Robustness Testing of Java Server Applications

Chen Fu, Ana Milanova, *Member*, *IEEE Computer Society*,
Barbara Gershon Ryder, *Member*, *IEEE Computer Society*, and
David G. Wonnacott, *Member*, *IEEE Computer Society*

Abstract—This paper presents a new compile-time analysis that enables a testing methodology for white-box coverage testing of error recovery code (i.e., exception handlers) of server applications written in Java, using compiler-directed fault injection. The analysis allows compiler-generated instrumentation to guide the fault injection and to record the recovery code exercised. (An injected fault is experienced as a Java exception.) The analysis 1) identifies the *exception-flow "def-uses"* to be tested in this manner, 2) determines the kind of fault to be requested at a program point, and 3) finds appropriate locations for code instrumentation. The analysis incorporates refinements that establish sufficient context sensitivity to ensure relatively precise def-use links and to eliminate some spurious def-uses due to demonstrably infeasible control flow. A runtime test harness calculates test coverage of these links using an *exception def-catch* metric. Experiments with the methodology demonstrate the utility of the increased precision in obtaining good test coverage on a set of moderately sized server benchmarks.

Index Terms—Reliability, def-use testing, Java, exceptions, test coverage metrics.

1 Introduction

The emergence of the Internet as a ubiquitous computing infrastructure means that a wide range of applications—such as online 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. A good analogy is to the telephone system, a technology that one expects to be "always working;" the national telephone system demands only minutes of down time per year from its software. New testing technologies are needed to address the issue of reliability in this environment. Besides the traditional testing of functionality, there is a need to ensure reasonable application response to system/resources problems, in order to have performance gracefully degrade rather than experience application crashes. The robustness testing research in this paper addresses the problem of how to test the reliability of server applications written in Java, in the face of infrequent but anticipatable system problems that the program may respond to via Java's exception handling mechanism.

Traditional fault-injection testing of software in the operating system community is conducted in a black-box manner, using a probabilistic analysis to determine whether or not a software component will work properly when subjected to specific fault loads and workloads [1]. Testing is accomplished by simulating faults caused by environmental errors during test through *fault injection* [2], [3], [4], [5], [6]. 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 [1].

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

Our methodology uses the tools of white-box def-use testing to aid a tester of a server application in this task. There is a large body of existing work on *white-box* testing methodologies [7], [8], [9], aimed at exercising as much application code as possible during testing, and measuring code coverage using various program constructs such as control-flow edges, branches, and basic blocks. However, error recovery code—code which handles errors that occur with small probability, especially due to interactions with the 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 executable by merely manipulating program inputs.

- *C. Fu and B.G. Ryder are with the Rutgers University Department of Computer Science, Piscataway, NJ 08854. E-mail: {chenfu, ryder}@cs.rutgers.edu.*
- *A. Milanova is with the Rensselaer Polytechnic Institute Department of Computer Science, Troy, NY 12180. E-mail: milanova@cs.rpi.edu.*
- *D.G. Wonnacott is with the Haverford College Department of Computer Science, Haverford, PA 19041. E-mail: davew@cs.haverford.edu.*

Manuscript received 26 Oct. 2004; revised 4 Feb. 2005; accepted 29 Mar. 2005; published online 26 May 2005.
Recommended for acceptance by G. Rothermel.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSESI-0247-1004.

Our analysis techniques identify program points vulnerable to certain faults and the corresponding error recovery code for these specific system faults. The techniques provided allow compiler-inserted instrumentation to inject appropriate faults as needed and to gather recovery code coverage information. This enables a tester to systematically exercise the error recovery code, by causing execution of the vulnerable operations. Thus, the methodology provides a means to obtain validation of application robustness in the presence of system faults. Although our experiments are based on server applications, the technique can be applied to general Java applications.

In our approach, it is important to be able to identify as precisely as possible where an exception thrown in response to an experienced fault (i.e., a def) is handled (i.e., a use) [10]. A key concern, in general, for def-use testing is how to minimize the number of spurious def-uses reported by the analysis. Since these def-uses cannot be exercised by any test, a human being has to examine them, among the uncovered def-use links after testing, and determine (if she can) that they are spurious. This is a time-consuming, difficult job, especially for large object-oriented applications that use polymorphism heavily. Therefore, it is crucial to use a very precise analysis that, while practical in cost, can eliminate many of these spurious def-uses. This is a key goal of our new *exception-catch link analysis*.

Our initial work in this area [11], [12] focused on the identification of an appropriate definition of coverage for fault-tolerant server applications, and on the definition of the compiler/fault-injector interface necessary to measure and induce coverage of fault-handling code. We presented a proof-of-concept case study in which a proxy server application was instrumented by hand, and fault injection was performed and recorded by executing the instrumentation.

In [13], we demonstrated that automatic compile-time analysis was sufficient to analyze the proxy server that we had studied, as well as several other moderately sized server applications. This analysis consisted of an *exception-flow* analysis phrased as an interprocedural dataflow problem using limited context sensitivity, coupled with a novel *data reachability analysis* to prune infeasible edges produced by the conservative approximations used in the initial analysis.

This paper is an extension of [13] that makes the following additional contributions:

- Reformulating our data reachability 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 [14]).
- Definition and exploration of several new variants of our schema (which we call *C-DataReach*, *M-DataReach*, and *V-DataReach*), as well as restatement of our original data reachability algorithm in our new schema. This exploration compares the relative accuracies and computational complexities of these four variants of our analysis.
- Empirical studies of the use of several variants of our DataReach algorithm, and several variants of the earlier stages of our analysis, on our prior benchmarks and three additional larger applications. These studies include aggregate accuracy and timing information, as well as specific discussions of the cases in which static analysis is difficult.

Overview. The rest of this paper is organized as follows: In Section 2, we describe our coverage metric, which is a slight variant of the original metric described in [11], and give an overview of the compiler-directed fault injection methodology. In Section 3, we discuss our compile-time analyses for exception-flow def-uses and our data reachability schema (including the specific instantiations of this schema used later). In Section 4, we report our empirical results on the moderate and larger-sized Java applications, describing the impact on the exception-flow def-uses obtained by varying the compile-time analysis used. In Section 5, we describe related work. Finally, we present our conclusions.

2 Measuring Coverage of Fault-Handling Code

We take advantage of the Java exception handling mechanism to help identify error recovery code. *Exceptions* in Java are used to respond to error conditions [15]. Each `catch` block is potentially the starting point of error recovery code for a matching error/exception raised during the lifetime of the corresponding `try` block.

2.1 Faults, Exceptions, Coverage Metric

A *fault* is some component failure (e.g., disk crash or network congestion [1]). A *fault-sensitive operation*, which is either an explicit `throw` statement or a call to some unknown method, is *affected* by a fault in that an exception is produced when the operation occurs and experiences a fault as a runtime error.

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. Many Java server applications, as can be seen from our benchmarks, are I/O intensive. In this paper, we focus on faults related to Java *IOExceptions*, which are related to disk and network errors. Some fault-sensitive operations correspond to I/O operations in the user code being tested or the Java or C libraries it calls, but others are initiated by the Java virtual machine (e.g., those in class loading and security policy loading). We exclude JVM I/O from our testing by automatically instrumenting the code to identify user-instigated I/O.

We denote P to be the set of all fault-sensitive operations in the code under test that may be affected by any element in the specific set of faults of interest. We assume P is known, because the relationship between faults and fault-sensitive operations 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 a given program, each element of P could possibly produce an exception that reaches some subset of the program's `catch` blocks. By viewing fault-sensitive operations as the definition points of exceptions and `catch`

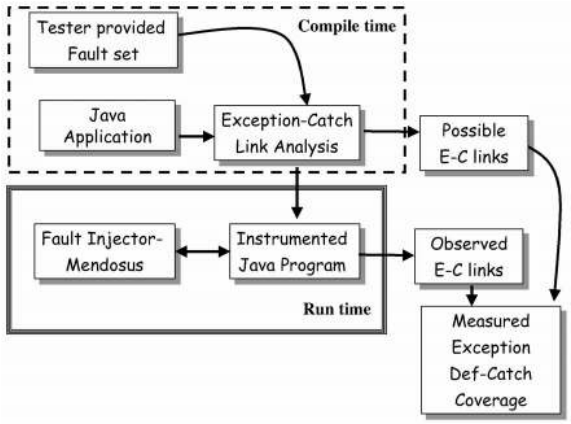


Fig. 1. Compiler-directed fault injection framework.

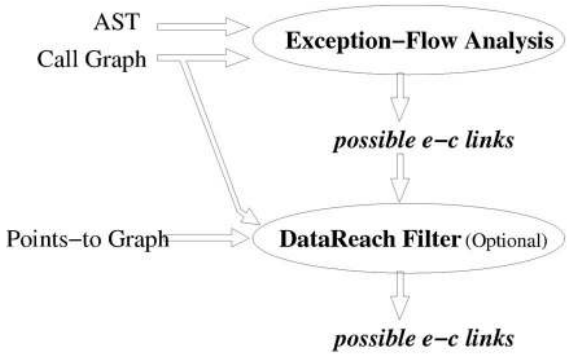


Fig. 2. Two phases of exception-catch link analysis.

blocks as uses of exceptions, we can define a coverage metric in terms of *exception-catch (e-c) links*.

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

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

Note that our exception def-catch coverage metric differs slightly from the *overall fault def-catch coverage* metric used in our earlier work [11] (where it was termed *overall fault-catch coverage*), due to the different emphasis of this work. Fault def-catch coverage measures links from specific faults to handling code, rather than from fault-sensitive operations to handling code. For example, consider code in which x distinct faults could trigger a single fault-sensitive operation and transfer control to a single `catch` block. Our fault def-catch metric would treat this as x links from faults to the catch block, and our exception def-catch metric would treat this as one possible *e-c link*. The exception-based metric is appropriate here because we wish to emphasize the ability of static analysis to prune infeasible links. This ability is not determined by the number of faults that can cause a given exception, and the use of the fault-based metric would skew our results by the size of the fault sets chosen for operations in which our analysis succeeds or fails. For a more detailed discussion of possible coverage metrics for fault-tolerant code, see [11], [12].

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

2.2 Fault Injection Framework

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

Fig. 1 shows the organization of our fault-injection system. The box labeled *compile time* shows that for a chosen set of faults, corresponding to some set of exceptions and their fault-sensitive operations, the analysis presented in Section 3 calculates the possible *e-c links* and the vulnerable statements that are susceptible to them. The compiler uses the set of *e-c links* to identify where to place instrumentation that will communicate with *Mendosus* [16], 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. The compiler also instruments the code to record the execution of the corresponding `catch` block. The tester runs the program and gathers the *observed e-c links* from that run. The tester then may have to try to make the program execute other vulnerable statements (i.e., by varying the inputs) in order to cover more of the possible *e-c links*. Finally, the test harness calculates the overall exception def-catch coverage for this test suite.

3 Compile-Time Analysis

Fig. 2 illustrates the high level structure of the two-phased compile-time exception-catch link analysis which we designed to calculate *e-c links* in Java programs.

Exception-flow analysis takes a static representation (i.e., AST) of a Java program as well as its call graph, and produces the *e-c link* set of the given program. Each of the **DataReach** analysis algorithms described in this section can serve as a postpass filter that uses the reference points-to graph [17], [18] 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. We present three distinct DataReach algorithms and report on empirical findings obtained with two of them. Intuitively, our two 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 [19], [20], [21] which differ in their cost and the precision of the resulting graph. The empirical results discussed in Section 4 show that the precision of the call graph and points-to graph has significant impact on the precision of the final *e-c link* set obtained.

3.1 Exception-Flow Analysis

In Java, if code in some method throws an 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 [15]. We want to find the relationship between `catch` blocks and fault-sensitive operations. We use "`throw` statement" to represent all fault-sensitive operations in our discussions for simplicity; we actually mean all instructions or calls that may throw some exception, if a fault occurs.

A naive analysis that relies only on examination of user declared exception types in `catch` blocks and method signatures is too inaccurate to yield information of practical use. 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 [22]—from the `throw` statement to the `catch` block where it is handled.

According to the semantics of exception handling in Java [15], 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 [23] 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, 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` or method call, and an edge to each outermost `try-catch` block.

We define for each method the set of `throw` statements that can reach its entry:

Definition (ReachingThrows(method M)). *The set of all `throw` statements for which there exists an exception handling path [22] from the throw statement to method M, and the exceptions are not handled in method M.*

Fig. 3 gives an example illustrating the definition of *ReachingThrows*. We can see that the call site `bar()` inside method `foo()` is inside the `try` block, so that `SocketException` thrown in `bar()` will be handled (i.e., killed) in `foo()` because it is 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

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

2. In Java, `try` blocks can be nested within each other. Handlers are associated with exceptions in inner to outer order [15].

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

Fig. 3. Example of ReachingThrows.

`try-catch` blocks to the call sites and `throw` statements within them.

The dataflow equations for the *ReachingThrows* problem are defined on the annotated call graph of the program.[4] 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 runtime 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 [24] and, therefore, amenable to fixed-point iteration.[5]

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

The exception-flow analysis described above relies on having an annotated call graph for the program. In order to increase precision, we added selective context sensitivity to the points-to analysis that we use to build the call graph. Rather than building a full and costly context-sensitive points-to analysis, we performed *selective constructor inlining*; that is, we inlined each constructor at its call sites, when that constructor contained a *this* reference field initialization using one of its parameters. Without this transformation, a context-insensitive analysis would make it seem that the same-named fields of all objects initialized in this constructor could point to all the parameters so used [25], [26]. We run a context-insensitive points-to analysis 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. Fig. 4 is an example of typical use of the Java network-disk I/O packages. Fig. 5 illustrates how infeasible *e-c links* are introduced even given a fairly precise call graph for the code. As we can see, the `try` block in `readFile` is only 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()`[6] and propagated to both `readFile` and `readNet`; thus, two infeasible *e-c links* are introduced reducing the achievable runtime coverage to 50 percent or less.

This inaccuracy can be resolved by using a different program representation such as a call tree [27] 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 Fig. 5, the call tree algorithm must be able to find that there are only two feasible call chains which share a middle segment of length 3. Separating these two chains would require a context-sensitive points-to analysis analogous to 4-CFA [28],

4. Under certain conditions [15], `finallys` behave like `catches` and/or `throws`. Our algorithm handles these situations correctly, but we omit the details involving `finallys` for brevity.

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

6. 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.

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

Fig. 4. Code example for Java I/O usage.

[29], 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`).

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 Fig. 4, 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 1) one of the global variables (i.e., Java static fields) or 2) 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.

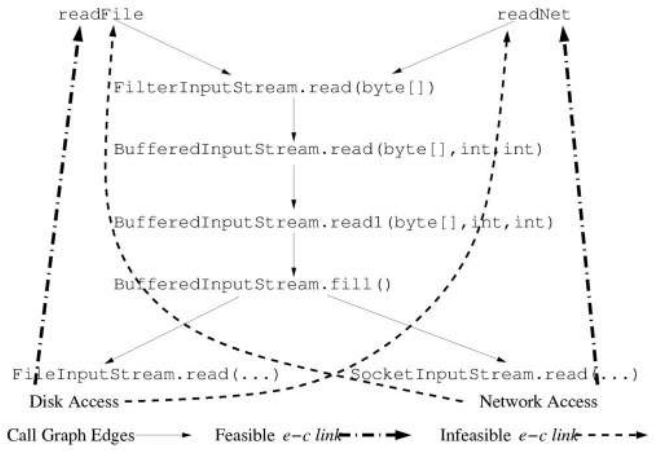


Fig. 5. Call graph for Java I/O usage.

In this specific case, `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, 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 [30], [17] to obtain the high-quality data reachability information.

The rest of this section describes DataReach, the original data reachability algorithm from [13] and discusses sources of its imprecision. Section 3.3 presents a schema of successively more precise data reachability algorithms.

3.2.1 Original DataReach Algorithm

In previous work [13], we introduced a data reachability algorithm referred to as *DataReach* that requires as input a points-to graph. 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 [25]. 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 [14]). 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. For brevity, static calls are omitted from the discussion since they can be trivially handled. 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 proven spurious.

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

Fig. 6. Imprecision of DataReach Algorithm.

3.2.2 Imprecision of DataReach

The original data reachability algorithm produced relatively precise results which led to an average of 84 percent *e-c link* coverage on an initial set of benchmarks [13]. 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 Fig. 6. 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 1) between `throw new SomeIOException` and the `catch` in `Read1`, and 2) 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 [14], 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 1) our algorithm propagates objects rather than class types, and 2) 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 Fig. 6. 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 U throughout the analysis, a set U_M is kept for each method M.

This paper discusses three instantiations of the schema: one set U valid throughout the data reachability analysis (i.e., the original DataReach discussed above), separate sets U_M for each method M (this instantiation is referred to as *M-DataReach*), and separate sets U_V for each reference variable V (referred to as *V-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 [14], $ParamTypes(M)$ 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 nontarget 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 Fig. 6. 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 $MethodLocal(o)$ 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 [17].

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

- **input**: Pt: $Ref \rightarrow \mathcal{P}(O)$.
- **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 $StaticLookup(o, m) = M'$:

$$(M \in R) \wedge (o \in U_e) \Rightarrow \begin{cases} M' \in R \wedge \\ U_{e_i} \subseteq U_{M'.f_i} \text{ where } f_i \text{ are the formal parameters of } M' \wedge \\ U_{M'.ret\_var} \subseteq U_l \wedge \\ o \in U_{M'.this}. \end{cases}$$

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:

$$\begin{cases} (M \in R) \Rightarrow o_i \in U_l \\ (M \in R) \wedge MethodLocal(o_i) \Rightarrow o_i \in Local. \end{cases}$$

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.

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

Fig. 7. Imprecision of M-DataReach algorithm on different references.

Example. Consider the set of statements in Fig. 7. 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. 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

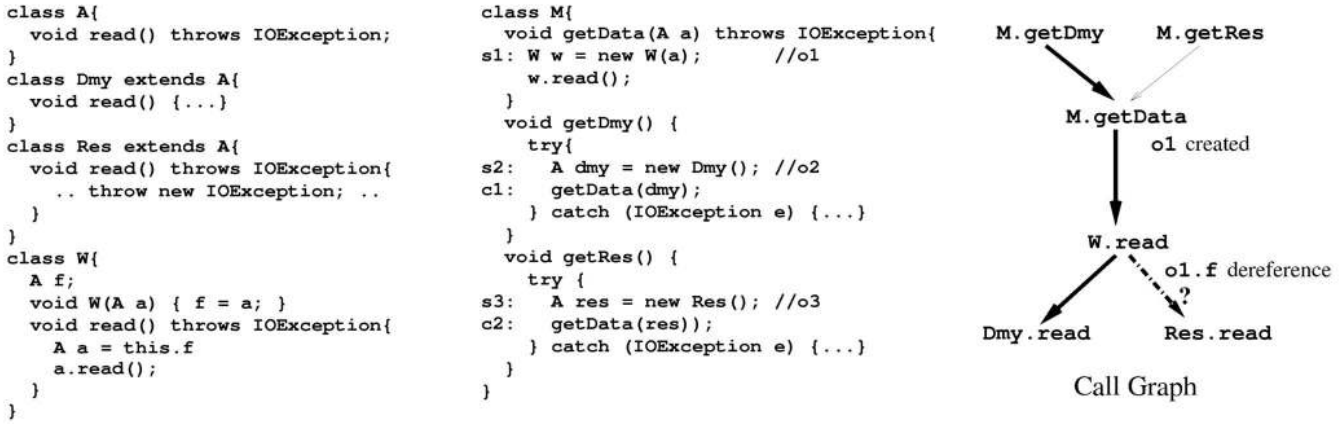


Fig. 8. Imprecision of M-DataReach algorithm on local objects.

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.

Example. Consider the example in Fig. 8. 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: 1) the complexity of generating inclusion constraints for program statements (constraints 1-3 for DataReach and M-DataReach, and 1-4 for V-DataReach), 2) the complexity of solving the system of inclusion constraints, and 3) 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 1) is $O(\mathcal{O} * E)$. The complexity of 2) 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 3) is $O(\mathcal{O}^2 * \mathcal{F} * k)$. Therefore, the complexity of our algorithms parameterized by k, the number of U sets, is: $O(\mathcal{O} * E + \mathcal{O} * k^2 + \mathcal{O}^2 * \mathcal{F} * k)$.

Table 1 summarizes our analysis in order of growing precision and complexity, because E is dominated by M^2 and V^2.

4 Empirical Results

In this section, we report our empirical findings and discuss some case histories from our experiments, whose goal was to demonstrate the effectiveness of our methodology. Initial findings on a set of four moderate-sized Java server applications have been reported previously in [13]. In this paper, we report the results of additional analysis applied to these programs and present extensive inspection results of them. New experiments with three additional, larger

TABLE 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	$\mathcal{V}$	$O(\mathcal{O} * \mathcal{V}^2 + \mathcal{O}^2 * \mathcal{F} * \mathcal{V})$

applications, including one written with the *Tomcat* framework, are presented and discussed as well.

4.1 Experimental Setup and Benchmarks

We implemented Exception-flow analysis and DataReach/M-DataReach analysis as two separate modules in the Java analysis and transformation framework Soot [18] 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) [19], and *Spark*, a field-sensitive, flow-insensitive and context-insensitive points-to analysis (a form of 0-CFA) [29], [31], [17], [30]. We implemented another call graph builder using *Rapid Type Analysis* (RTA) [20]. 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.

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

1. CHA—Build call graph with Class Hierarchy Analysis.
2. RTA—Build call graph with Rapid Type Analysis.
3. PTA—Build call graph using Spark.
4. InPTA—Build call graph with Spark plus selective constructor inlining.
5. PTA-DR—Use Spark to provide the points-to graph and call graph 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 [32].
- JNFS, The Java Network File System The server communicates with various clients via RMI [33].
- Muffin, a Web filtering proxy server [34].
- Haboob, a simple Web server based on SEDA, a staged event-driven architecture [35].
- HttpClient, an HTTP utility package from the *Apache Jakarta Project* [36]. We collected its unit tests to form a whole program to serve as a benchmark.
- SpecJVM, a standard benchmark suite [37] that measures performance of Java virtual machine, especially for running client side Java programs.
- VMark, a Java server side performance benchmark. It is based on *VolanoChat* [38]—a Web based chat server. The benchmark includes the chat server and simulated client.

Column 2 of Table 2 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 multinode application.[8]

TABLE 2
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

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.

As shown in Fig. 1, 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.

We made the usual assumptions that 1) faults are independent of each other and 2) faults occur rarely [39], [40]. We only injected one fault per run,[9] 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 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 statement (and fault-sensitive operation) were executed.

7. Selective constructor inlining, DataReach, and M-DataReach were only used where stated explicitly.

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

9. Multiple faults can be injected in one run when a vulnerable operation is inside some `catch` block.

TABLE 3
Number of *e-c links*

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**

4.2 Empirical Data

Table 3 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 shows the overall exception def-catch coverage for all the benchmarks derived from the data in Table 3. 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.

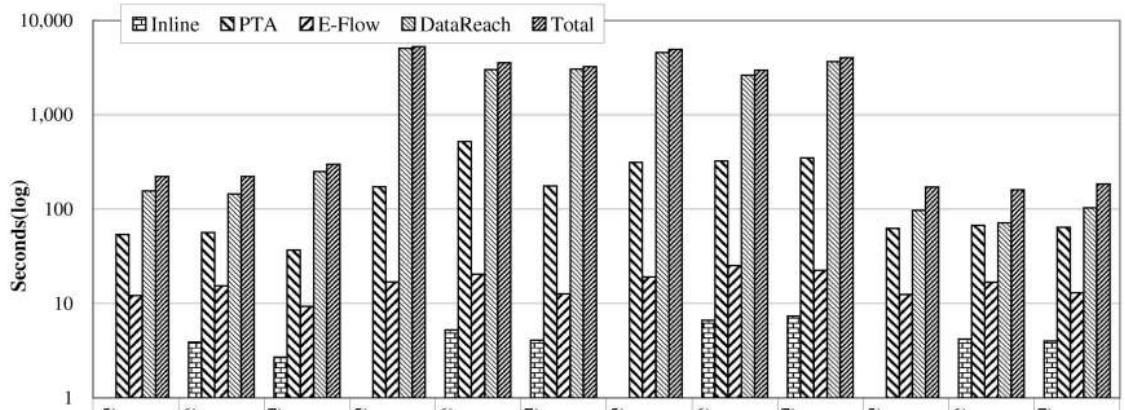
We offer two different calculations for the percentage *e-c links* covered. In columns 2-8 of Table 4, we use the metric described in Section 2 (i.e., the ratio of *e-c links* covered to possible *e-c links* found by our analysis). In the last column 9 of Table 4, we calculate the ratio of the number 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 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.

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 3). 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 3). For the *e-c links* reported by InPTA-DR, the coverage percentage of the four smaller benchmarks was stabilized at approximately 84 percent 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 nonblocking 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 3. From this data, we see that DataReach is a client of precise points-to analysis for which 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 3).

On the three larger benchmarks the coverage varied across the programs from 15 percent to 72 percent. 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.

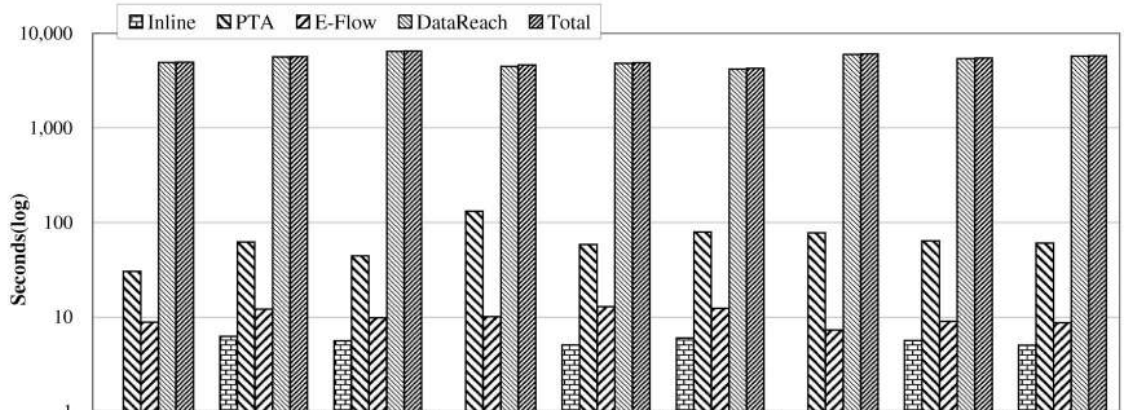
TABLE 4
Overall Exception Def-Catch Coverage

Program	CHA	RTA	PTA	InPTA	PTA-DR	InPTA-DR	InPTA-MDR	Effective Coverage
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%



	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

(a)



	HttpClient			SpecJVM			Vmark		
	5)	6)	7)	5)	6)	7)	5)	6)	7)
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

(b)

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

Fig. 9. Time cost break-down of static program analysis.

Fig. 9 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 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 3). For FTPD and Haboob, DataReach used about 50 percent of the total running time; for other benchmarks, it used more than 90 percent of the total running time. M-DataReach is slower than Data-Reach in most of the benchmarks, except SpecJVM. It takes 72 percent more time to finish in FTPD, 43 percent in Haboob, 40 percent in Muffin, and 15 percent in HttpClient. It takes 14 percent less time to finish in SpecJVM. We believe that optimized implementations of DataReach and

TABLE 5
Number of Uncovered *e-c Links* in Categories 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

M-DataReach will improve overall analysis performance significantly.

Note also that for JNFS, Muffin, Haboob, and VMark, the more precise configuration, InPTA-DR, ran more quickly than the related less precise configuration, 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.

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

4.3 Detailed Inspection

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, 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 Muffin, SpecJVM, and HttpClient, we manually inspected all the *e-c links* whose `try` blocks are reached during the testing while the *e-c links* are not experienced.[10] We categorize these *e-c links* as follows:

1. Feasible *e-c links* uncovered 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 5 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*. We will show examples extracted from each benchmark to illustrate each category in detail.

4.3.1 Muffin

There are three *e-c links* discovered in Muffin in category 3, which may be eliminated using context-sensitive points-to analysis. 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). Below is one of the possible call chains found by our analysis for one of these *e-c links*.[11] 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`.

Call chains for all three *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. Although we believe that additional context sensitivity added to the points-to analysis would further improve the precision of our *e-c links*, 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

10. We were not successful in doing this detailed study for VMark because we do not have access to its source code.

11. Parameters are omitted for readability.

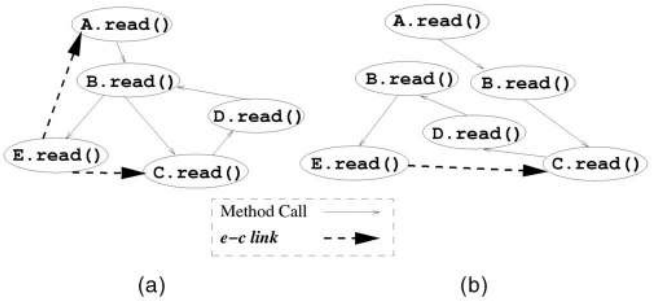


Fig. 10. Recursive call graph.

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 seven 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, we still cannot cover a large portion of the *e-c links* whose `try` blocks have been reached; 87 percent of these *e-c links* belong to category 3.

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 Fig. 10a. 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 Fig. 10b, it may become possible to compute more precise *e-c link* information.

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 percent 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 percent) and SpecJVM (13 percent) 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 six *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 Fig. 8 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* [41], 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 three 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.

5 Related Work

This paper presents exception-catch link analysis and its use in def-use testing of Java program recovery code. There is much previous research relevant to this work in: fault-injection testing, dataflow testing coverage metrics, exception-handler analysis and compilation, points-to analysis (for reference variables), and infeasible path analysis. We will discuss the most relevant research results in these areas each in turn.

5.1 Fault Injection

There has been considerable previous work in the operating systems community on using runtime 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 [42]. A stochastic model of expected fault occurrance is used to guide the selection of faults that are then injected into a running program and the resulting execution is observed [1]. This approach yields a stochastic-based fault coverage that treats the running program as a *black box* [8]; the behavior of the program after the fault is injected is the criteria by which coverage is acheived or not. In contrast, the experiments in this paper 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 [10], [8].

Recently, there has been some research in the dependability community that uses similar program-based coverage measures to those in this paper. Tsai et al. [43] 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. [44] explored an alternative approach where a fault is injected by violating a set of pre or postconditions 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 [45], 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 [46], who surveyed the entire software dependability process, our method can be classified as a stress-test because it generates unlikely inputs to the program.

5.2 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 [10], [47]. 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 [11].

Sinha et al. defined an interesting and novel set of coverage metrics for testing exception constructs and gave their subsumption relations [48]. The metrics were defined for checked exceptions explicitly thrown in user code; however, they seem easily extensible to both implicit and explicit checked exceptions. Our overall exception def-catch coverage metric seems equivalent to an extended version of their *all-e-deacts* criteria defined for both implicit and explicit exceptions. Because we are most interested in recovery code that deals with problems due to system interactions, we focus on implicit checked exceptions that are thrown in JDK libraries, whereas they deal with user-thrown exceptions, that are probably user-defined as well. No exception analysis or implementation experience with their metrics is presented.

The overall exception def-catch coverage metric for *e-c links*, that relates resource-usage faults to specific exception objects, differs slightly from our previous *overall fault-catch* coverage metric [11]. Our original metric required the injection of each kind of fault that could trigger a particular exception for a fault-sensitive instruction, rather than trying to cause a specific exception to occur. Both metrics are analogous to the *all-uses* metric in traditional def-use testing [10], with fault-sensitive operations corresponding to definitions of exceptions and `catch` blocks corresponding to uses. Overall fault-catch coverage requires the application of the complete range of faults during testing, consistent with existing operating systems fault-injection technology. In this paper, because we are injecting faults at the interface between JDK I/O methods and native methods rather than at the device level [11], we cannot differentiate between some device-level faults that result in the same exception; thus, we inject only one fault to trigger each exception.

As stated in Section 1, traditional fault-injection testing is performed by treating the application as a black box. Success is judged by how often the application does not

crash in response to an injected fault. Other white-box, controlflow coverage metrics have been proposed by some groups for use with fault-injection testing; these correspond to previous metrics (e.g., branch, edge, and basic block coverage) and have been summarized previously [11].

5.3 Analysis of Exception Handling

Two previous exception-flow analyses were aimed at improving exception handling in programs, for example, avoiding exception handling through subsumption [49], [50]. These differ from our exception-catch link analysis in significant ways. First, their call graph is constructed using class hierarchy analysis, which yields a very imprecise call graph [19], [20]. Second, these analyses trace exception types through the call graph of the program to the relevant `catch` clauses that might handle them. Conceptually, these analyses use one abstract object per class. An operation that can throw a particular exception is treated as a source of an abstract object that is then propagated along reverse control-flow paths to possible handlers (i.e., `catch` blocks).

Jo et al. [50] present an interprocedural set-based [51] 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 1,000 methods. Robillard et al. [49] describe a dataflow analysis that propagates both checked and unchecked exception types interprocedurally. Neither approach analyzes Java libraries unless source code is available (not the case for the JDK). They each handle a large subset of the Java language, but make the choice to omit or approximate some constructs (e.g., *static initializers*, *finally*s). Both of these analyses are less precise than ours, especially in their approximation of interprocedural control-flow.

Another analysis of programs containing exception handling constructs [52] calculates control dependences in the presence of implicit checked exceptions in Java. This analysis focuses on defining a new interprocedural program representation that exposes exceptional control-flow in user code. In a more recent publication [53], Sinha et al. present an interprocedural program representation which more accurately embeds the possible intraprocedural control-flow through exception constructs (i.e., `try`s, `catch`s, and `finally`s). Class hierarchy analysis is used to construct the call edges in this representation. An exception-flow analysis is defined by propagation of exception types on this representation to calculate links between explicitly thrown checked exceptions in user code and their possible handlers. It seems clear that this analysis could be extended to include implicit checked exceptions as well, assuming that the program representation could be constructed from the bytecodes of the JDK library methods, and that the fault-sensitive operations could be identified. The CHA version of our analysis seems the most similar to the analysis presented in [53]; this version is shown on our benchmarks to be too imprecise for obtaining coverage of *e-c links* corresponding to implicit checked exceptions, the focus of our work.

Choi et al. [54] designed a new intraprocedural control-flow representation, that accounted for operations that might generate unchecked exceptions called *PEIs*, *potentially excepting instructions*; they used this representation as a basis for safe dataflow analyses for an optimizing compiler. It is difficult to compare their representation with the others described here because they capture different sorts of exceptions, such as *NullPointerException*, that correspond to different possibly excepting instructions.

6 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 [37]. The IBM Tokyo JIT compiler [22] 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 percent without affecting performance of nonexception-intensive ones. In *LaTTe* [55], 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* [56] performs lazy exception throwing, in that it avoids creating exception objects, where possible, unless they are live on entry to their handler.

6.1 Points-to Analysis

There is a wide variety of reference and points-to analyses for Java which differ in terms of cost and precision. The information computed by these analyses can be used as input to our exception-flow and data reachability analyses; clearly, the precision of the underlying analysis affects the quality of the computed coverage requirements. A detailed discussion of points-to and reference analyses and the dimensions of precision in their design spectrum appears in [31]. Our partially context-sensitive points-to analysis is most closely related to the context-sensitive analyses in our previous work [25], [26]. These approaches avoid the cost of nonselective 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 [25], [26], or partial constructor inlining as in our current algorithm. Other context-sensitive points-to analyses that seem to be substantially more costly than ours are presented in [57], [21], [58], [59]; these analysis algorithms implement nonselective context sensitivity.

6.2 Infeasible Paths

Bodik et al. present an algorithm for static detection of infeasible paths using branch correlation analysis, for the purposes of refining the computation of def-use coverage requirements in C programs [60]. Our data reachability analysis focuses on the detection of infeasible paths in Java which arise due to object-oriented features and idioms such as polymorphism; this is not addressed in [60]. Souter and Pollock present a methodology (without empirical

investigation) for demand-driven analysis for the detection of type infeasible call chains [61], [62]. 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 [61] and [62]. 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. [63] investigate 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 reenforces our observation of uncovered infeasible *e-c links* in our experiments, that involved complex control conditions which "fooled" the analysis.

7 Conclusions

We have defined an exception-flow analysis that is (according to our studies of benchmarks) precise enough to support the approach to white-box testing of fault-recovery code that we presented in [11]. 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. We hope this methodology will prove to be a valuable tool for developers of server applications that must provide high reliability and, thus, improve the experience of users who rely on such servers.

Exception-flow information derived solely from prior analysis techniques such as Class Hierarchy Analysis, Rapid Type Analysis, and Spark (a field-sensitive, flow-insensitive, and context-insensitive points-to analysis) is not suitable for our approach, as it contains too many infeasible links from exception throws to `catch` clauses. The most precise of these analyses found 179 *e-c links* in our set of four moderate-size benchmarks and 475 in our set of three larger benchmarks; these numbers dwarf the actual number of *e-c links* that are exercised during tests, 72 and 85, primarily because most of the *e-c links* are provably infeasible.

By performing inline substitution of constructors prior to exception-flow analysis based on Spark and, subsequently, pruning infeasible *e-c links* with our basic DataReach analysis, we can produce an analysis that finds only 86 *e-c links* in the moderate-sized benchmarks. Many of the 14 that are not exercised during tests are in fact infeasible. However, it is often difficult or impossible for static analysis to determine this fact, and we believe this number is small enough to permit manual examination by the tester (who must already supervise the testing process to check for appropriate behavior when a link is exercised). In terms of our approach to testing fault-recovery code, this corresponds to a measurement of 84 percent coverage. Most (but not all) of the lack of measured coverage is due to inaccuracy in analysis, but the number of false links that must be ruled out manually is still much smaller than the number of links that must be examined during testing.

Our basic DataReach algorithm still finds 229 *e-c links* in the set of larger benchmarks. This number can be reduced to 201 by applying the M-DataReach variant of our analysis, at a cost of about $8\frac{1}{2}$ minutes of additional analysis time for the three larger benchmarks (M-DataReach runs faster than the original analysis on one benchmark). Manual analysis suggests the number of *e-c links* could be reduced further by applying our V-DataReach variant, though we have not implemented and tested this algorithm. However, of the 201 *e-c links* found, 85 are exercised during testing (for 42 percent coverage), and at least 51 are uncovered due to the fact that the distributed data do not sufficiently test the software. Thus, the primary activities of the tester are once again the observation of relevant tests and the search for better test data, rather than manual examination of spurious *e-c links*.

The total analysis time varies from under five minutes for our smallest benchmark, up to almost two hours for full analysis including M-DataReach on one of the larger benchmarks. We believe this time is acceptable in the overall context of software testing.

Our future plans include testing application uses of other Java JDK libraries, such as *java.rmi*, and expanding our analysis to handle multinode programs and middleware that use configuration files for dynamic loading of classes. We also plan to investigate other uses of our analysis. Our precise exception-flow analysis may also prove valuable in contexts other than testing, for example, in helping programmers understand the exception handling structure of an unfamiliar program. Furthermore, our technique of using data reachability information to refine information about interprocedural control paths is not specific to the problem of exception flow, and could be applied to other analysis problems.

Acknowledgments

This work was supported in part by US National Science Foundation grants EIA-0103722 and CCR-9900988.

References

[1] J. Arlat, A. Costes, Y. Crouzet, J.-C. Laprie, and D. Powell, "Fault Injection and Dependability Evaluation of Fault-Tolerant Systems," *IEEE Trans. Computers,* vol. 42, no. 8, pp. 913-923, Aug. 1993.

[2] M. Cukier, R. Chandra, D. Henke, J. Pistole, and W.H. Sanders, "Fault Injection Based on a Partial View of the Global State of a Distributed System," *Proc. Symp. Reliable Distributed Systems,* pp. 168-177, 1999.

[3] S. Dawson, F. Jahanian, and T. Mitton, "ORCHESTRA: A Fault Injection Environment for Distributed Systems," *Proc. 26th Int'l Symp. Fault Tolerant Computing (FTCS-26),* pp. 404-414, June 1996.

[4] S. Han, K. Shin, and H. Rosenberg, "DOCTOR: An Integrated Software Fault Injection Environment for Distributed Real-Time Systems," *Proc. Int'l Computer Performance and Dependability Symp. (IPDS '95),* pp. 204-213, Apr. 1995.

[5] G.A. Kanawati, N.A. Kanawati, and J.A. Abraham, "FERRARI: A Tool for the Validation of System Dependability Properties," *Proc. 22nd Int'l Symp. Fault Tolerant Computing (FTCS-22),* pp. 336-344, 1992.

[6] 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," *Proc. 18th Int'l Symp. Fault-Tolerant Computing (FTCS-18),* pp. 102-107, 1988.

[7] R.V. Binder, *Testing Object-Oriented Systems.* Addison Wesley, 1999.

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

[9] R.G. Hamlet, "Testing Programs with the Aid of a Compiler," *IEEE Trans. Software Eng.,* vol. 3, no. 4, pp. 279-290, July 1977.

[10] S. Rapps and E. Weyuker, "Selecting Software Test Data Using Data Flow Information," *IEEE Trans. Software Eng.,* vol. 11, no. 4, pp. 367-375, Apr. 1985.

[11] C. Fu, R.P. Martin, K. Nagaraja, T.D. Nguyen, B.G. Ryder, and D. Wonnacott, "Compiler-Directed Program-Fault Coverage for Highly Available Java Internet Services," *Proc. Int'l Conf. Dependable Systems and Networks (DSN 2003),* June 2003.

[12] C. Fu, R.P. Martin, K. Nagaraja, T.D. Nguyen, B.G. Ryder, and D.G. Wonnacott, "Compiler-Directed Program-Fault Coverage for Highly Available Java Internet Services," Technical Report DCS-TR-518, Dept. of Computer Science, Rutgers Univ., Jan. 2003.

[13] C. Fu, B.G. Ryder, A. Milanova, and D. Wonnacott, "Testing of Java Web Services For Robustness," *Proc. Int'l Symp. Software Testing and Analysis (ISSTA),* pp. 23-33, July 2004.

[14] F. Tip and J. Palsberg, "Scalable Propagation-Based Call Graph Construction Algorithms," *Proc. Conf. Object-Oriented Programming, Languages, Systems and Applications,* pp. 281-293, Oct. 2000.

[15] K. Arnold and J. Gosling, *The Java Programming Language,* second ed. Addison-Wesley, 1997.

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

[17] A. Rountev, A. Milanova, and B.G. Ryder, "Points-To Analysis for Java Using Annotated Constraints," *Proc. Conf. Object-Oriented Programming, Languages, Systems and Applications,* pp. 43-55, 2001.

[18] Sable, McGill, "Soot: A Java Optimization Framework," http://www.sablemcgill.ca/soot, 2003.

[19] J. Dean, D. Grove, and C. Chambers, "Optimization of Object-Oriented Programs Using Static Class Hierarchy," *Proc. Ninth European Conf. Object-Oriented Programming (ECOOP '95),* pp. 77-101, 1995.

[20] D. Bacon and P. Sweeney, "Fast Static Analysis of C++ Virtual Functions Calls," *Proc. ACM SIGPLAN Conf. Object-Oriented Programing Systems, Languages and Applications (OOPSLA '96),* pp. 324-341, Oct. 1996.

[21] D. Grove and C. Chambers, "A Framework for Call Graph Construction Algorithms," *ACM Trans. Programming Languages and Systems (TOPLAS),* vol. 23, no. 6, 2001.

[22] T. Ogasawara, H. Komatsu, and T. Nakatani, "A Study of Exception Handling and Its Dynamic Optimization in Java," *Proc. ACM SIGPLAN Conf. Object-oriented Programing Systems, Languages and Applications (OOPSLA '01),* pp. 83-95, 2001.

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

[24] T.J. Marlowe and B.G. Ryder, "Properties of Data Flow Frameworks: A Unified Model," *Acta Informatica,* vol. 28, pp. 121-163, 1990.

[25] A. Milanova, A. Rountev, and B.G. Ryder, "Parameterized Object Sensitivity for Points-To Analysis for Java," *ACM Trans. Software Eng. and Methodology,* vol. 14, no. 1, Jan. 2005.

[26] A. Milanova, "Precise and Practical Flow Analsis of Object-Oriented Software," PhD dissertation, Rutgers Univ., 2003.

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

[28] M. Sharir and A. Pnueli, "Two Approaches to Interprocedural Data Flow Analysis," *Program Flow Analysis: Theory and Applications,* S. Muchnick and N. Jones, eds., Prentice Hall, pp. 189-234, 1981.

[29] O. Shivers, "Control-Flow Analysis of Higher-Order Languages," PhD dissertation, Carnegie Mellon Univ., 1991.

[30] O. Lhoták and L. Hendren, "Scaling Java Points-To Analysis Using Spark," *Proc. Int'l Conf. Compiler Construction,* pp. 153-169, 2003.

[31] B.G. Ryder, "Dimensions of Precision in Reference Analysis of Object-Oriented Programming Languages," *Proc. 12th Int'l Conf. Compiler Construction,* pp. 126-137, Apr. 2003.

[32] P. Sortokin, "Ftp Server in Java," http://peter.sorotokin.com/ftpd/ftpd.html, 2003.

[33] M.J. Radwin, "The Java Network File System," http://www.radwin.org/michael/projects/jnfs/, 2003.

[34] "The Muffin World Wide Web Filtering System," http://muffin.doit.org/, 2003.

[35] M. Welsh, D.E. Culler, and E.A. Brewer, "SEDA: An Architecture for Well-Conditioned, Scalable Internet Services," *Proc. Symp. Operating Systems Principles,* pp. 230-243, http://www.cs.harvard.edu/~mdw/proj/seda/, 2001.

[36] Apache Software Foundation, "Apache Jarkarta Project," http://jakarta.apache.org/, 2004.

[37] Specbench.org, "Spec jvm98 Benchmarks," http://www.spec.org/jvm98/, 1998.

[38] Volano LLC, "Volanomark," http://www.volano.com/benchmarks.html, 2004.

[39] D. Tang and R.K. Iyer, "Analysis and Modeling of Correlated Failures in Multicomputer Systems," *ACM Trans. Computer Systems,* pp. 567-577, May 1992.

[40] K. Nagaraja, X. Li, B. Zhang, R. Bianchini, R.P. Martin, and T.D. Nguyen, "Using Fault Injection to Evaluate the Performability of Cluster-Based Services," *Proc. Fourth USENIX Symp. Internet Technologies and Systems (USITS 2003),* Mar. 2003.

[41] Apache Software Foundation, "Apache Jakarta Tomcat," http://jakarta.apache.org/tomcat/, 2004.

[42] W.G. Bouricius, W.C. Carter, and P. Schneider, "Reliability Modeling Techniques for Self Repairing Computer Systems," *Proc. 24th Nat'l Conf. the ACM,* pp. 295-309, Mar. 1969.

[43] T. Tsai, M. Hsueh, H. Zhao, Z. Kalbarczyk, and R. Iyer, "Stress-Based and Path-Based Fault Injection," *IEEE Trans. Computers,* vol. 48, no. 11, pp. 1183-1201, Nov. 1999.

[44] J. Bieman, D. Dreilinger, and L. Lin, "Using Fault Injection to Increase Software Test Coverage," *Proc. Seventh Int'l Symp. Software Reliability Eng. (ISSRE '96),* pp. 166-174, 1996.

[45] D. Hamlet, "Foundations of Software Testing: Dependability Theory," *Proc. Second ACM SIGSOFT Symp. Foundations of Software Eng.,* pp. 128-139, 1994.

[46] D. Tang and H. Hecht, "An Approach to Measuring and Assessing Dependability for Critical Software Systems," *Proc. Eighth Int'l Symp. Software Reliability Eng.,* pp. 192-202, Nov. 1997.

[47] P. Frankl and E. Weyuker, "An Applicable Family of Data Flow Testing Criteria," *IEEE Trans. Software Eng.,* vol. 14, no. 10, pp. 1483-1498, Oct. 1988.

[48] S. Sinha and M.J. Harrold, "Criteria for Testing Exception-Handling Constructs in Java Programs," *Proc. Int'l Conf. Software Maintenance,* 1999.

[49] M.P. Robillard and G.C. Murphy, "Static Analysis to Support the Evolution of Exception Structure in Object-Oriented Systems," *ACM Trans. Software Eng. and Methodology (TOSEM),* vol. 12, no. 2, pp. 191-221, 2003.

[50] J.-W. Jo, B.-M. Chang, K. Yi, and K.-M. Cho, "An Uncaught Exception Analysis for Java," *J. Systems and Software,* 2004.

[51] N. Heintze, "Set-Based Analysis of Ml Programs," *Proc. ACM Conf. Lisp and Functional Programmig,* pp. 306-317, 1994.

[52] S. Sinha and M.J. Harrold, "Analysis and Testing of Programs With Exception-Handling Constructs," *IEEE Trans. Software Eng.,* vol. 26, no. 9, pp. 849-871, Sept. 2000.

[53] S. Sinha, A. Orso, and M.J. Harrold, "Automated Support for Development, Maintenance, and Testing in the Presence of Implicit Control Flow," *Proc. 26th Int'l Conf. Software Eng. (ICSE '04),* 2004.

[54] J.-D. Choi, D. Grove, M. Hind, and V. Sarkar, "Efficient and Precise Modeling of Exceptions for Analysis of Java Programs," *Proc. ACM SIGPLAN-SIGSOFT Workshop Program Analysis for Software Tools and Eng.,* pp. 21-31, Sept. 1999.

[55] S. Lee, B.-S. Yang, S. Kim, S. Park, S.-M. Moon, K. Ebcioglu, and E. Altman, "Efficient Java Exception Handling in Just-in-Time Compilation," *Proc. ACM SIGPLAN Java Grande Conf.,* 2000.

[56] M. Cierniak, G.-Y. Lueh, and J.M. Stichnoth, "Practicing Judo: Java under Dynamic Optimzations," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* pp. 13-26, 2000.

[57] J.D.D. Grove, G. DeFouw, and C. Chambers, "Call Graph Construction in Object-Oriented Languages," *Proc. ACM SIGPLAN Conf. Object-Oriented Programing Systems, Languages and Applications (OOPSLA '97),* pp. 108-124, Oct. 1997.

[58] R. O'Callahan, "The Generalized Aliasing as a Basis for Software Tools," PhD dissertation, Carnegie Mellon Univ., 2000.

[59] R. Chatterjee, B.G. Ryder, and W.A. Landi, "Relevant Context Inference," *Proc. ACM SIGACT/SIGPLAN Symp. Principles of Programming Languages,* Jan. 1999.
[60] R. Bodik, R. Gupta, and M.L. Soffa, "Refining Data Flow Information Using Infeasible Paths," *Proc. Sixth European Software Eng. Conf. (ESEC/FSE 97),* M. Jazayeri and H. Schauer, eds., pp. 361-377, 1997.
[61] A.L. Souter and L.L. Pollock, "Type Infeasible Call Chains," *Proc. IEEE Int'l Workshop Source Code Analysis and Manipulation,* 2001.
[62] A.L. Souter and L.L. Pollock, "Characterization and Automatic Identification of Type Infeasible Call Chains," *Information and Software Technology,* vol. 44, no. 13, pp. 721-732, Oct. 2002.
[63] A. Rountev, S. Kagan, and M. Gibas, "Static and Dynamic Analysis of Call Chains in Java," *Proc. Int'l Symp. Software Testing and Analysis,* pp. 1-11, July 2004.

Chen Fu received MS degree in computer science from the Institute of Computing Technology, Chinese Academy of Science, China. He is currently a PhD candidate in Computer Science in Rutgers University. His research interests are in program analysis and its application in testing and program understanding tools.

Ana Milanova received the PhD degree in computer science from Rutgers University in 2003. She is currently an assistant professor in the Department of Computer Science at Rensselaer Polytechnic Institute. Her research interests focus on static and dynamic program analysis and its applications in software productivity tools and optimizing compilers. She is a member of the IEEE Computer Society, ACM, SIGSOFT, and SIGPLAN.

Barbara Gershon Ryder is a professor of computer science at Rutgers University, New Brunswick, New Jersey. She became a fellow of the ACM in 1998 and was selected as a CRA-W Distinguished Professor in 2004. She was selected as Professor of the Year for Excellence in Teaching by the Computer Science Graduate Students Society of Rutgers University in 2003 and received the ACM SIGPLAN Distinguished Service Award in 2001. She was the general chair of the 2003 Federated Conference on Research in Computing and served on the board of directors of the Computer Research Association (CRA) from 1998-2001. She was elected a member of ACM Council in 2000 and 2004, and served on the ACM SIGPLAN Executive Committee from 1989-1999 (as SIGPLAN Chair, 1995-1997). She was a recipient of a US National Science Foundation Faculty Award for Women Scientists and Engineers (1991-1996). Dr. Ryder's research focuses on static and dynamic program analyses for object-oriented languages and practical software tools. Applications include: change impact analysis, program understanding, software testing, and testing availability of Web services.

David G. Wonnacott recieved the PhD degree in computer science from The University of Maryland in 1995, for his work on the Omega Test and Omega Library. His current research interests include the creation and use of static analysis algorithms and computer science education. He is now an associate professor at Haverford College, and a member of the ACM and SIGPLAN.