance field writes to objects in *Local*. For those objects, all field writes occur during the lifetime of the original call and the values assigned to their fields can be collected from the right-hand-side of the field write statement in set $U_{o.f}$. Constraint 6 accounts for propagating field values.

```
abstract class X {
    void abstract read() throws IOException;
}
class Y extends X{
    void read() throws IOException {
        ... if (...) throw new SomeIOException();
    }
}
class Z extends X{
    void read() throws IOException {
        ... if (...) throw new OtherIOException();
    }
}

class A {
    void m(X x1,X x2) throws IOException {
         ... x1.read();
    }
}

class B{
s1: static X xy = new Y();//o1
s2: static X xz = new Z();//o2
}

    void Read1(){
        try {
s3:         A a = new A();//o3
c1:         a.m(B.xy,B.xz);
        } catch (IOException e) {...}
    }

    void Read2(){
        try{
s4:         A a = new A();//o4
c2:         a.m(B.xz,B.xy);
        } catch (IOException e) {...}
    }
```

Figure 3.6: Imprecision of M-DataReach algorithm on different references

For objects $o \in Local$ (i.e., objects whose lifetime does not exceed the lifetime of the original call), the values of an accessed field f are collected from sets $U_{o.f}$. For objects $o \notin Local$ (i.e., objects whose lifetime may exceed the lifetime of the original call) the possible field values are approximated from the global points-to solution since those fields may be set outside of the original call. The following example taken from the *HttpClient* benchmark illustrates the additional precision gained from separating writes to fields of local objects.

```
class A{
  void read() throws IOException;
}
class Dmy extends A{
  void read() {...}
}
class Res extends A{
  void read() throws IOException{
    .. throw new IOException; ..
  }
}

class W{
  A f;
  void W(A a) { f = a; }
  void read() throws IOException{
    A a = this.f
    a.read();
  }
}

class M{
  void getData(A a) throws IOException{
s1: W w = new W(a);      //o1
    w.read();
  }
  void getDmy() {
    try{
s2:   A dmy = new Dmy(); //o2
c1:   getData(dmy);
    } catch (IOException e) {...}
  }
  void getRes() {
    try {
s3:   A res = new Res(); //o3
c2:   getData(res));
    } catch (IOException e) {...}
  }
}
```

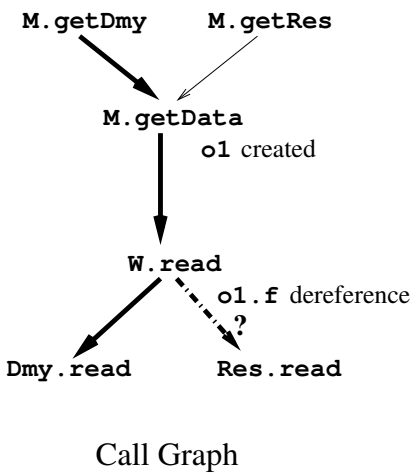


Figure 3.7: Imprecision of M-DataReach algorithm on local objects

Example. Consider the example in Figure 3.7. Starting V-DataReach from original call c_1 in `getDmy` we have $U_{getData.w} = \{o_1\}$ and $U_{getData.a} = \{o_2\}$. Clearly, object o_1 does not escape its creating method (i.e., its lifetime does not exceed the lifetime of the

original call); therefore the instance fields of o_1 are assigned during the lifetime of the original call. Therefore, as a result of constraint 5 for instance field write `this.f = a` in the constructor of class `W`, we have $U_{o_1.f} = \{o_2\}$. Similarly, as a result of constraint 6 for instance field read `a = this.f` in `W.read`, the set U_a will be read from the set $U_{o_1.f}$. Therefore, $U_{read.a} = \{o_2\}$ and as a result the only possible target at the call `a.read()` is `Dmy.read`. Consequently, V-DataReach concludes that no exception will be thrown and caught in `getDmy`. In contrast if U_a was read from $Pt(o_1.f)$, $U_{read.a}$ would be $\{o_2, o_3\}$, so we have to consider this *e-c link* feasible while it is actually not. With M-DataReach, $U_{W.read} = \{o_1, o_2, o_3\}$, so the same imprecision occurs. Analogously, V-DataReach concludes that starting from original call c_2 the exception in `Res.read` may be thrown and caught in `getRes` which leads to the only *e-c link*.

3.3.3 Complexity of algorithms in schema

For a given program let $\mathcal{C}$ be the number of classes, $\mathcal{M}$ be the number of methods, $\mathcal{V}$ be the number of reference variables, including static fields, $\mathcal{O}$ be the number of object allocation sites, and $\mathcal{F}$ be the number of instance field identifiers.

The complexity of a data reachability analysis that fits our schema depends on the number k of U sets kept during propagation. The overall complexity can be broken into three components: (i) the complexity of generating inclusion constraints for program statements (constraints 1-3 for DataReach and M-DataReach, and 1-4 for V-DataReach), (ii) the complexity of solving the system of inclusion constraints, and (iii) the complexity of computing the field closure for sets U (constraints 4 and 5 for DataReach and M-DataReach and 5 and 6 for V-DataReach). The complexity of constraint generation is dominated by the time to process virtual calls. Let E be the number of call graph edges and let there be an array a_o for each object o indexed by the unique identifiers i of sets U_i. Field $a_o[i].value$ equals 1 if $o \in U_i$ and 0 if $o \notin U$; field $a_o[i].edges$ contains the set of call graph edges triggered whenever $a_o[i].value$ becomes 1 (i.e., whenever o is added to U_i). Constraints for virtual calls are generated whenever o is added to U_i. Since each edge can belong to at most $\mathcal{O}$ $a_o[i].edges$ sets, the complexity of (i) is $O(\mathcal{O} * E)$. The complexity of (ii) is $O(\mathcal{O} * k^2)$ since for every

U_i there are at most $\mathcal{O}$ objects that can be propagated through U_i to at most k sets U_j. Finally, the complexity of (iii) is $O(\mathcal{O}^2 * \mathcal{F} * k)$. Therefore the complexity of our algorithms parameterized by k, the number of U sets, is: $O(\mathcal{O} * E + \mathcal{O} * k^2 + \mathcal{O}^2 * \mathcal{F} * k)$.

The following table summarizes our analysis in order of growing precision and complexity, because E is dominated by M^2 and V^2:

Table 3.1: Data Reachability Algorithms

Algorithm	*U sets*	*Complexity*
DataReach	1	$O(E * \mathcal{O} + \mathcal{O}^2 * \mathcal{F})$
C-DataReach	$\mathcal{C}$	$O(\mathcal{O} * E + \mathcal{O} * \mathcal{C}^2 + \mathcal{O}^2 * \mathcal{F} * \mathcal{C})$
M-DataReach	$\mathcal{M}$	$O(\mathcal{O} * \mathcal{M}^2 + \mathcal{O}^2 * \mathcal{F} * \mathcal{M})$
V-DataReach	$\mathcal{V}$	$O(\mathcal{O} * \mathcal{V}^2 + \mathcal{O}^2 * \mathcal{F} * \mathcal{V})$

Chapter 4

Experiment Results on Compiler-Directed Fault Injection Testing

To demonstrate the effectiveness of our methodology, we collected a set of Java server programs and conducted *compiler-directed fault injection testing* experiments. In this chapter we report our empirical findings and discuss some case histories from our experiments. Some of these discussions appeared earlier in publications [23, 24].

4.1 Experimental setup & benchmarks

We implemented Exception-flow analysis and DataReach/M-DataReach analysis as two separate modules in the Java analysis and transformation framework Soot[59] 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/M-DataReach postpass. Soot provides a call graph builder using *Class Hierarchy Analysis* (CHA)[20], and *Spark*, a field-sensitive, flow-insensitive and context-insensitive points-to analysis (a form of 0-CFA)[64, 58, 57, 36]. We implemented another call graph builder using *Rapid Type Analysis* (RTA)[7]. We also implemented the instrumentation phase as a separate module in Soot, which automatically instruments the program according to the set of possible *e-c links*, as described in the end of Section 2.2.1.

We experimented with the following seven different analysis configurations:[1]

1. CHA — Build call graph with Class Hierarchy Analysis.

2. RTA — Build call graph with Rapid Type Analysis.

[1] Selective constructor inlining, DataReach and M-DataReach were only used where stated explicitly.

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 and 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.

7. InPTA-MDR — Use Spark plus selective constructor inlining to provide the points-to graph and the call graph, and use M-DataReach as a postpass filter.

We used seven Java applications as our benchmarks:

- FTPD, a Ftp Server in Java [68]
- JNFS, The Java Network File System, 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 [51]
- Muffin, a web filtering proxy server [44]
- Haboob, a simple web server based on SEDA, a staged event-driven architecture [78]
- HttpClient, an HTTP utility package from the *Apache Jakarta Project* [2]. We collected its unit tests to form a whole program to serve as a benchmark.
- SpecJVM, a standard benchmark suite[71] that measures performance of Java virtual machines, especially for running client side Java programs
- VMark, a Java server side performance benchmark. It is based on *VolanoChat* [77] — a web-based chat server. The benchmark includes the chat server and a simulated client

Column 2 of Table 4.1 shows the number of user classes, with those in parentheses comprising the JDK library classes reachable from each application. The data in column

3 shows the number of user methods and those in parenthesis are the JDK library methods reachable from each application. Column 4 gives the number of `try` blocks in user code. The last column shows the size of the *.class* files (in bytes) of each benchmark, excluding the Java JDK library code. The reachable method counts are calculated by Spark. JNFS is the only multi-node application.[2]

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 hinders the process of interpreting our experimental results.

Table 4.1: Benchmarks

Name	*Classes*	*Methods*	*Try Blocks*	*.class Size*
FTPD	11(1407)	128(7479)	17	39,218
JNFS	56(1664)	447(9603)	36	175,297
Muffin	278(1365)	2080(7677)	270	727,118
Haboob	338(1403)	1323(7432)	134	731,413
HttpClient	252(2210)	1334(4741)	536	1,049,784
SpecJVM	484(2161)	2489(4592)	219	2,817,687
VMark	307(2266)	1565(5029)	502	2,902,947

As shown by the the workflow in Figure 2.2, we ran the instrumented code with various workloads to exercise different vulnerable operations in the applications. Experienced *e-c links* were recorded in a log file during the testing. By processing the *e-c link* information file and the log file after the testing we obtained 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[74, 47]. We only injected one fault per run, resulting in at most one *e-c link* covered per test; therefore, we needed to run each benchmark multiple times, each time targeting one *e-c link*. Because we lack a model

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

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. Note that we ran all the benchmarks in SpecJVM together as one Java program, because the I/O module in SpecJVM is shared across all the benchmarks. As usual it was the tester's responsibility to find proper inputs and program configurations, so that designated vulnerable statements (and fault-sensitive operations) were executed.

4.2 Empirical data

Table 4.2 lists the number of *e-c links* reported for each benchmark in each analysis configuration. Column 9 (*Reached*) lists the number of links, among those discovered in InPTA-MDR, whose corresponding `try` block (but not necessarily the `catch` block) was executed by a test. The last column (*Covered*) shows the number of *e-c links* actually covered for each benchmark by the testing. Table 4.3 shows the overall exception def-catch coverage for all the benchmarks derived from the data in Table 4.2. 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 positive *e-c links* reported in most benchmarks are 7 to 150 times more than the actual *e-c links* that we can cover in the testing. 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.

We offer 2 different calculations for the percentage *e-c links* covered. In columns 2-8 of Table 4.3, we use the metric described in Section 2.2.1 (i.e., the ratio of *e-c links* covered to possible *e-c links* found by our analysis). In the last column (9) of Table 4.3, we calculate the ratio of the number of *e-c links* exercised to the number of links whose corresponding `try` block was executed by a test execution. Effectively, this second measure factors in how well the tests we are using to execute the program actually cover the set of `try` blocks in the code. If we cannot cause execution to reach the `try` block containing a vulnerable operation, then we cannot expect to inject a fault to test the recovery code corresponding to that operation. The difference between the

values of these two metrics indicates the need for additional tests for our benchmarks and also distinguishes between possible spurious *e-c links* which have not been covered from *e-c links* (spurious or not spurious) which had no chance of being covered in these executions.

Table 4.2: Number of *e-c links* Reported

Program	CHA	RTA	PTA	InPTA	PTA DR	InPTA DR	InPTA MDR	Reached	Covered
FTPD	34	34	16	16	16	13	13	13	**11**
JNFS	104	104	39	39	22	19	19	19	**16**
Muffin	480	258	112	112	87	42	42	42	**35**
Haboob	96	73	12	12	12	12	12	12	**10**
HttpClient	1946	1946	255	251	238	118	107	105	**65**
SpecJVM	511	511	90	82	72	54	47	37	**7**
VMark	2039	2039	130	100	109	57	47	18	**13**

Table 4.3: Overall Exception Def-catch Coverage

Program	CHA	RTA	PTA	InPTA	PTA DR	InPTA DR	InPTA MDR	Effective
FTPD	32%	32%	69%	69%	69%	85%	85%	85%
JNFS	15%	15%	41%	41%	72%	84%	84%	84%
Muffin	7%	14%	31%	31%	40%	83%	83%	83%
Haboob	10%	14%	83%	83%	83%	83%	83%	83%
HttpClient	3%	3%	25%	26%	27%	55%	61%	62%
SpecJVM	1%	1%	8%	9%	10%	13%	15%	19%
VMark	1%	1%	10%	13%	12%	23%	28%	72%

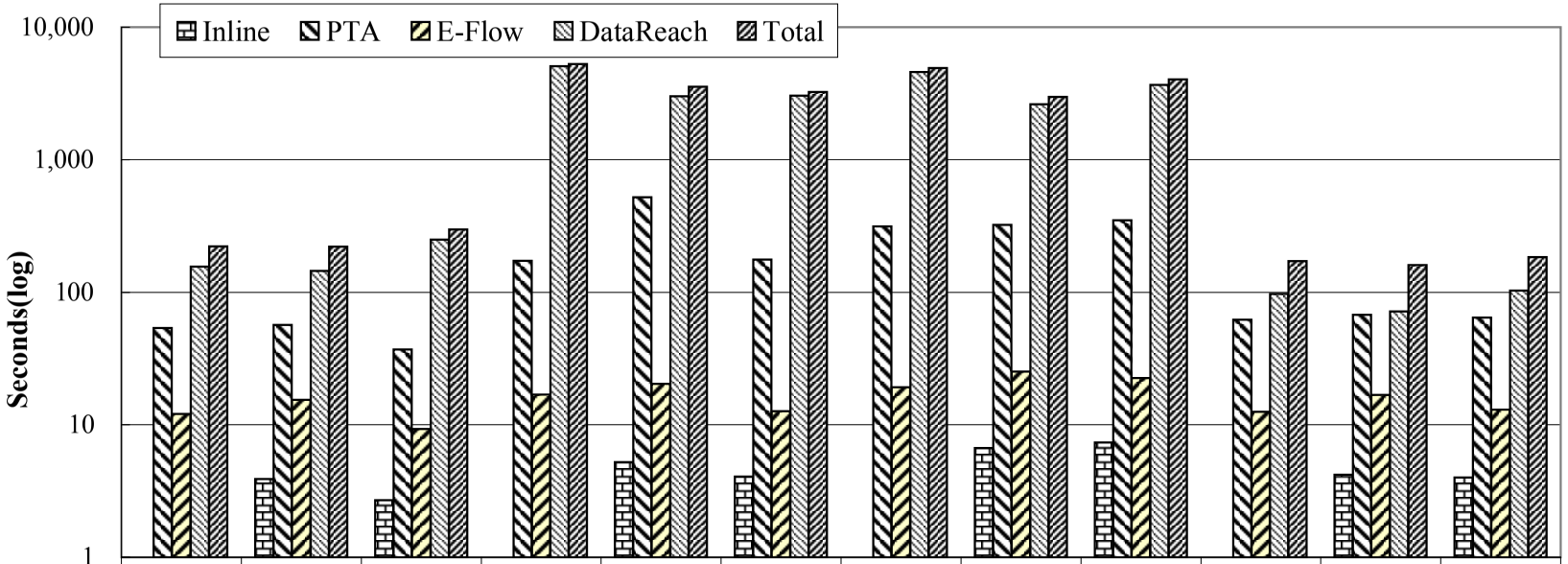
The context sensitivity obtained by adding selective constructor inlining before performing points-to analysis had effect only on the larger three benchmarks (i.e., compare columns PTA and InPTA in Table 4.2). However, when combined with the DataReach postpass, the additional precision provided reduced the number of reported *e-c links* in six of the seven benchmarks (i.e., compare columns PTA and InPTA-DR in Table 4.2). For the *e-c links* reported by InPTA-DR, the coverage percentage of the four smaller benchmarks was stable at approximately 84% with small variance. In Muffin and HttpClient, the additional precision helped cut the number of reported *e-c links* by more than

half. Haboob is special because 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. Thus the simple PTA analysis is sufficient to analyze Haboob, as shown in Table 4.2. From this data we see that DataReach is a client of precise points-to analysis for which the added precision can make a difference. In all three larger benchmarks, M-DataReach provides more precision over original DataReach algorithm (i.e., compare columns InPTA-DR and InPTA-MDR in Table 4.2).

On the three larger benchmarks the coverage varied across the programs from 15% to 72%. Sections 4.3.2, 4.3.3 and 4.3.4 discuss these benchmarks and describe the causes for the lack of coverage gleaned from code inspection, where possible.

Figures 4.1 and 4.2 shows the running times of each part of the static analysis on all benchmarks using configurations PTA-DR, InPTA-DR and InPTA-MDR. Running times of the instrumentation phase are too small to be shown, under 5 seconds for all the benchmarks. Our analysis always finished in less than 2 hours. In the worst case for the InPTA-MDR configuration, the time our analysis took to find one *e-c link* in a program on average was less than 3 minutes. DataReach is time consuming compared to Exception-flow analysis and points-to analysis in Spark, 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 4.2). For FTPD and Haboob, DataReach used about 50% of the total running time; for other benchmarks, it used more than 90% of the total running time. M-DataReach is slower than Data-Reach in most of the benchmarks, except SpecJVM. It takes 72% more time to finish in FTPD, 43% in Haboob, 40% in Muffin and 15% in HttpClient. It takes 14% less time to finish in SpecJVM. We believe that optimized implementations of DataReach and M-DataReach will improve overall analysis performance significantly.

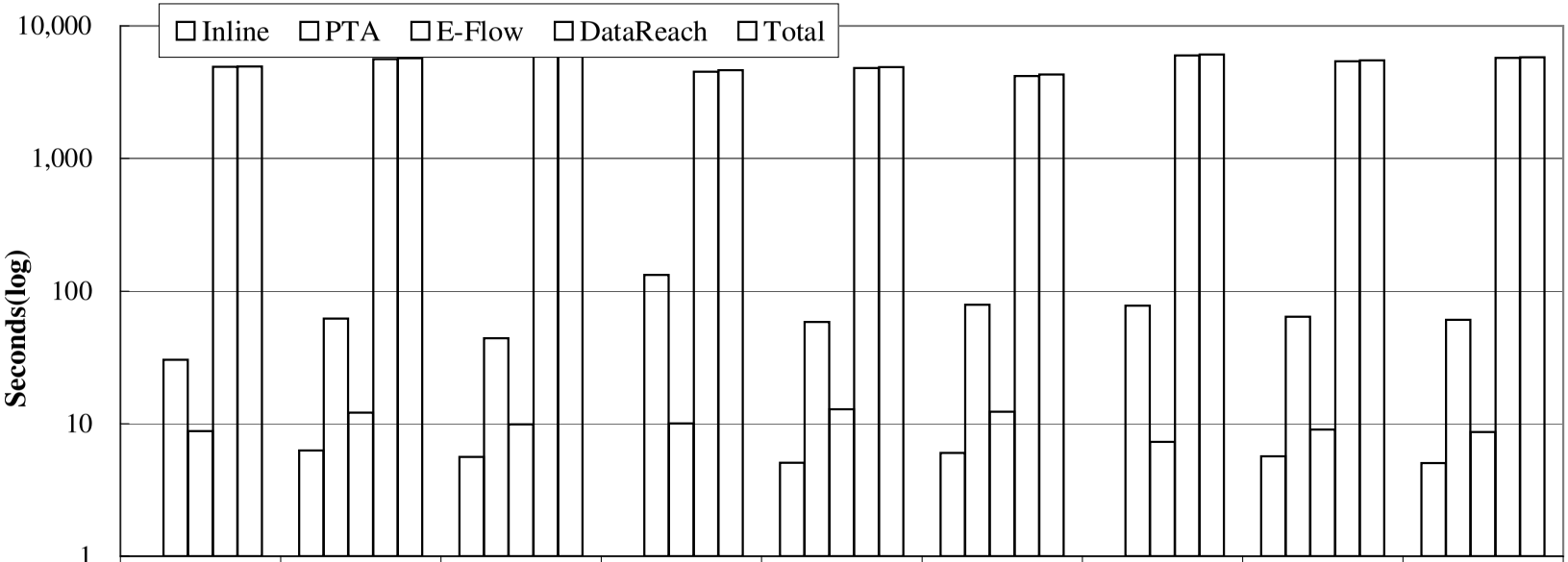
Note also that for JNFS, Muffin and VMark, the more precise analysis, InPTA-DR, ran more quickly than the related less precise analysis, PTA-DR. This is a phenomenon often seen in practice in static analysis, when a more precise analysis eliminates so much spurious information from a solution, that it actually finishes more quickly than a worst-case more efficient, less precise analysis.



	FTPD			JNFS			Muffin			Haboob		
		5)	6)	7)	5)	6)	7)	5)	6)	7)	5)	6)	7)
Inline		3.9	2.7	0.0	5.2	4.1	0.0	6.7	7.4		4.2	4
PTA	53.7	56.7	37	172.4	522.2	176.5	313.2	323.1	348.4	62.2	67.7	64.5
E-Flow	12.1	15.4	9.3	16.9	20.4	12.6	19.2	25.2	22.5	12.5	16.8	13
DataReach	155.8	144.6	249.2	5071.2	3010.6	3046.3	4582.0	2619.9	3664.8	97.2	71.7	103
Total	221.6	220.6	298.2	5260.5	3558.4	3239.5	4914.5	2974.9	4043.1	171.9	160.4	184.5

5) PTA-DR 6) InPTA-DR 7) InPTA-MDR

Figure 4.1: Time Cost Break-down of Static Program Analysis – Smaller Benchmarks



	5)	6)	7)	5)	6)	7)	5)	6)	7)
	HttpClient			SpecJVM			Vmark		
Inline	0.0	6.3	5.6	0.0	5.1	6.0	0.0	5.7	5.0
PTA	30.3	62.1	44.1	132.2	58.4	78.9	77.5	64.1	60.7
E-Flow	8.8	12.1	9.9	10.0	12.8	12.3	7.3	9.0	8.7
DataReach	4911.7	5616.9	6444.9	4503.7	4807.8	4192.9	6005.2	5423.0	5736.0
Total	4950.9	5697.3	6504.5	4645.9	4884.1	4290.1	6090.1	5501.7	5810.4

5) PTA-DR 6) InPTA-DR 7) InPTA-MDR

Figure 4.2: Time Cost Break-down of Static Program Analysis – Bigger Benchmarks

In the remainder of this section we will discuss the performance of our methodology in detail on Muffin, HttpClient, SpecJVM and VMark.

4.3 Code Inspection and Insights

Finding benchmarks for the experimental validation of our approach has been hard. We need benchmarks which include input data that exercises different parts of the program code. There is no standard benchmark suite designed for this purpose. Of all the programs that are used as benchmarks in this paper, VMark, HttpClient and SpecJVM came with input data or tests; for the others, we had to compose tests. By comparing columns 8 and 9 of Table 4.3, we can see that the input data or tests included in these benchmarks are not sufficient to drive the programs to all `try` blocks that contain vulnerable operations.

For the three larger benchmarks, we were not able to manipulate input data fully or to compose enough new tests to the extent that can fully exercise different paths in the program so as to ensure that each *e-c link* 's vulnerable operation was reached, or to maximize the coverage as we did for the smaller benchmarks.

For Muffin, SpecJVM and HttpClient, we manually inspected all the *e-c links* reported by the most precise analysis configuration whose `try` blocks were reached during the testing although these *e-c links* were not experienced[3]. We categorize these *e-c links* as follows:

1. Feasible *e-c links* not covered because of insufficient tests or input data.

2. Infeasible *e-c links* that will be difficult for any static analysis to prune.

3. Infeasible *e-c links* that may be eliminated using more precise static analysis.

Table 4.4 shows the number of inspected *e-c links* in each of the categories for each benchmark studied, and as a percentage of the total number of inspected *e-c links* in that benchmark. The last column lists the total number of inspected *e-c links*. Next

[3]We cold not perform this detailed study for VMark because we don't have its source.

Table 4.4: Number of uncovered *e-c links* in category 1, 2 and 3

Program	1	2	3	Total
Muffin	1(14%)	3(43%)	3(43%)	7
SpecJVM		4(13%)	26(87%)	30
HttpClient	10(25%)	24(60%)	6(15%)	40

we will show examples extracted from each benchmark to illustrate each category in detail.

4.3.1 Muffin

The one *e-c link* in the first category 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 does not occur in our tests since the messages are just domain names and IP addresses. Therefore, to cover this *e-c link* a test needs to include extremely long URL names to force use of TCP.

As for the three *e-c links* in the second 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. But the program code was so written that no remote file would ever be written, so that these three *e-c links* can never be covered.

There are 3 *e-c links* discovered in Muffin in category 3, which may be eliminated using points-to analysis with higher level of context-sensitivity. 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 given these call chains, the job of deciding whether an uncovered *e-c link* in this category is actually feasible is hard, since these call paths are prohibitively long and confusing to trace.

Below is one of the possible call chains found by our analysis for one of these *e-c*

links.[4] There are several hundred call chains 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()`

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 `InputStream`s. Although no object of `GZIPInputStream` has ever been assigned to the subsequent input stream of `SequenceInputStream`, the usage of the container 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`.

[4]Parameters are omitted for readability.

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 the imprecision of the underlying context-insensitive points-to analysis in a manner similar to the analysis imprecision for constructors discussed previously.

Recall that we use inlining of constructors that set object fields through `this`, to gain partial context sensitivity in our points-to analysis. Although this introduces some additional precision into our analysis, it remains a context-insensitive points-to analysis. By using M-DataReach, rather than DataReach, we may be able to increase further the precision of our analysis. This result has been confirmed in our experiments. However, even M-DataReach can have some imprecision. For example, when the receiver of a virtual method invocation is an element extracted from a container, as in the call chains corresponding to these three *e-c links*, many spurious method calls may be introduced and they can not be eliminated by M-DataReach.

More precise points-to analysis [43] addresses this problem by distinguishing calls by their receiver object when analyzing methods, thus producing a more sparse (and precise) points-to graph; this should reduce the call chains for a *e-c link*, or maybe even make it possible for DataReach to judge that the *e-c link* is actually infeasible. Further experimentation is needed to confirm this hypothesis.

4.3.2 SpecJVM

There is no network related program in SpecJVM; therefore, we were surprised to see both disk and network I/O related *e-c links* found by our analysis.

After code inspection we discovered that SpecJVM has a dedicated I/O package that is shared among all the benchmark programs. All the I/O requests are handled in this package; requests can be fulfilled by reading files either on a local disk or on a remote HTTP server. Input data is read from HTTP server when the benchmark is running as a Java applet; otherwise data is read from local disks. When the program is running as a Java applet, it is either enclosed in some web browser or in a *Java Applet Viewer* that is provided with the Java JDK. In either case, unfortunately, we failed to set up the current implementation of the fault injection system to perform fault injection targeted

solely on the applet, without affecting the enclosing program: either the Web browser or the *Java Applet Viewer*. Thus, we could not cover the network-related *e-c links* without changing the code in the SpecJVM slightly. We discovered that `spec.harness` package maintains an `SpecBasePath` variable which is the base location of SpecJVM itself. The value of `SpecBasePath` is set to a remote URL when SpecJVM is running as a Java applet. We modified 7 lines of source code in the benchmark to keep the value of `SpecBasePath` as a URL pointing to a remote file so that I/O requests are fulfilled through network access, even when SpecJVM is running as a stand-alone Java program. This enabled the network-related *e-c links* to be covered.

Even after this process, as can be seen from Table 4.3, we still cannot cover a large portion of the *e-c links* whose `try` blocks have been reached; 87% of these *e-c links* belong to category 3, specifically: infeasible *e-c links* that may be eliminated using context-sensitive object renaming.

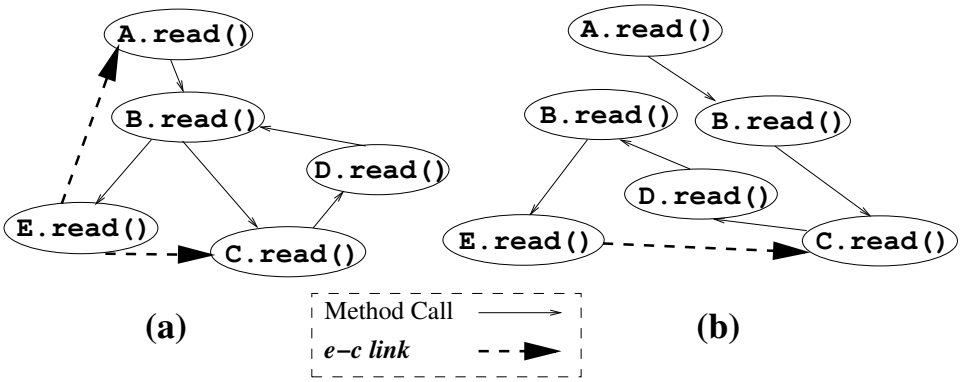


Figure 4.3: Recursive Call Graph

The call chains corresponding to these 26 *e-c links* share a pattern. We use a simplified example to illustrate this for better readability. Consider call chain: `A.read()` → `B.read()` → `C.read()` → `D.read()` → `B.read()` → `E.read()`. The fault-sensitive operation is `E.read()` and when executed, it will throw an `IOException` if an appropriate fault is injected. There are `try-catch` clauses in both `A.read()` and `C.read()` that catch `IOException`. The two outgoing edges from `B.read()` come from a single polymorphic call site. The call graph and the generated *e-c links* are shown in Figure 4.3 (a). The

e-c link from `E.read()` to `A.read()` is infeasible, because the actual points-to relationship between objects in the program causes the call chain `A.read()` → `B.read()` → `E.read()` to be infeasible. If method `B.read()` is analyzed context-sensitively for each of its callers, as shown in Figure 4.3 (b), it may become possible to compute more precise *e-c link* information.

Four of the *e-c links* are in category 2 (i.e., hard to prune by any static analysis). As mentioned above, SpecJVM specifies its input files using URLs, which are string values, containing information about where and how to open and read the files. Loading a file specified by URL introduces huge number of possible ways to load data, depending on the protocol contained in the URL. However, it's hard coded in SpecJVM that only two kind of protocols are permitted: "file" and "http" (i.e., URLs start with "file://" or "http://"), which makes the *e-c links* introduced by other protocols infeasible.

4.3.3 HttpClient

Control flow in HttpClient is complicated. Many control-flow decisions depend on values of string variables (e.g., protocol names, HTTP response code and data encoding method names). In this benchmark, 10 *e-c links* fall into category 1: feasible but we do not have sufficient tests to drive the program into the specific control paths for these *e-c links*. For example, when some connection object is to be recycled (i.e., closed and reused for another host), HttpClient will try to read over the network **only if** the previous HTTP response on this connection is encoded as *chunked*, **and** the previous response content is not fully consumed. So the *e-c link* from a network read to the `catch` block in the network connection recycling method is feasible. Unfortunately none of our tests fits this scenario. More carefully designed tests and specialized HTTP responses are needed to drive the program into different control-flow paths in order to cover these 10 links.

There are 24 *e-c links* in category 2 which account for 60% of all inspected *e-c links* in HttpClient. Recall that this category includes infeasible *e-c links* that are hard for any static analysis to prune. In many tests of the HttpClient package, the HTTP requests and responses are faked in the local memory instead of being sent and received

through network. This is done so that some functionality of HttpClient which does not necessarily involve I/O operations can be tested quickly. A special HTTP connection class is defined for this purpose. In general, yet another network connection will be established if the connection uses a secured protocol (i.e. `https`) and a proxy server is specified in the connection properties, even if the current connection is already *opened*. It is hard coded in these tests that the special HTTP connection class never uses secure protocol or any proxy server in order to avoid real I/O operations. However, even the most precise flow- and context-sensitive static analyses assume that all paths in the control flow graph are executable; thus, in general static analysis cannot recognize the infeasibility of such paths (i.e., paths due to complex control-flow) and consequently it cannot eliminate the resulting *e-c links*.

Significant portions of the inspected *e-c links* fall in category 2 in Muffin(43%) and SpecJVM(13%) too. All of these *e-c links* correspond to infeasible control-flow paths, when the infeasibility of these paths cannot be recognized by static analysis.

There are 6 *e-c links* of HttpClient in category 3: they may be eliminated using V-DataReach, or a context-sensitive object naming scheme. An example extracted from code related to these *e-c links* is previously showed in Figure 3.7 and discussed in detail in Section 3.3.2.

4.3.4 Vmark

By testing these benchmarks, we found that the tests and/or input data that came with HttpClient, SpecJVM and VMark are insufficient to drive execution into most `try` blocks of these programs. We believe this is the reason why there are so many *e-c links* whose `try` blocks are not reached during our experiments, especially in Vmark. VMark is a web chat server built on top of *Tomcat*[3], which is a Java servlet container. When used as a Java server-side performance benchmark in VMark, many parts of *Tomcat* are not exercised, which results in many of the *e-c links* found by the analysis being unreached by the tests. For instance, in *Tomcat* an operator can change the configuration and force reloading of the affected servlets. Also when *Tomcat* receives a shutdown request, the changed configuration must be flushed to the disk. Because this

part of *Tomcat* is not exercised in VMark, *e-c links* corresponding to the I/O operations necessary to perform these functionalities are left unreached and therefore, uncovered. By examining the call chains of the *e-c links* in VMark, we found that in the *e-c links* whose `try` blocks are not reached, only 3 are related to the chat server code; the call chains of all the other *e-c links* are completely within the *Tomcat* code. In the 18 reached *e-c links*, 13 *e-c links* are related to the chat server. Thus, a significant portion of *Tomcat* is left unexercised in VMark.

Overall from the results of these experiments we can see that 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 testing methodology allows developers of fault-tolerant server applications to quantify (and improve) the coverage of fault-recovery code, as is done with any other code subjected to white-box testing.

Chapter 5

Visualization of *e-c links*

As already discussed previously, the Java programming language provides a program-level exception handling mechanism in response to error conditions that happen during program execution. This exception handling mechanism helps separate exception handling code from code that implements functionalities during normal execution, which, to some extend, helps program understanding.

However, exception handling code that deals with certain kinds of faults is still widely scattered over the whole program and mixed with other exception handling code, or even irrelevant code, making it hard to understand the behavior of the program under certain system fault conditions.

We have shown that the static program analyses described in Chapter 3 can be used to report *e-c links* in a given Java program with very good precision, which can be used by the exception def-use testing system. In addition to that, the information produced by these analysis, if carefully organized and visually displayed in an integrated development environment (IDE), can greatly facilitate both testing and program understanding of the exception handling code. We developed an Eclipse plug-in – *ExTest*, which invokes these analysis and organizes the output data into tree views for this purpose.

5.1 Problem of Manual Inspection of Exception Handling Code

During the study, we found that exception handlers that deal with certain kinds of faults are often scattered in the program and mixed with handlers that handle other kinds of error conditions. For instance, a `catch` clause that handles an I/O exception may appear at each program point where some I/O channel is active. Each of these `catch`

clauses may handle I/O exceptions triggered by different fault-sensitive operations (e.g. DSK_READ or NET_READ). Worse, some of these `catch` clauses never handle any I/O exception.

In Figure 5.1 we show a small Java program – a single class containing the `main` method calling all of these three methods (also defined in this class). Note that although these 3 methods looks similar, the `catch` clause in the method `readString` will **never** be triggered. The reason is the code in the corresponding `try` block only reads from a string buffer in the memory. Although it takes the form of an input stream, no actual I/O operation is involved. Yet the `try-catch` structure is necessary for the program to compile.

```
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){ ...}
}

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

Figure 5.1: Code Example for Java I/O Usage

If a programmer wants to learn this program's behavior under disk failure, she has to find all the `catch` clauses that may handle exceptions that result from disk faults. Suppose a powerful lexical search tool with Java language knowledge as well as program specific information (e.g. types) is available. Then she can easily locate all the `catch` clauses that handle `IOException` or more general types of exceptions, but she still has to manually inspect at least all three `try-catch` blocks in all of the methods shown in Fig. 5.1, instead of just the one in method `readFile` that actually handles the exception result from disk failure. The problem becomes much more severe in real Java server applications.

5.2 ExTest Tool

Using the analysis mentioned in Chapter 3, we can compute all the potential *e-c links* of the program. Each *e-c link* (p, c) tells us the fault-sensitive operation that triggers the exception and where it is handled. Thus, we can help solve the above problem by grouping *e-c links* according to their p value. For instance, one can just browse *e-c links* starting with fault-sensitive operations that relate to disk I/O to get a good estimate of all the `try-catch` blocks that are related to disk I/O.

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

5.2.1 Tool Structure

Figure 5.2 illustrates the structure of the *ExTest* tool. Our program analysis is implemented as modules in the Soot Java Analysis and Transformation Framework [59]

version 2.0.1. Upon user request, *ExTest* starts another process running Soot with our modules enabled, and reads the output data of the Soot modules after the process finishes.

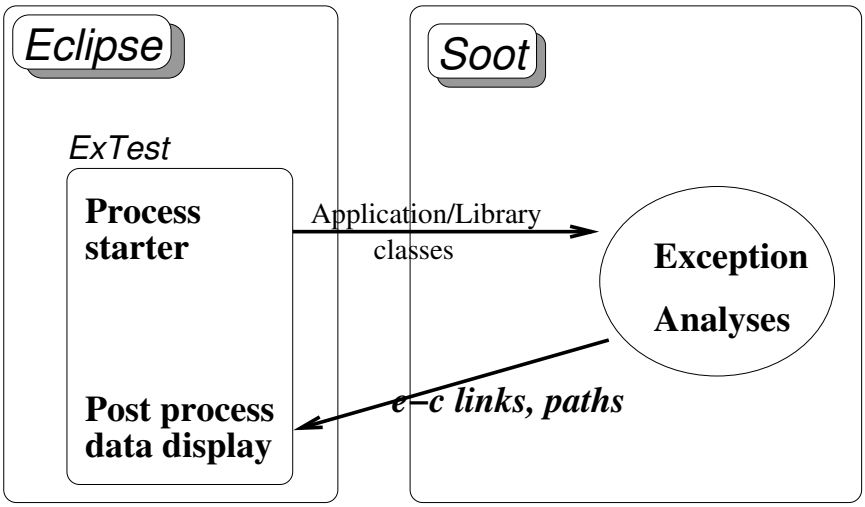


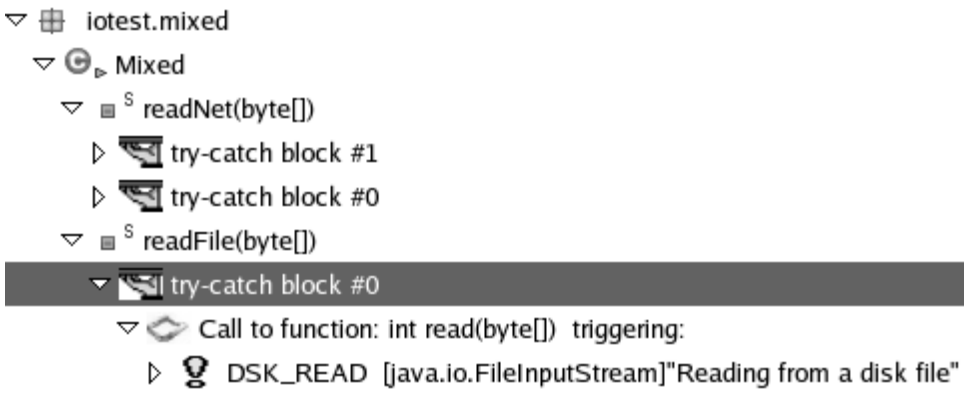
Figure 5.2: Tool Structure

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

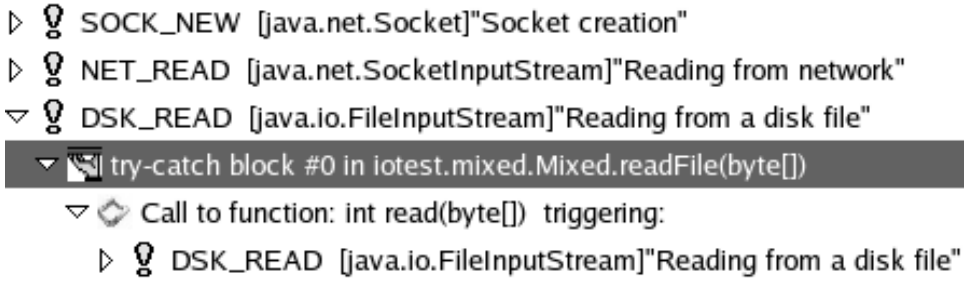
5.2.2 Browsing *e-c links*

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

Figure 5.3(a) shows the *Handlers* view, where *e-c links* are grouped by the `try-catch` blocks. These `try-catch` blocks are further grouped by their definition positions: the



(a) *Handlers* View



(b) *Triggers* View

Figure 5.3: Tree Views of *e-c links*

methods, classes, packages in which they are defined. Each `try-catch` block can be expanded to show all the fault-sensitive operations that may trigger exceptions reaching the `catch`. The last `try-catch` block in the figure is highlighted and expanded. It is defined in package `iotest.mixed`, class `Mixed` and method `readFile`. We can see that one method call in the `try` block reaches a fault-sensitive operation in the JDK: "DSK_READ".

Figure 5.3(b) shows the *Triggers* view, where the *e-c links* are grouped by the fault-sensitive operations. By expanding the "DSK_READ" operation we can see that only one `try-catch` block in the program handles an exception thrown by read of a file. So if a user is interested in program behavior under a disk fault, she can just concentrate on this one `catch` block.

Thanks to the environment provided by Eclipse IDE, these two tree views can be interactively explored. The `try-catch` block, the statements in the `try` block that may

lead to the fault-sensitive operation, etc., can be opened and highlighted in the Java source file editors, upon double click on the corresponding items in the view. For example, in both views, we can see the actual code for the `try-catch` block #0 by double clicking on the line.

5.2.3 Displaying All Paths for an *e-c link*

We also want to display the paths that show how p in an *e-c link* (p, c) can be reached from the `try` block that corresponds to c. Selecting and displaying only one (the shortest) path for each *e-c link* is not enough, especially with the presence of the false positives. In order for a programmer to decide that an *e-c link* is spurious, she has to make sure that **all** the control-flow paths from the corresponding `try` to p are actually infeasible. So it is necessary for *ExTest* to display **all** these paths to be practically useful. But the total number of paths may be exponential to the size of the program [8]! Clearly, the approach of gathering and dumping all these paths into an output file after the analysis finishes will not scale.

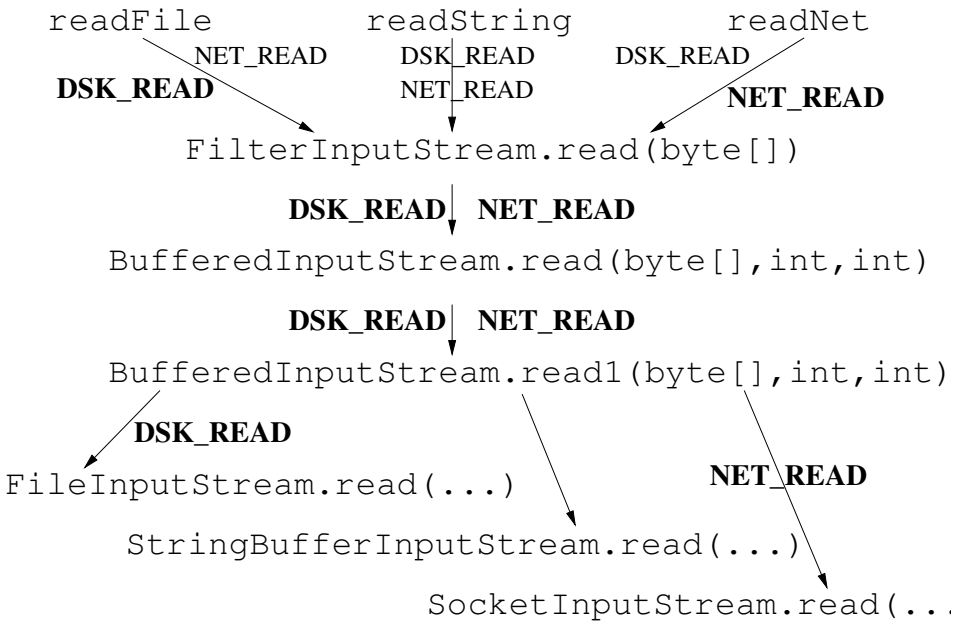


Figure 5.4: Annotated Call Graph

Note that the *Exception-flow* analysis described in Chapter 3 identifies *e-c links*

by essentially propagating fault-sensitive operations along the call edges in a reverse direction to execution. It is not hard to modify *Exception-flow* analysis to record the propagation paths of each $p \in P$ by annotating call edges in the call graph. Figure 5.4 shows the annotated call graph for the code in Figure 5.1. Edges of the call graph are annotated with IDs of the fault-sensitive operations according to the result of *Exception-flow* analysis. Since the set of fault-sensitive operations P is pre-selected according to the fault set provided by the user (not depending on the program being analyzed), the size of the annotated call graph is at most linear in the size of the original call graph.

The problem with this approach is that the results of *DataReach* are ignored. As stated before, *Exception-flow* analysis alone would leave too many false positives in the graph; with this data, the user must manually explore many unnecessary call edges to decide that a certain *e-c link* is infeasible. So we need to take the advantage of *DataReach* to reduce this workload.

Recall that *DataReach* proves that some of the *e-c links* are infeasible by showing the infeasibility of the all the control-flow paths supporting these *e-c links*. To be able to incorporate its results into the annotated call graph, we modified *DataReach* so that for each *e-c link* (p, c), the annotations of p on all the call edges associated with (p, c) are *confirmed* only if we *cannot* prove the infeasibility of (p, c). During the output of the call graph, only the *confirmed* annotations are written with the graph. In Fig. 5.4 confirmed annotations are shown in bold face.

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

Figure 5.5 is the expanded view of the last *e-c link* shown in Fig. 5.3(b). Only one witness path was discovered by the analysis, which precisely reflects the analysis result shown in Fig. 5.4.

Figure 5.5: Exception Propagation Path

However, we are not always so lucky in bigger programs; paths in these programs can get very complicated, especially inside the JDK library classes that make heavy use of polymorphism. Figure 5.6[1] shows part of the *Triggers* view displayed when browsing *e-c links* in one of the testing benchmarks used in Chapter 4 – a FTP server written in Java [68]. Witness paths of one *e-c link* are partly expanded in the figure, with the fault-sensitive operation `SocketInputStream.read()` highlighted.

As can be seen from the figure, the "fan out" of some of the nodes along the paths is large (e.g., `InputStreamReader.close()`). Furthermore, many of the methods appear more than once, which indicates the possibility of recursion introducing a path with unbounded length. Since these paths are extracted out of a call graph, expanding the second appearance of a method on a path brings exactly the same set of children in the tree view. This is wasteful and introduces unnecessary complexity into the view. Manually identifying the recursion in a complex view like this is not trivial. Therefore we have automated recursion detection in *ExTest*. As shown in Fig. 5.6, many methods are annotated with "..." and they are **not** expandable, which shows that the method has been called recursively and further expansion is not necessary.

If we only show only one witness path of the *e-c link*– the natural selection would be the shortest one – the view can be greatly simplified, but the real complexity of the

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

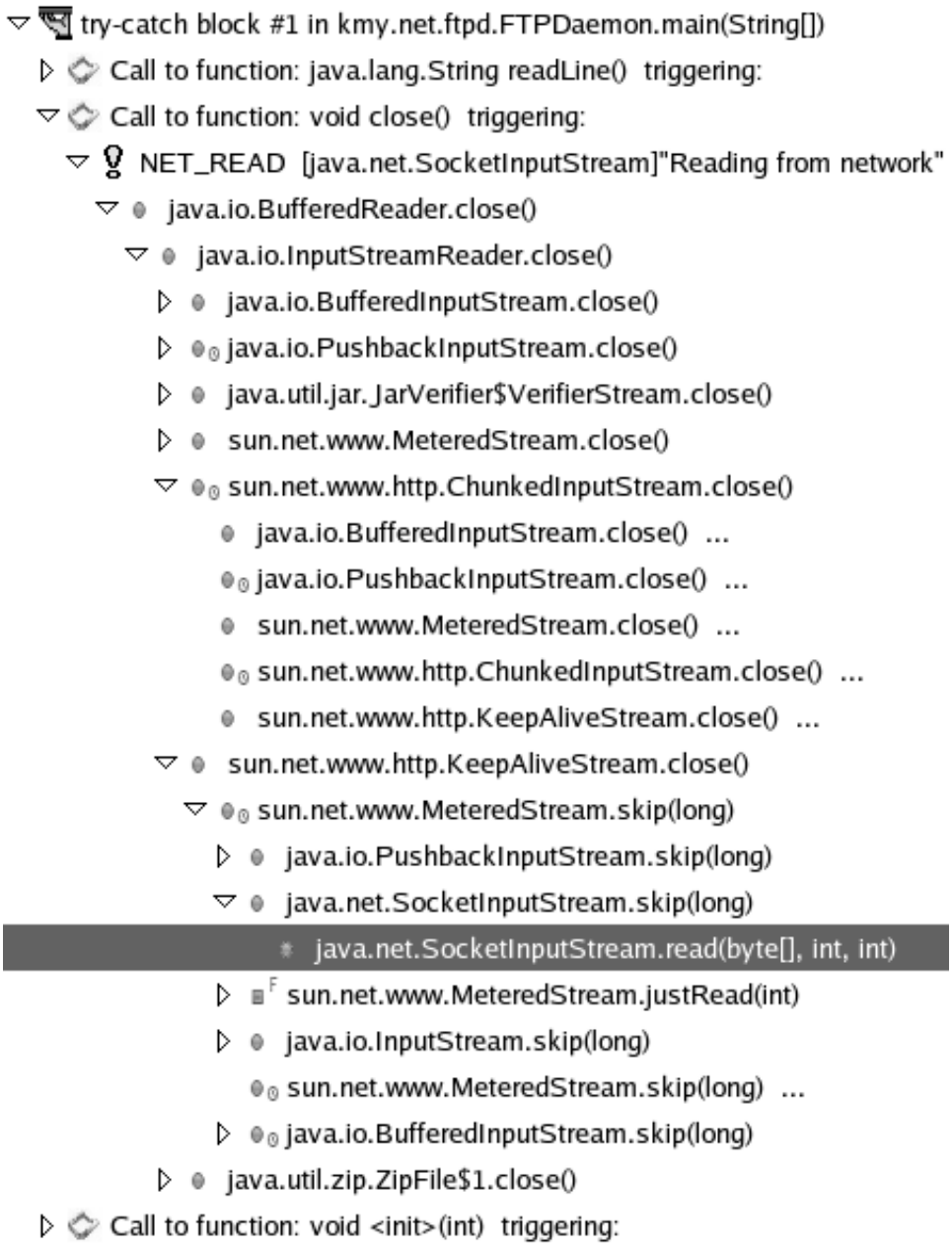


Figure 5.6: Exception Propagation Path

problem would be hidden from the user: the user would not be helped in identifying infeasible paths; however, expanding and highlighting the shortest path automatically among all paths may help in understanding the overall program structure.

5.3 Summary

ExTest is a tool that facilitates navigating code related to the exception handling feature of Java programs, based on exception def-use analysis discussed in Chapter 3. We want to reveal all information needed to the user, while carefully organizing the data to help browsing and reasoning.

Despite of our current efforts, exploring program code based on conservative static program analysis results can be difficult (see Figure 5.6). One way to alleviate the situation is to use more precise (but possibly more expensive) analyses to eliminate more spurious results.

Chapter 6

Exception-chain Analysis

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 goes by without logging.

The analysis in Chapter 3 can be used to reveal the *e-c links* in a Java program (i.e., `throw`, `catch` pairs with chains of calls between them) with relatively high precision. 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? It would be nice if this information can be used to help understand exception handling behavior of a module-based system.

Our first attempt was to build a graph out of these *e-c links* to review the overall exception handling structure of the whole system, thinking that the *e-c links* reported by the program analysis represent the propagation paths of exceptions in the program. But we found that the analysis that discovers the *e-c links* cannot capture the behavior

of one of the common practices in exception handling – rethrow of caught exceptions. This chapter discuss an analysis which identifies these exception rethrows and thus finds chains of semantically related *exception-catch links*.

6.1 Exception Catch and Re-throw

Shenoy mentions the following as "some of the generally accepted principles of exception handling" in [63]:

1. If you can't handle an exception, don't catch it.
2. If you catch an exception, don't swallow[1] it.
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.
5. Structure your methods according to how fine-grained your exception handling must be.
6. Use as many typed exceptions as you need, particularly for application exceptions.

Reimer and Srinivasan [53] also point out that a "large distance between throw and catch" may make debugging more difficult. 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 [72], 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.

Figure 6.1 shows a `catch` clause that is slightly simplified from a real one found in MySQL Connector/J 2.0.14[46] – a native Java driver that converts JDBC (i.e., Java

[1] An exception is *swallowed* if no use is made of the exception object in the `catch` clause.

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

Figure 6.1: Caught Exception Rethrow Example

Database Connectivity) calls into the network protocol used by the MySQL database. 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.[2]

In Java, an exception object contains a snapshot of the execution stack of its thread at the time it was created. In the handler in Figure 6.1, the new exception object only contains the class name of the old one. Thus part of the execution stack – from the method where the old exception was created to the one before the enclosing method of this handler – is lost. As an alternative, enclosing the old exception object into a new object can preserve the opportunity to reconstruct the whole stack if some problem occurs at runtime. But as mentioned in [53], it is not always a good idea to keep all the stack information. During a load surge, if we try to log the entire stack in the final handler, it may do as much harm as good, because with system resources already very low, they may not be sufficient to allow the task to complete.

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, including the algorithms in Chapter 3, cannot connect these closely related *e-c links* into a chain, which makes it difficult to trace back to the root cause of the exception given its final handler. Because reconstructing the whole stack in the final handler is not always possible (or desirable), an programmer trying to diagnose

[2]In the remaining discussions, 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.

and repair a system degradation (or crash) may have very limited information to aid in determining the source of the problem. What's more, if the actual exception flow is a chain spanning many software layers in the system, the testing framework in Section 2.2 is limited to exploring only individual segments of this chain.

6.2 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`. This information can be used to illustrate all exception flows in the entire system, especially those flows across different components, thus revealing the exception handling architecture of the system.

6.2.1 *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 within the `catch` clause.

When the Java code shown in Figure 6.1 is translated to bytecode, each statement in the source code will be broken down into multiple simple bytecodes. A Java bytecode analysis tool can translated these bytecodes into the sequence of expression statements shown in Figure 6.2 to facilitate further analysis and optimization. We are using Soot [59] for this translation. In the translation, `@caughtexception` represents the reference to the caught exception in the `catch` clause and `<init>` signals a call to a constructor.

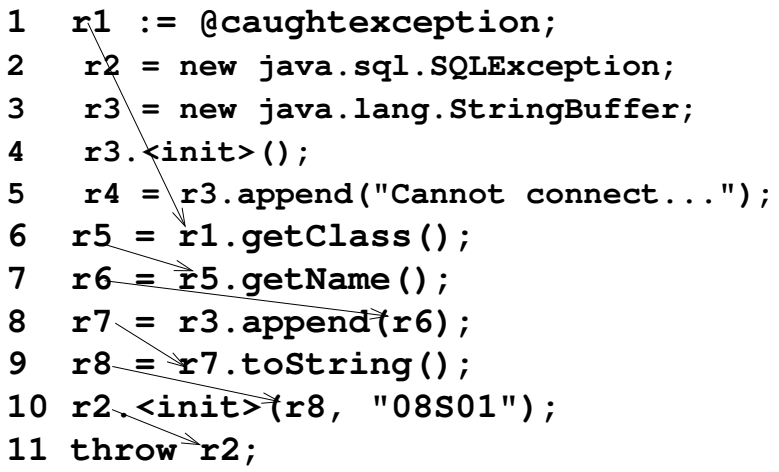


Figure 6.2: Exception Rethrow Bytecode Representation

Each arrow shown in Figure 6.2 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.

This process of variable usage tracing can be automated. Figure 6.3 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

```
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 6.3: *Handler-inspection* Analysis Algorithm

variable.[3] A variable is considered to be *defined* only when it appears on the left-hand-side of an assignment operator. As a consequence of choosing to do a local analysis, we make conservative assumptions at method calls; that is, at a method invocation, the receiver and all the actual parameters are considered to be *defined* by the call statement. However, we give special treatment to string and exception manipulation methods. For example, receiver and actual parameters of method `StringBuffer.append()` are not

[3]The first assumption is satisfied by the way Java bytecodes are defined [39] and the way they are translated into Soot internal representation. The second function relies on the def-use analysis provided by Soot [59].

considered to be defined, but only used.

In Figure 6.3, 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 6.2. 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., it allows false negatives).

6.2.2 *E-c chain* construction

Both *Handler-inspection* analysis and the *Exception-flow* analysis in [24] 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. Recall that Soot includes an intraprocedural reaching definition analysis that provides local def-use links. We modified it to fit our needs by assuming each reference parameter may be modified in a method invocation.

After obtaining both *e-c links* and *interconnecting points*, it is easy to construct *e-c chains* $(p, c, p, c, p, c...)$ 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 `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, giving an estimate of the *vulnerability* of certain components and showing the *service dependence relations* between components (see Chapter 7). 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 Chapter 2, the quality of the recovery code can be tested in advance of installing the web service application.

6.2.3 Testing of *E-c chain*s

In Chapter 2 a testing framework is introduced, in which *e-c links* can be tested one by one: for a given *e-c link* (p, c), the corresponding `try` block is instrumented so that during runtime, the fault injection engine is informed to trigger an exception if p, which is usually an operation that may trigger some environmental exception, is executed. This is to make sure that the exception thrown from p is actually handled in c (i.e., the given link is exercised by the test). Entry point of c is also instrumented to record which *e-c link* (p, c) is actually exercised during runtime.

This testing framework is extended to help cover *e-c chains* reported by the *e-c chain* analysis. As we know an *e-c chain* $(p, c, p, c...p, c)$ 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. Thus a thread local stack can be 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*. At the same time, each entry point

of c is also instrumented to record the execution of the `catch` clause and also the calling stack embedded in the caught exception object. These data can be easily processed offline to reconstruct the *e-c chain* exercised during test run.

Note that our original *e-c link* testing framework relies on an extended version of an existing fault-injection engine *Mendosus* [38], which can only inject I/O related faults into the system. So currently our testing system can only test *e-c chains* originated from an I/O operation that may trigger some I/O operation.

The extended testing framework can collect the following information during the testing process: all the possible *e-c chains* in the program and those that are exercised during the test. Based on these, we define two different testing coverage metrics, *Chain Coverage* and *Link Coverage*.

- *Chain Coverage*: Given a set FC of the possible *e-c chains* of a program, and a set EC of the *e-c chains* that are experienced during a set of test runs, *Chain Coverage* is defined as $\frac{|EC|}{|FC|}$.
- *Link Coverage*: Given a set FL of all the *e-c links* decomposed from *e-c chains* from FC, and a set EL of the *e-c links* $\in FL$ that are experienced during a set of test runs, *Link Coverage* is defined as $\frac{|EL|}{|FL|}$.

Chapter 7

Experimentation on Exception-chain Analysis

In this section we report our empirical findings and discuss a case history from our experiments, whose goal was to demonstrate the effectiveness of our methodology. The case history about Tomcat demonstrates the complexity and the inter-component nature of the *e-c chains* determined by our analysis. Some of these discussions appeared earlier in publications [26].

7.1 Experimental setup & benchmarks

We implemented the analysis in the Java analysis and transformation framework Soot [59] 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:

- Muffin, a web filtering proxy server (http://muffin.doit.org/).
- SpecJVM, a standard benchmark suite that measures performance of Java virtual machine (http://www.spec.org/jvm98).
- VMark, a Java server side performance benchmark. It is based on *VolanoChat* – a web based chat server (http://www.volano.com/benchmarks.html).
- Tomcat, a Java servlet server from the *Apache Software Foundation*, version 3.3.1 (http://tomcat.apache.org/). The servlets application running on top of Tomcat is an online auction service modeled after EBay – part of the DynaServer project [54] at Rice University. This application communicates with MySQL database using MySQL Connector/J [46].

- 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 7.1 shows the sizes of the benchmarks. Spark, a points-to analysis based call graph constructor provided with Soot[59], 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 7.1: 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

According to the size of the *.class* files, Muffin is significantly smaller than the other four benchmarks. It contains a smaller number of handlers than the other benchmarks. VMark, Tomcat and HttpClient are composed of many components, identified by multiple *jar* files in the distribution.[1]

The reason we are including the relatively small and simple Muffin as one of the benchmarks is that despite of its size, according to data presented in [24], the number of *e-c links* involving I/O found in Muffin is comparable to the other larger benchmarks. Moreover, it takes a rather expensive analysis to remove a significant portion of false

[1]We recognize components by assuming one component per *jar* file provided by each benchmark. Users of our analysis can override this by providing the component membership of classes according to a provided XML schema. There is no *jar* file defined in Muffin or SpecJVM.

positive *e-c links* in Muffin produced by the cheaper analysis, which we believe shows that its recovery code structure is relatively complex.

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 [24]. (Recall this analysis does not execute at runtime.)

7.2 Catch Clause Categorization and E-C Chains

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 7.1 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 benchmarks, the percentage of handlers that rethrow exceptions ranges from 15% to 35%, something that we *can not* ignore. But such activity is not very visible in Muffin: only about 2%. 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.

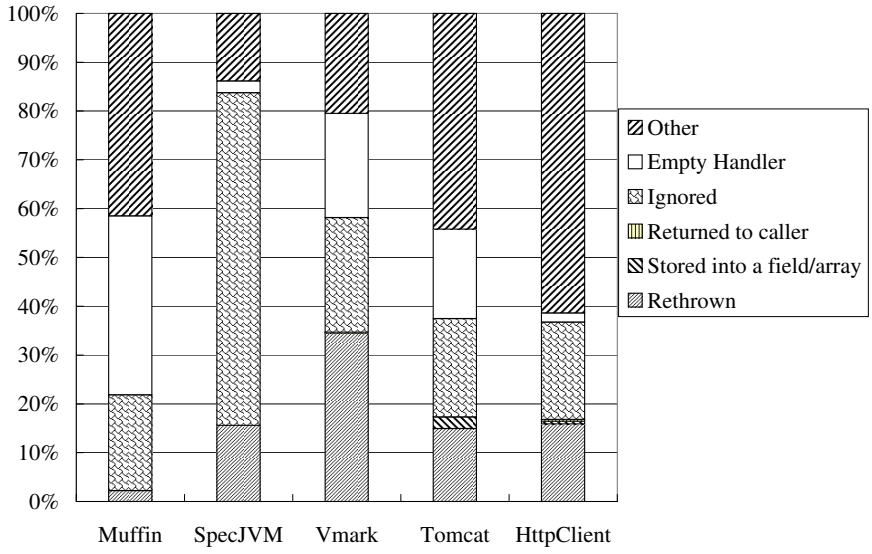


Figure 7.1: Usage of Caught Exceptions in `catch` Clauses

Not surprisingly, all of the handlers in category (vi) contain invocations to other methods with information from the original exception used as either the receiver or a parameter. The reason we did not name the category *method calls* is that handlers in category (i), (ii) and (iii) also may make such method calls. Figure 7.2 shows the kinds of method calls that appear in all of 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. We can see that most of the time the *Handler-inspection* 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, (i.e., 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. Handlers that directly call printing or logging functions dominate in 4 out of 5 benchmarks (i.e., except for SpecJVM).

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 the *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).

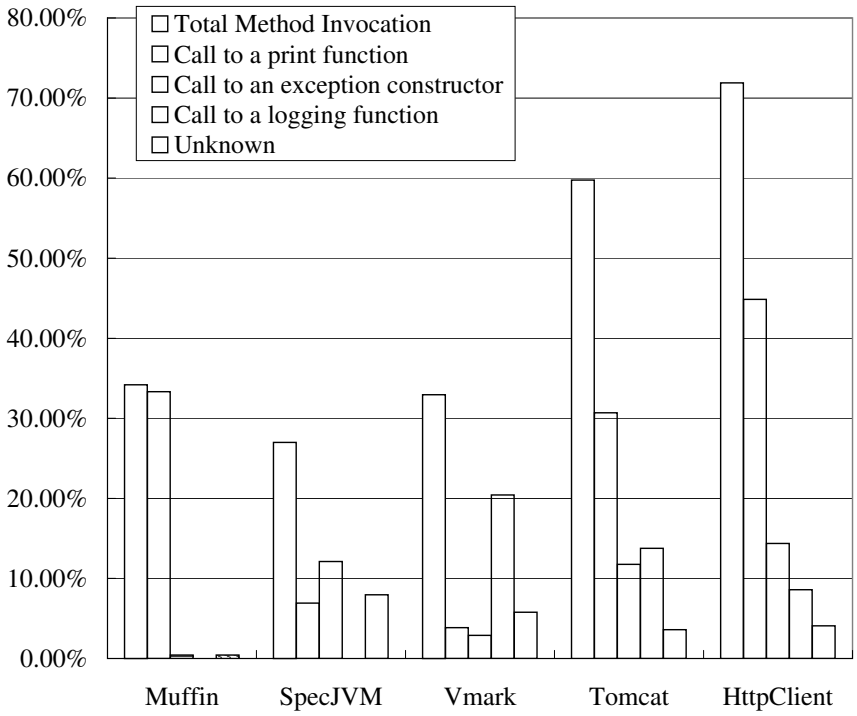


Figure 7.2: Methods Calls Related to Caught Exception

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 than those in a `catch` clause that can not be reached by any checked exception.

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 7.3 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.

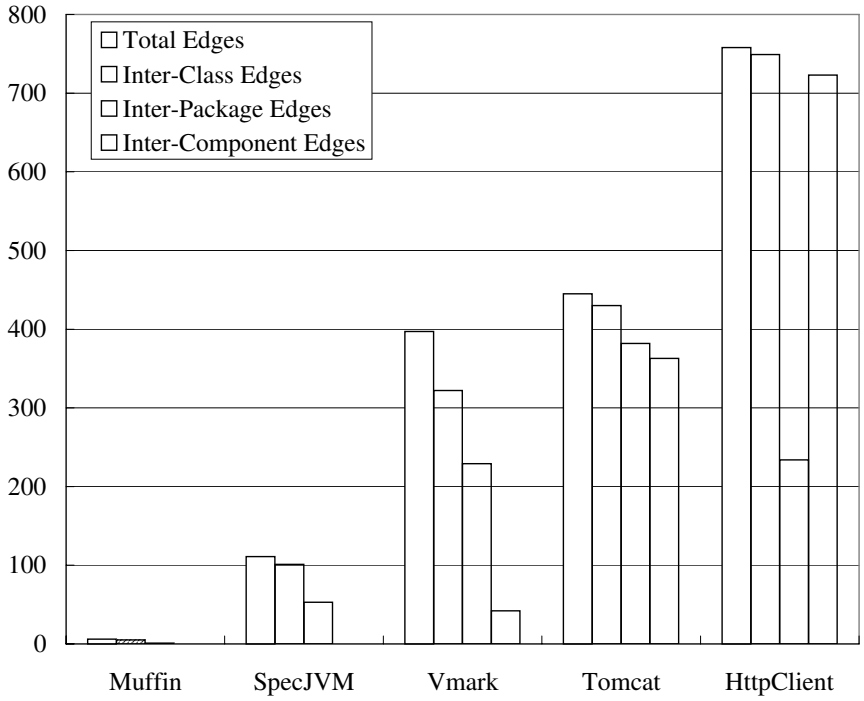


Figure 7.3: Number of *e-c links* Starting from a Rethrow

One interesting fact about HttpClient is that 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 and rely on their caller to handle them.

Table 7.2: Number of Chains of Difference Length

Length	**1**	**2**	**3**	**4**	**5**	**6**	**Total**
Muffin	6						6
SpecJVM	69	46					115
VMark	300	81	12				393
Tomcat	312	365	31	3	2	10	723
HttpClient	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 7.2 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 7.2, 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; then the number of *e-c chains* will be 4.

From the data presented above we can see that in Muffin, although the number of I/O related *e-c links* is not very small as shown in Chapter 4, the *e-c links* are fairly independent from each other. At the same time, in all the other benchmarks, 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. Thus, we can reveal the whole exception propagation path, instead of just discrete segments of it. As often these paths go across different components, a programmer diagnosing the root cause of a problem 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.

7.3 E-C Chains in Tomcat

The data presented above, especially the long *e-c chains* found in Table 7.2, 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? Are the* e-c chains *mostly independent or tangled together? What can these* e-c chains *tell us about the overall exception-handling behavior of the system?*

7.3.1 Precision

We are primarily interested the precision of recognizing *interconnection points* in all the `catch` clauses. As mentioned previously, 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 result 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.

There is another `catch` clause in the Tomcat Facade package that calls a method which throws its parameter directly. All of these rethrows happen in the method directly called from the handler, not in other methods that are reachable from the callee.

According to Java library API specification [72], "A throwable contains a snapshot of the execution stack of its thread at the time it was created." In one of the above methods, a new exception was created that wraps the original exception and then is thrown. Since it is *not* created "on the spot" (i.e., within the `catch` clause, as most exceptions are), this exception object contains a stack snapshot that takes a little "detour" from the original exception propagation path. If this snapshot is logged by the final handler and subsequently used for problem localization, the "detour" may become a source of confusion. In the other method mentioned above, since the original exception was rethrown, the original stack snapshot was preserved. But in both cases, the handling complicates the program understanding task by keeping the `throw` site further away from the problem path, which may present difficulties to system diagnosis, especially when the call stack is *not* completely logged in the final handler due to error-handling-time system resources concerns.

We may also 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 some call path associated with each *e-c link* is 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 Section 3.2 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 due to the interprocedural part of the analysis. 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.

7.3.2 *E-c chain* Graph.

The *e-c chains* can be depicted in a graph and shown in differing granularity to help in different tasks. *e-c chains* represent exception travel paths in the system, and as observed in our experiment, often span several components. 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. This can increase confidence in the expected robustness of the application when problems occur.

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. In program understanding tasks, the component-level exception-flow structure can help a system integrator better understand the interaction between components of an application. This structure also can increase confidence in the expected robustness of the application when problems occur.

We manually collected all the *e-c chains* with length greater than 2 and displayed them in Figure 7.4, which shows the exception-flow architecture of the Tomcat 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 7.4, we can easily make two observations. First, on the left-hand-side of the graph, MySQL Connector/J relies on Java network

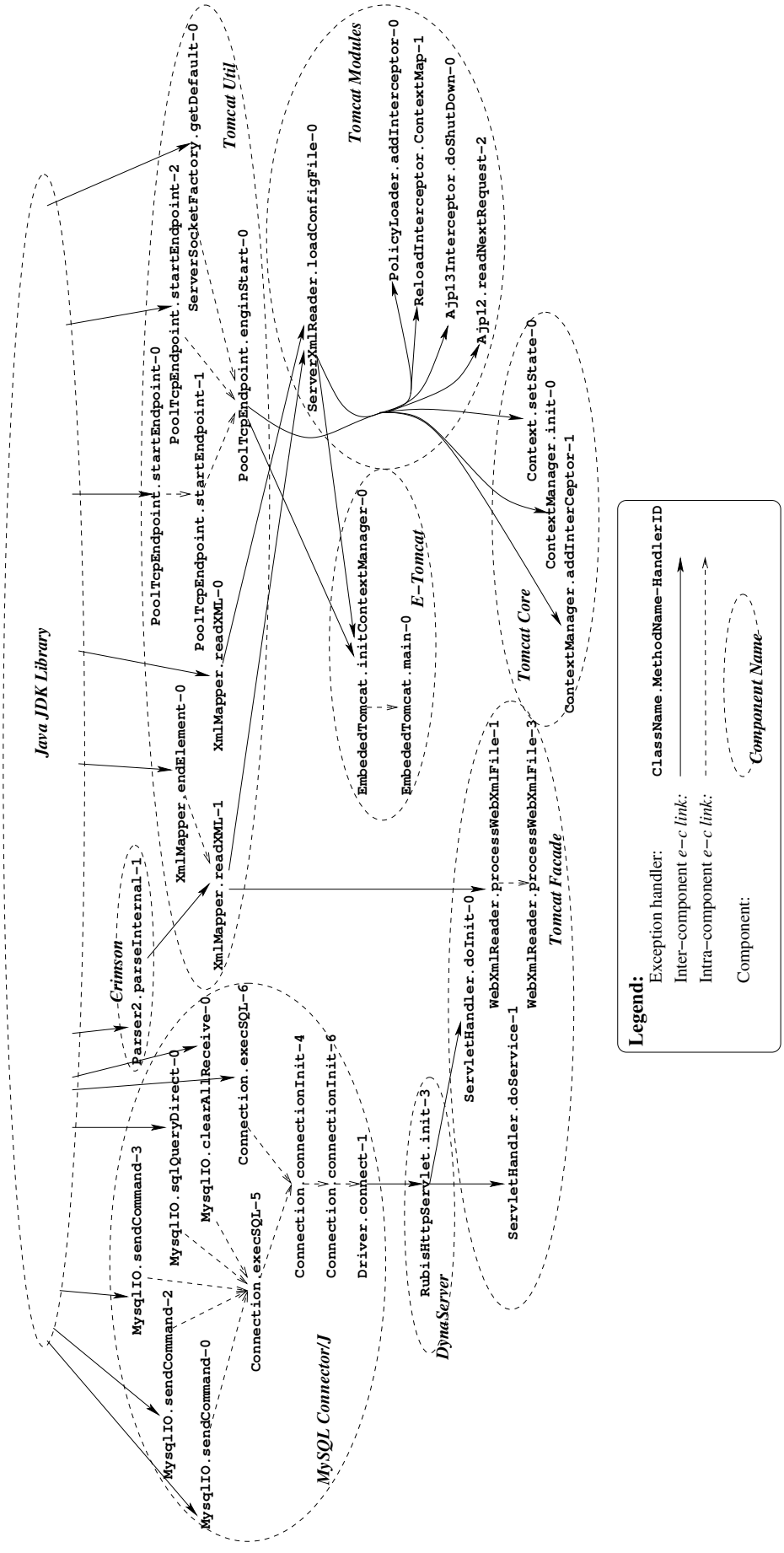


Figure 7.4: *E-c chain* Graph of Tomcat

library to communicate with the MySQL database, and 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 other non-Facade parts of Tomcat are very likely not to be affected. In this sense, the Facade component serves as a good firewall between the servlet application and other parts of Tomcat, and thus is identified as a critical component in error recovery. If this component is well tested to handle/log exceptions properly, the whole system will likely to handle/log errors properly. Second, according to the structure on the right-hand-side of the graph, the system is a lot more vulnerable to I/O problems during start up, because operations such as starting a server socket or reading some configuration file fail, and thus may cause trouble in many major parts of the system, including the core component.

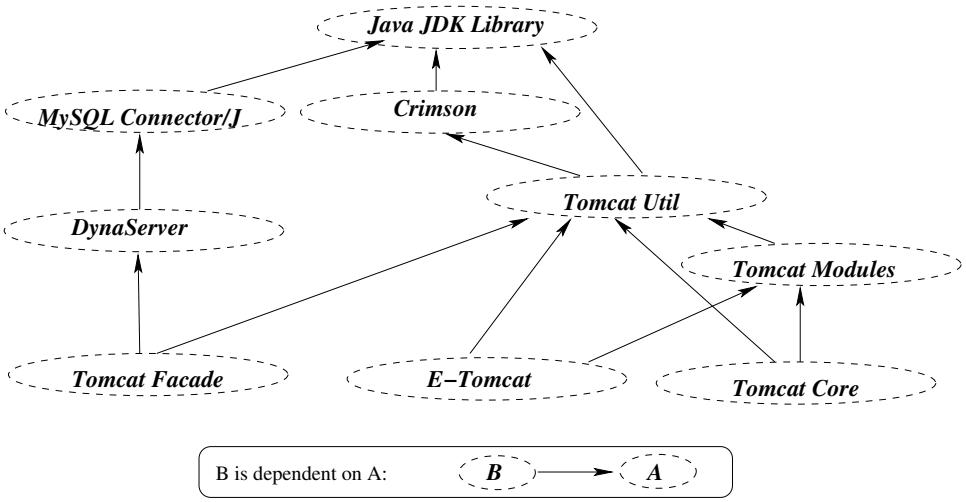


Figure 7.5: Service Dependence Graph of Tomcat

The *e-c chain* graph can also be presented in a coarser granularity to reveal dependences between components, and thus form 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 7.5 is the service dependence graph of Tomcat. For example, the graph tells us that all core Tomcat

components, excluding servlets components running on top of Tomcat, use services provided by component named *Tomcat Util* instead of accessing I/O library directly. This is the component that delegates I/O and provides a higher level of abstraction in both functionality and exception.

The *e-c chains*, when depicted in the graphs in Figure 7.4 and 7.5, 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 shown above without consulting the source code of the system.

7.4 Testing of *E-C Chains* in Tomcat

We implemented the *e-c chain* testing framework as described in 6.2.3 and used Tomcat as our benchmark because of the complex *e-c chain* structure discovered in it. Note that our testing framework can only inject I/O faults into the system. So we could only test *e-c chains* originated from I/O operations that may trigger an exception.

In Tomcat, our *e-c chains* analyses found 308 *e-c chains*[2] originated from I/O operations. 16 (about 5%) of these *e-c chains* span 2 different components and the rest of them span 3 or more different components. We believe this is more evidence that exception catch and rethrow is used mainly as means of inter-component exception propagation in Tomcat.

These 308 *e-c chains* can be decomposed to 184 edges, each taking from of an *e-c link*. As you can imagine from the number, some of the *e-c links* are shared among many different *e-c chains*. In the following discussion, the term *sharing degree* denotes for a given *e-c link*, the number of different *e-c chains* that share it. Figure 7.6 shows how *e-c links* are shared among *e-c chains*. On X axis, each vertical bar is labeled

[2]We only consider chains with length greater or equal to 2, because shorter chains are essentially *e-c links* whose testing has already been discussed.

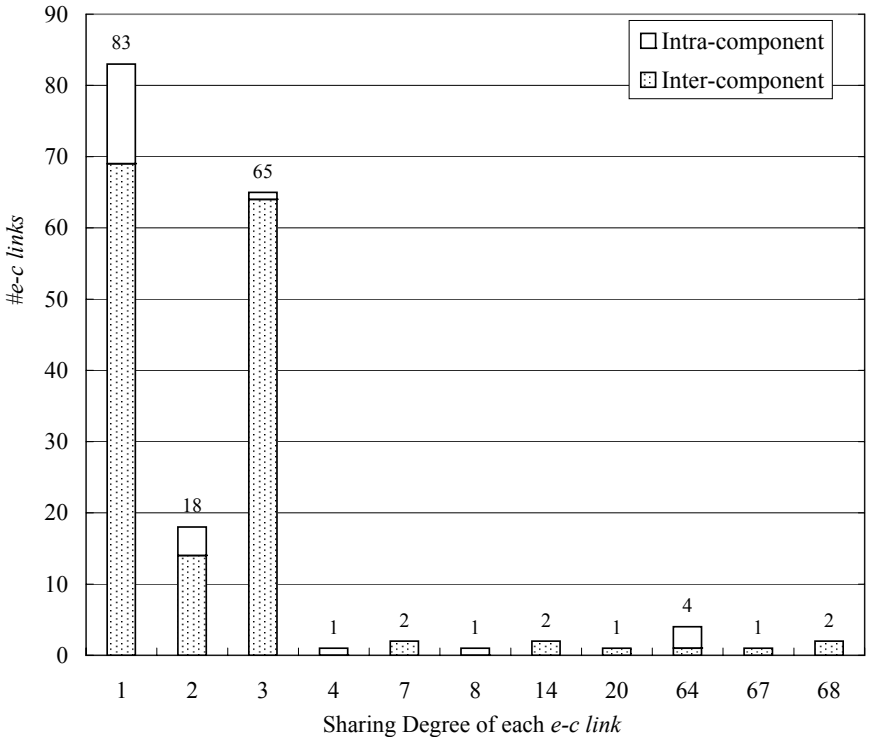


Figure 7.6: Links Shared by *e-c chains*

with the sharing degree of the *e-c links* in this group and the height of the bar (also specified by the number on top of each bar) represents the number of *e-c links* in this group. For example, the rightmost bar tells us there are 2 *e-c links* each of which is shared among 68 different *e-c chains*. Moreover, each bar is divided into two parts to show the number of *e-c links* that are inter-component or intra-component, respectively. We can see from the chart that i) in total, a significant number of *e-c links* have high (equal or greater than 3) sharing degree, ii) a majority of the shared *e-c links* are inter-component. This information should draw testers' attention to the `catch` clauses associated with those highly shared *e-c links*, because they may handle exceptions from many different program points, and often from different modules.

After locating the *e-c chains* in Tomcat, our goal was to use our extended testing framework to exercise the program as much as possible, to cover reported *e-c chains*. Tomcat, with the DynaServer Ebay emulator as servlet application running on top of it, functions as a dynamic content web server. Initially we used the client emulator [54]

that came with DynaServer to generate client requests (as input data to the server) trying to exercise different parts of the server. We only covered 105 *e-c chains* this way. When we tried to inspect the uncovered *e-c chains*, we found feasible program paths that could enable execution of many of these *e-c chains*. So additional client requests were manually crafted to cover more *e-c chains*. Finally, out of 308 *e-c chains* reported, we managed to cover 234 *e-c chains* in total – 76% coverage. Although this client emulator is very successful in testing the server's performance, it does not try to fully explore the server as an test input generator.

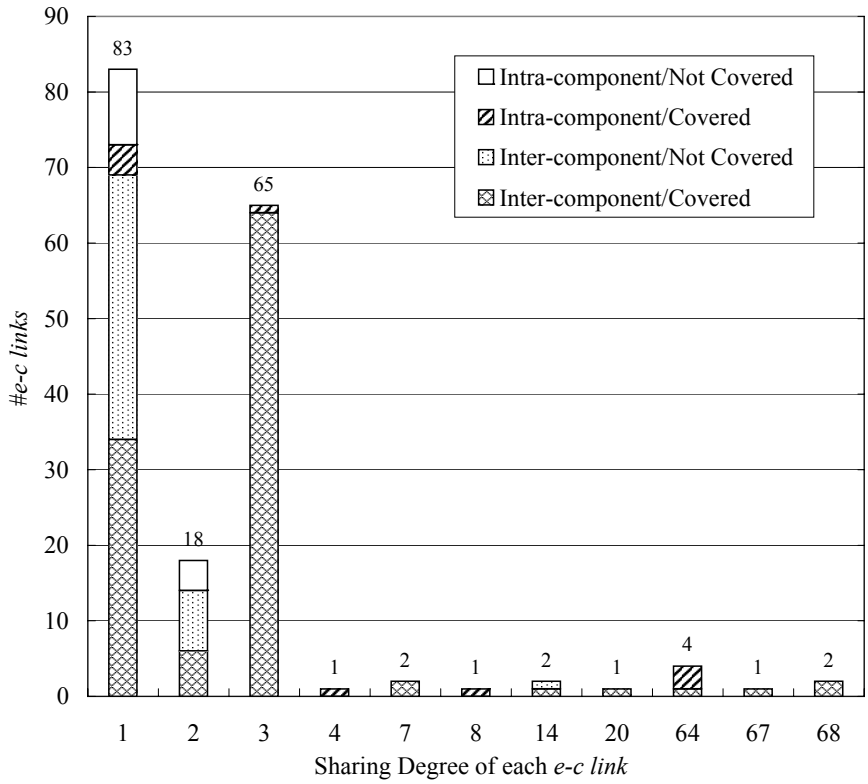


Figure 7.7: Link Coverage

As mentioned above, the 308 *e-c chains* can be decomposed to 184 *e-c links* . Among these, 126 *e-c links* were experienced during the test, 68% coverage. Figure 7.7 is a refinement of Figure 7.6 where *Link Coverage* is shown. The bar chart in Figure 7.7 shows that most of the uncovered *e-c links* have low "sharing degrees". In fact for all *e-c links* shared by 3 or more *e-c chains*, there is only one not covered during the test. This

explains why the *Chain Coverage* is higher than the *Link Coverage* in our experiment. In this case we are lucky in our experimentation because we covered almost all the more important (with higher sharing degree) *e-c links*.

During the test, there were 28 covered *e-c chains* for which after the final exception handlers were reached, either array out of bound exceptions or null pointer exceptions were raised. All of these 28 *e-c chains*' final exception handlers are in the DynaServer servlet application. We believe that these exceptions being raised is an apparent indicator of bugs in the exception-handling code. All of them caused a stack dump to appear on the affected clients' browsers, but none of these exceptions caused Tomcat to crash, nor did any of them stop the servlet application being used by other concurrent clients. This is partly because of the stateless nature of servlet applications, and partly because of the carefully placed *protective* `try-catch` blocks that catch unexpected exceptions. These `try-catch` blocks provide good protection between different parts of the program.

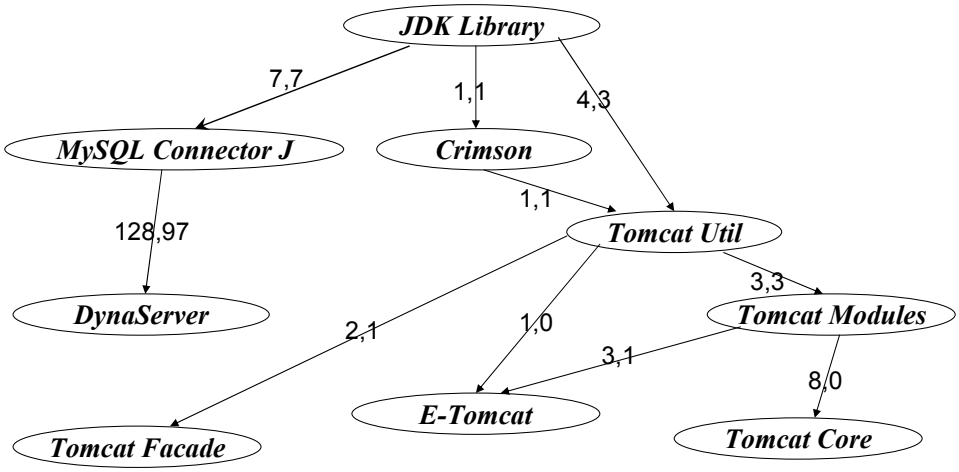


Figure 7.8: Inter-component Link Coverage

Figure 7.8 depicts the *Link Coverage* of inter-component *e-c links* in a graphical manner. It has same layout with that of Figure 7.5. Ovals represent program modules and arrows represent inter-component *e-c links* as segments of I/O related *e-c chains*. One arrow may represent several *e-c links* with the same direction. Some arrows in Figure 7.5 are missing here, because only I/O related *e-c chains* are shown in this figure, whereas all possible exception propagation paths are shown in Figure 7.5. Each arrow

is labeled with a pair of integers x, y where x is the number of *e-c links* represented by the arrow and y is the number of these *e-c links* that were covered during the test. This figure allows easy identification of poorly tested *e-c links* between program modules. We can see from the picture that many *e-c links* from MySQL Connector J to DynaServer servlet application were not covered. Also many *e-c links* starting from Tomcat modules were not covered. Next we will present the investigation results of these *e-c links*.

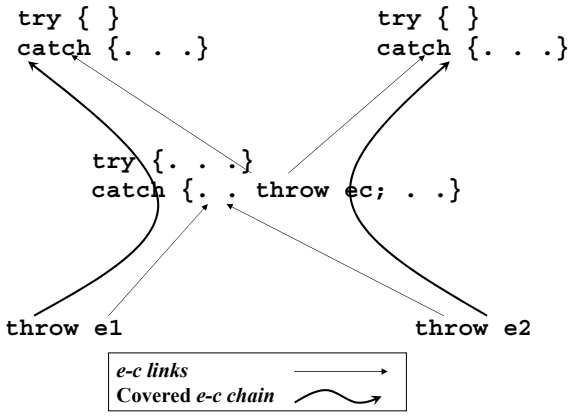


Figure 7.9: X Shape *e-c chains*

We had expected to see a combinational coverage problem near `catch` clauses that appear in many different *e-c chains*. For example, as shown in Figure 7.9, four *e-c links* form a big **X** shape which contains 4 *e-c chains*. If only 2 of them were covered as shown in the figure, we would cover all the *e-c links* but only half of the chains in this part of the system. But to our surprise, we found no such case in the experiment results. In another words, for each *e-c chain* not covered, there was at least one *e-c link* as an edge of the *e-c chain* that was not covered during the entire testing process. For this reason, we investigated in detail the reason why some *e-c links* as segments of *e-c chains* were not covered during the test, but did not try to inspect entire uncovered *e-c chains*. There are total of 58 such *e-c links* not covered in our test, which, after our inspection, can be divided into the following categories (detailed explaination of each category will follow shortly):

1. *e-c links* whose `try` blocks are not reached by normal execution.

2. *e-c links* corresponding to infeasible call chains.

3. *e-c links* whose feasibility depended on call chains beyond the span of the *e-c links* themselves.

4. *e-c links* whose feasibility can not be decided by manual inspection.

Table 7.3: Categories of Uncovered edges in *e-c chains*

Category	*#Edges*	*Start Module*	*End Module*
1	8	MySQL Connector/J	MySQL Connector/J
	1	Tomcat Util	E-Tomcat
	1	Tomcat Util	Tomcat Facade
2	1	JDK Lib	Tomcat Util
3	31	MySQL Connector/J	DynaServer
4	6	Tomcat Modules	Tomcat Modules
	8	Tomcat Modules	Tomcat Core
	2	Tomcat Modules	E-Tomcat

Table 7.3 lists the number of uncovered *e-c links* in each category. The third and fourth columns give these *e-c links*' direction at the level of program modules, and the second column lists number of *e-c links* in each direction.

There are totally 10 *e-c links* in the first category whose corresponding `try` block can not be reached by normal execution. These *e-c links* end in 5 different `try-catch` clauses, where we can not find ways to push the program to even execute the `try` block. Two of these `try` blocks execute "safety check" code (e.g., try to ensure some resource is initialized before actually using it). Because this backup initialization code is guarded by some flag, it is very hard to actually execute the `try` blocks in it. The other 3 `try` blocks are guarded by complex control flow that we can not fully understand.

In the second category, one *e-c link* corresponds to infeasible call chains, and thus can never be covered. We believe it is very hard for any static program analysis to prune this infeasible *e-c link*, because its infeasibility is decided by the value of some string object. The *e-c link* starts from a network operation and ends in a `try-catch` block that reads a configuration file whose position is specified by an URL. Since the first part of the URL is hard coded as `file://`, it can never be accessed over the network. So the

corresponding operation will never be affected by network faults. This *e-c links* starts from JDK library and ends in Tomcat Util module. This is also the only uncovered *e-c link* with high "sharing degree" (14).

In total there are 31 *e-c links* in the third category that correspond to feasible call chains, but these *e-c links* are infeasible themselves. This is a very interesting new finding, because it never happened in the coverage testing of individual *e-c links* described in Chapter 4. When we say that one *e-c link* (p, c) corresponds to feasible call chains, we mean there exists at least one feasible call chain starting from the `try` block corresponding to the c and ending at the method containing the `throw` statement p. Usually in test of *e-c links* in Chapter 4, locating this feasible call chain and driving the program through it would lead to the *e-c link* being covered. But in this case, p is located inside another `catch` clause, as a rethrow, so things get more complex than what we experienced before.

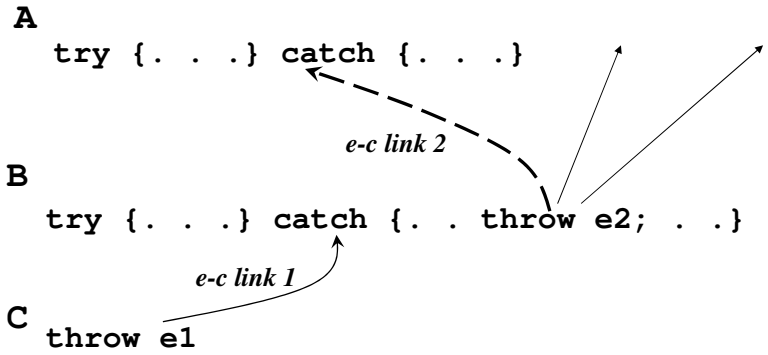


Figure 7.10: To Cover a *e-c chain*

Figure 7.10 shows a simplified example illustrating this case. Arrows in the picture representing *e-c links* . Our goal is to cover *e-c link* 2 (shown as a dashed arrow in the picture) in our test. We have already confirmed by using profiling that there are feasible call chains from method `A` to method `B`, and we successfully covered *e-c link* 1 (shown as a solid arrow in the picture) during our test. This tells us that there is feasible call chain from method `B` to method `C`, but we were not able to cover *e-c link* 2. After a detailed inspection we found that in this case, driving the normal execution to reach

method `B` via method `A` is not enough to cover *e-c link* 2, when the `throw` statement in `B` is hard to reach.

We found that to cover *e-c link* 2, we have to first drive the program to reach method `B` via method `A`, and at the same time trigger the `catch` clause in `B`. To do that, we have to cover *e-c link* 1 (i.e., continue driving the program to reach method `C`). Now the task becomes driving the execution from method `A`, via method `B` and reaching method `C`. Then an exception will be raised in `C`, which will trigger `catch` clauses in both method `B` and `A`, covering both *e-c link* 1 and 2 in the picture. Bottom line: we need a feasible call chain starting from method `A`, that goes through method `B` and reaches method `C`.

Unfortunately, we found, by manual inspection, there is no such feasible call chain to support the corresponding 2-link chain. That is, although `A` to `B` is feasible and `B` to `C` is also feasible, `A` via `B` to `C` is not. Method `B` in the example represents `MysqlIO.sqlQueryDirect()` in MySQL JDBC connector. It would call different methods depending on the kind of request received. Only a call to one of these methods may lead to *e-c link* 1 in the picture. Requests from method `A` will not trigger the call necessary for covering *e-c link* 1. This is controlled by a sequence of `if` statement in the method (i.e., complicated control flow), so our flow insensitive analysis will not be able to identify the infeasibility of this path.

All of these 31 *e-c links* go from MySQL Connector/J to the DynaServer servlet application. By looking at Figure 7.8, we can see that we have covered all feasible I/O related exception propagation paths between these two modules.

In the fourth category, there are 16 uncovered *e-c links* with undeterminable feasibility by our manual inspection of the source code and analysis results. These *e-c links* start from rethrow operations in 3 different `catch` clauses, and end in 10 different `catch` clauses. We were able to drive the program into these `try` blocks but still could not cover them in the test. These 10 `catch` clauses are in methods that are very close to the root of the program call graph, one of them being in `main` method itself. Call chains corresponding to these *e-c links* are complicated and hard to explore; thus we are not sure whether these *e-c links* are actually feasible or not. One observation we can make is that in these *e-c links*, the shortest call chain from any p to c is 6. We

feel this distance between where the exception is thrown and where it is handled is too large to allow effective inspection.

From the result of these experiments, we believe that with *e-c chain* analyses and the extended testing framework, we can help programmers categorize `catch` clauses according to their behavior, understand exception-handling structure of a given system, and systematically test that structure to find problems in the exception handling code. The case study in *e-c chain* testing shows that our testing framework has been successfully extended to test exception handling structure of Java systems. With our extended testing framework, we converted the task of testing exception handling structure of a given program into driving the execution into different program paths, a traditional testing problem. Carefully crafted functional test cases can be reused in exception handling code testing to improve reliability of the system under test.

Chapter 8

Related Work

There is much previous research related to the systems discussed in this thesis. Those most related to our work can be divided into several categories: fault-injection testing, dataflow testing coverage metrics, exception-handler analysis and compilation, points-to analysis (for reference variables) and infeasible path analysis, which will be discussed in the following sections.

8.1 Fault Injection

There has been considerable previous work in the operating systems community on using run-time fault injection for testing the robustness of programs. In the dependability community, (program) *coverage* is defined as the conditional probability that the system properly processes a fault, given that a fault occurs [12, 21]. A stochastic model of expected fault occurrence is used to guide the selection of faults that are then injected into a running program and the resulting execution is observed [5]. This approach yields a stochastic-based fault coverage that treats the running program as a *black box* [45]; the behavior of the program after the fault is injected is the criteria by which coverage is achieved or not. In contrast, the fault-injection testing experiments described in this thesis measure coverage in a manner similar to the software engineering testing community, which uses the percentage of program entities (e.g., branches, methods, def-use relations) exercised as a quantitative measure of coverage [52, 45].

There has been some research in the dependability community that uses similar program-based coverage measures to those in this thesis. Tsai et. al [76] placed breakpoints at key program points along known execution paths and injected faults at each point, (e.g., by corrupting a value in a register). Their work differs from ours in its goal,

the kinds of faults injected, and their definition of coverage. The primary goal of their approach was to increase fault activations and fault coverage, not to increase program coverage. They injected a set of hardware-centric faults such as corrupting registers and memory; these faults primarily affected program state, not communication with the operating system or I/O hardware. They used a basic-block definition of program coverage, rather than measuring coverage of a program-level construct such as a `catch` block. Bieman et. al [9] explored an alternative approach where a fault is injected by violating a set of pre- or post-conditions in the code, which are required to be expressed explicitly in the program by the programmer. This approach used branch coverage, a program-coverage metric.

In the terminology of Hamlet's summary paper reconciling traditional program-coverage metrics and probabilistic fault analysis [28], our work can be classified as a probabilistic input sequence generator, exploring the low-frequency inputs to a program. Using the terminology presented by Tang and Hecht [73], which surveyed the entire software dependability process, our method can be classified as a stress-test, because it generates unlikely inputs to the program.

8.2 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 [52, 22]. 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 especially important to ensure the dependability of the web applications that are our focus (see Chapter 4).

The overall exception def-catch coverage metric for *e-c links*, that relates resource-usage faults to specific exception objects, is analogous to the *all-uses* metric in traditional def-use testing [52], with fault-sensitive operations corresponding to definitions of exceptions and `catch` blocks corresponding to uses.

8.3 Points-to analysis and Infeasible Path Pruning

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 [32, 58]. Our partially context-sensitive points-to analysis is most closely related to the context-sensitive analyses in [43, 42]. These approaches avoid the cost of non-selective 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 [43, 42], 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 [15, 27, 49, 37, 79]; these analysis algorithms implement non-selective context sensitivity.

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 [11]. Ngo et al. proposed a novel approach to identify intraprocedural infeasible paths by recognizing preselected code patterns using static program analysis [48]. 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, which is not addressed in thesis works.

Souter and Pollock present a methodology (without empirical investigation) for demand-driven analysis for the detection of type infeasible call chains [69, 70]. Their work is related to our data reachability analysis for the computation of infeasible *e-c links*. 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 can 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 [69] and [70]. 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.

Rountev et. al [56] investigates the potential of various call graph construction algorithms to weed out infeasible call chains. They find that Andersen's points-to analysis (the same points-to analysis that we are using) achieves close to the 'best solution' possible for any analysis which considers all control branches to be feasible. This finding re-enforces our observation of uncovered infeasible *e-c links* in our experiments, that involved complex control conditions which 'fooled' the analysis.

8.4 Exception Handling Analyses and Tools

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.

8.4.1 Static Exception-Flow Analysis

Jo et. al [33] present an interprocedural set-based [31] 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. A tool [14] was built based on this analysis which shows, for a selected method, uncaught exceptions and their propagation paths. It is unclear from the paper whether a certain path for each exception is selected and displayed, or if all of the paths are displayed.

Robillard et. al [55] describe a dataflow analysis that propagates both checked and unchecked exception types interprocedurally. Their tool *Jex* presents a user with a graphical interface displaying the exception type structure of her program, for better

understanding; specific emphasis is on the correcting of exception handling by subsumption.

Sinha et. al defined a set of coverage metrics for testing exception constructs and gave their subsumption relations [65]. The metrics were defined for checked exceptions explicitly thrown in user code. 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. We focus on implicit checked exceptions that are thrown in JDK libraries, whereas they deal with user-thrown exceptions. Analysis presented in [66] 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. 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 their analysis; 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.

These algorithms 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 [20, 7]. 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). Third, 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). Moreover although all of these static analyses identify individual exception-flow links, none of

them discover the possible semantic relations between these links, induced by shared exception objects or exception data. This prevents the tools built upon them from discovering overall exception flow and handling architecture in module-based systems.

8.4.2 Dynamic Exception-Flow Analysis

A dynamic analysis of exception-flow is presented by Candea et. al [13]. 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.

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). Thus, a programmer trying to locate an exception cause may have insufficient information to succeed.

8.4.3 Tools

Reimer and Srinivasan [53] introduced *SABER*, part of which targets at a wide range of exception usage issues in order to improve exception handling code in large J2EE applications. These issues include swallowed exceptions, single `catch` for multiple exceptions, a handler too far away from the source of the exception and costly handlers. Warnings are given to the programmer upon recognizing one of these problems. Unfortunately, the underlying analysis is not discussed.

Sinha and colleagues [67] showed schematic views of a visualization tool (i) to visualize exception anomalies similar to those defined in [53] by using the static analysis of [66], and (ii) to display exception-based test coverage requirements, i.e. list of code entities need to be exercised to achieve the designated coverage. The static analysis used for call graph building for both of these tasks is based on CHA. Our experiments on testing interprocedural exception handling in moderately large benchmarks (e.g., 2080 methods, 278 classes) showed that more than 97% of the e-c links found using CHA were false positives. Thus, the analysis in [67] has been shown to be too imprecise for practical use on real programs. In addition, it is not clear how exceptions thrown within the Java JDK libraries are accounted for in [67]; the case example illustrated by schematic views shows the usage of exceptions in the code is sparse and does not seem to include exceptions thrown by the Java libraries and caught by the application. These factors raise serious questions about the practicality and scalability of the analysis in [67] and thus, the utility of the proposed tool.

8.5 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 [71]). The IBM Tokyo JIT compiler [50], 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* [35], 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* [17] performs lazy exception throwing, in that it avoids creating exception objects, where possible, unless they are live on entry to their handler.

Choi et. al [16] designed a new intraprocedural control-flow representation, that efficiently accounted implicit control flow caused by operations that might generate exceptions called *PEIs, potentially excepting instructions*; they used this representation

as a basis for safe dataflow analyses for an optimizing compiler.

Chapter 9

Summary

Exception handling code plays a very important role when a subsystem error occurs. The quality of the exception handling code directly affects system wide availability. We developed a series of analysis algorithms and tools to help both understanding, inspection of the exception handling code, as well as testing, for the purpose of improving the quality of the exception handling code so as to increase the system availability. Special effort has been put on improving the precision of the analysis to help programmers reduce the amount of time and effort spent in the spurious cases reported by the analyses. Carefully crafted static analyses are used to locate exception propagation paths in a given Java program. These paths, when provided in some structural way, can help a programmer navigate exception handling code that relates to certain kind of problem, or to locate all the problems that should be handled in a given handler. These paths can also help in testing exception handling code, when paired with our compiler-directed fault injection testing engine.

9.1 Exception Analysis

We presented our *Exception Flow* analysis and formalized it as an interprocedural dataflow analysis. The design of the analysis allows us to experiment with different supporting technology. As the result of the experimentation, we feel that exception-flow information derived solely from type-base analysis techniques such as Class Hierarchy Analysis, contains too many infeasible links from exception throws to `catch` clauses; however, using points-to analysis as a base significantly improved the precision of the *Exception Flow* analysis.

To further improve the precision of the *Exception Flow* analysis, we developed a

post-pass feasibility pruning technique: *DataReach*, which can be instantiated into a schema of successively more precise data reachability algorithms. Moreover, the usage of *DataReach* algorithms is *not* limited to exception analyses. They actually function as general call chain feasibility analysis algorithms. Experiments showed that *DataReach* algorithms are effective in reducing the number of false positives produced by *Exception Flow* analysis. We also found that *DataReach* algorithms are more effective when paired with a context-sensitive points-to analysis.

9.2 Fault Injection Testing

We proposed what we believe to be a new challenge in the field of highly available systems: to determine whether all of the fault-recovery code in a server application has been exercised. We have presented our *Exception Def-catch Coverage* metric, which formalizes what it means to meet this challenge successfully.

A *compiler-directed fault-injection testing* framework was designed and implemented. Paired with exception analysis summarized previously, we conducted experimentation on a set of reasonable sized Java Web server programs. We have shown algorithms and experiments results on automatically instrumenting programs to inject faults on vulnerable program points and collect coverage information at runtime. Our coverage metric and testing frame work combines ideas of testing software in response to injected faults, developed by the dependability community, with ideas of testing for coverage of specific program constructs, developed by the software engineering community.

9.3 Exception Flow Visualization

We presented a tool that facilitates navigating code related to the exception handling feature of Java programs, again, based on the exception def-use analysis summarized above. We want to reveal all information needed to the user, while carefully organizing the data to help browsing and reasoning. With this tool, a programmer can quickly and precisely locate all the exception handling code that may deal with a certain kind of problem. It can also locate all the problems that can reach a given `catch` clause.

This feature facilitate structural navigation of exception handling code which helps in exception handling code understanding and inspection.

Despite of our current efforts, exploring program code based on conservative static program analysis results can be difficult. One way to alleviate the situation is to use a more precise (but possibly more expensive) analysis to reduce member of spurious results.

9.4 Exception Chain Analysis

We have defined a static *Handler-inspection* analysis that examines reachable `catch` clauses to identify `catch` clauses that rethrow exceptions and to categorize caught exception usage. This categorization, when paired with *e-c links* information provided by the exception def-use analysis summarized above, can help a programmer decide the importance of a given `catch` clause and whether its behavior is appropriate.

Also, 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 exception-handling testing framework was extended to handle *e-c chains*, which can help a programmer ensure the quality of the exception handling structure of the whole system under test. One case study on Tomcat was conducted to demonstrate the effectiveness of the extension.

9.5 Limitation and Future Work

One focus of our work was to find *e-c links* in a given program with high precision. We put a lot of effort on finding the right technique to improve the precision of the analysis. As already mentioned above, experiments showed that points-to analysis is required for reasonable precision.

Although points-to analysis provides a very good base for this kind of analysis, it does have some limitations; one of them is not handling reflection well and another one is the *closed world* assumption, which means that static analysis must have access to *all* the program code at compile time. In big object-oriented systems in either Java or C#, we have yet to find a program that does not use reflection. It is true that at these dynamic class loading or reflection sites, some conservative assumptions can be used to continue the analysis. But the imprecision introduced by these conservative assumptions can be propagated and in many cases exaggerated, causing global performance degradation.

In today's software development, as programs become more and more dynamic, it becomes harder and harder for static program analysis to provide precise information. We feel that in the places where static program analysis falls short, we need to rely on dynamic analysis to provide precise information, which is very valuable for program understanding as well as for problem diagnostics.

But dynamic program analysis does *not* provide the *safety* guarantee that comes with static program analysis, which is often necessary in program white-box testing, where *all possible cases* are required to be the denominator in the coverage metrics.

Facing this dilemma, we feel that much research work is needed to design new testing frameworks and coverage metrics to accommodate more and more dynamic behavior in today's programs, and also to make these coverage metrics more practical.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers – Principles, Techniques and Tools*. Addison Wesley, 1988.

[2] Apache Software Foundation. Apache jarkarta project: Jakarta commons. Available at http://jakarta.apache.org/commons/.

[3] Apache Software Foundation. Apache tomcat. Available at http://tomcat.apache.org/.

[4] Ariane 501 Inquiry Board. Ariane 5 Filght 501 Failure, July 1996. Available at http://esamultimedia.esa.int/docs/esa-x-1819eng.pdf.

[5] Jean Arlat, Alain Costes, Yves Crouzet, Jean-Claude Laprie, and David Powell. Fault injection and dependability evaluation of fault-tolerant systems. *IEEE Transactions on Computers*, 42(8):913–923, August 1993.

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

[7] 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)*, October 1996.

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

[9] J. Bieman, D. Dreilinger, and L. Lin. Using fault injection to increase software test coverage. In *Proc. 7th Int. Symp. on Software Reliability Engineering (ISSRE'96)*, pages 166–74. IEEE Computer Society Press, 1996.

[10] Robert V. Binder. *Testing Object-oriented Systems*. Addison Wesley, 1999.

[11] Rastislav Bodik, Rajiv Gupta, and Mary Lou 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.

[12] W. G. Bouricius, W. C. Carter, and P. Schneider. Reliability modeling techniques for self repairing computer systems. In *Proceedings of the 24th National Conference of the ACM*, pages 295–309, March 1969.

[13] George Candea, Mauricio Delgado, Michael Chen, and Armando 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.

[14] Byeong-Mo Chang, Jang-Wu Jo, and Soon Hee Her. Visualization of exception propagation for java using static analysis. In *SCAM'02: Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation*, October 2002.

[15] Ramkrishna Chatterjee, Barbara G. Ryder, and William. A Landi. Relevant context inference. In *Proceedings of the ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, January 1999.

[16] Jong-Doek Choi, David Grove, Michael Hind, and Vivek 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.

[17] Michal Cierniak, Guei-Yuan Lueh, and James 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.

[18] Michel Cukier, Ramesh Chandra, David Henke, Jessica Pistole, and William 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.

[19] Scott Dawson, Farnam Jahanian, and Todd 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.

[20] 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.

[21] Joanne Bechta Dugan and Kishor S. Trivedi. Coverage modeling for dependability analysis of fault-tolerant systems. *IEEE Transactions on Computers*, 38(6):775–787, June 1989.

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

[23] Chen Fu, Richard P. Martin, Kiran Nagaraja, Thu D. Nguyen, Barbara G. Ryder, and David Wonnacott. Compiler-directed program-fault coverage for highly available Java internet services. In *In Proceedings of the International Conference on Dependable Systems and Networks (DSN 2003)*, June 2003.

[24] Chen Fu, Ana Milanova, Barbara G. Ryder, and David G. Wonnacott. Robustness Testing of Java Server Applications. *IEEE Transactions on Software Engineering*, 31(4):292–311, April 2005.

[25] Chen Fu and Barbara G. Ryder. Navigating error recovery code in java applications. In *eclipse '05: Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, pages 40–44, New York, NY, USA, 2005. ACM Press.

[26] Chen Fu and Barbara G. Ryder. Exception-chain analysis: Revealing exception handling architecture in java server applications. In *ICSE '07: Proceedings of the*

29th International Conference on Software Engineering, pages 230–239, Washington, DC, USA, 2007. IEEE Computer Society.

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

[28] Dick Hamlet. Foundations of software testing: dependability theory. In *Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of software engineering*, pages 128–139. ACM Press, 1994.

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

[30] 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, April 1995.

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

[32] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, Snowbird, UT, 2001.

[33] Jang-Wu Jo, Byeong-Mo Chang, Kwangkeun Yi, and Kwang-Moo Choe. An uncaught exception analysis for java. *Journal of Systems and Software*, 72(1):59–69, 2004.

[34] 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.

[35] Seungll Lee, Byung-Sun Yang, Suhyun Kim, Seongbae Park, Soo-Mook Moon, Kemal Ebcioglu, and Erik Altman. Efficient Java exception handling in just-in-time compilation. In Proceedings of the ACM SIGPLAN Java Grande Conference, 2000.

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

[37] Ondřej Lhoták and Laurie Hendren. Context-sensitive points-to analysis: is it worth it? In *Compiler Construction, 15th International Conference*, volume 3923 of *LNCS*, pages 47–64, Vienna, March 2006. Springer.

[38] Xiaoyan Li, Richard P. Martin, Kiran Nagaraja, Thu D. Nguyen, and Bin 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, January 2002.

[39] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification. Second Edition.* Addison Wesley, 1999.

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

[41] Microsoft Corporation, 2003. Available at http://msdn2.microsoft.com/en-us/library/618ayhy6(VS.71).aspx.

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

[43] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM Transactions on Software Engineering Methodology*, 14(1):1–41, January 2005.

[44] The Muffin world wide web filtering system. Available at http://muffin.doit.org/.

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

[46] MySQL. Mysql connector/j, 2003. Available at http://www.mysql.com/products/connector/j/.

[47] Kiran Nagaraja, Xiaoyan Li, Bin Zhang, Ricardo Bianchini, Richard P. Martin, and Thu D. Nguyen. Using fault injection to evaluate the performability of cluster-based services. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS 2003)*, Seattle, WA, March 2003.

[48] Minh Ngoc Ngo and Hee Beng Kuan Tan. Detecting large number of infeasible paths through recognizing their patterns. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 215–224, 2007.

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

[50] Takeshi Ogasawara, Hideaki Komatsu, and Toshio 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.

[51] Michael J. Radwin. The java network file system.

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

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

[54] Rice University. Dynaserver: System support for dynamic content web servers, 2003. Available at http://www.cs.rice.edu/CS/Systems/DynaServer/.

[55] Martin P. Robillard and Gail 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.

[56] Atanas Rountev, Scott Kagan, and Michael Gibas. Static and dynamic analysis of call chains in java. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 1–11, July 2004.

[57] Atanas Rountev, Ana Milanova, and Barbara 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.

[58] Barbara G Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In Proceedings of the Twelveth International Conference on Compiler Construction, LNCS, volume 2622/2003, pages 126–137, April 2003. invited paper.

[59] Sable, McGill. Soot: a java optimization framework, 2003. Available at http://www.sable.mcgill.ca/soot.

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

[61] 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.

[62] 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.

[63] Srikanth Shenoy. Best practices in EJB exception handling. IBM developerWorks Artical, May 2002. Available at http://www-128.ibm.com/developerworks/library/j-ejbexcept.html.

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

[65] Saurabh Sinha and Mary Jean Harrold. Criteria for testing exception-handling constructs in Java programs. In Proceedings of the International Conference on Software Maintenance, 1999.

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

[67] Saurabh Sinha, Alessandro Orso, and Mary Jean Harrold. Automated support for development, maintenance, and testing in the presence of implicit control flow. In *Proc. Int'l Conf. Software Engineering (ICSE'04)*, 2004.

[68] Peter Sortokin. Ftp server in java, 2003.

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

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

[71] Specbench.org. Spec jvm98 benchmarks. Available at http://www.spec.org/jvm98/.

[72] Sun Microsystems. Java 2 platform, standard edition, v 1.4.2 api specification. Available at http://java.sun.com/j2se/1.4.2/docs/api/.

[73] Dong Tang and Herbert Hecht. An approach to measuring and assessing dependability for critical software systems. In *In Proceedings of the Eighth International Symposium on Software Reliability Engineering*, pages 192–202, Albuquerque, NM, November 1997.

[74] Dong Tang and Ravishankar K. Iyer. Analysis and Modeling of Correlated Failures in Multicomputer Systems. In *ACM Transactions on Computer Systems*, pages 567–577, May 1992.

[75] Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. In *Proceedings of the Conference on Object-oriented Programming, Languages, Systems and Applications*, pages 281–293, October 2000.

[76] T. Tsai, M. Hsueh, H. Zhao, Z. Kalbarczyk, and R. Iyer. Stress-based and path-based fault injection. *IEEE Transactions on Computers*, 48(11):1183–1201, November 1999.

[77] Volano LLC. Volanomark. Available at http://www.volano.com/benchmarks.html.

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

[79] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 131–144, New York, NY, USA, 2004. ACM Press.

Vita

Chen Fu

2007 Ph. D. in Computer Science, Rutgers University,

2001 M.S. in Computer Science, Chinese Academy of Science, Beijing, China

1997 B.S. in Computer Science, Beijing University, Bejing, China

2001-2007 Teaching/Graduate Assistant, Department of Computer Science, Rutgers University

1998-2001 Research Assistant, Institute of Computing Technology, Chinese Academy of Science, Beijing, China

2007 Chen Fu and Barbara G. Ryder. Exception-chain Analysis: Revealing Exception Handling Architecture in Java Server Applications. In *Proceeding of the 29th Int. Conference on Software Engineering*, Minneapolis, MN, May 2007

2005 Chen Fu and Barbara G. Ryder. Navigating Error Recovery Code in Java Applications. In *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, San Diego, CA, Oct. 2005

2005 Chen Fu, Ana Milanova, Barbara G. Ryder, David G. Wonnacott. Robustness Testing of Java Server Applications. In *IEEE Transactions on Software Engineering*, 31(4), Apr. 2005.

2004 Chen Fu, Barbara G. Ryder, Ana Milanova, David G. Wonnacott. Testing of Java Web Services for Robustness. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2004)*, Boston, MA, Jul. 2004

2004 Dong-yuan Chen, Lixia Liu, Roy Dz-Ching Ju, Chen Fu, Shuxin Yang, Chengyong Wu. Efficient Modeling of Itanium Architecture during Instruction Scheduling using Extended Finite State Automata. In *Journal of Instruction-Level Parallelism*, vol. 6, 2004.

2003 Chen Fu, Richard P. Martin, Kiran Nagaraja, Thu D. Nguyen, Barbara G. Ryder, David G. Wonnacott. Compiler Directed Program-fault Coverage for Highly Available Java Internet Services. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN, IPDS track)*, San Francisco, CA, June 2003.

2003 Dong-Yuan Chen, Lixia Liu, Chen Fu, Shuxin Yang, Chengyong Wu, Roy Ju. Efficient Resource Management during Instruction Scheduling for the EPIC Architecture. In *Proceedings of 12th International Conference on Parallel Architectures and Compilation Techniques (PACT'03)*, New Orleans, Louisiana, Sep. 2003