1 234

Robustness Testing of Java Server Applications

Chen Fu, Ana Milanova, Barbara G. Ryder, David Wonnacott

Should we say something about the special issue here? Only if our paper is technically "invited".

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

Index Terms— **Reliability, Def-Use Testing, Java, Exceptions, Test Coverage Metrics**

FIX: Need to put in real entries in bib.bib for issta04, tipPalsberg, and tosem-issta04 – leaving these out entirely confused the IEEEtrans.bst

I. Introduction

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

Traditional fault-injection testing of software in the operating system community is conducted in a black-box manner, using a probabilistic analysis to determine whether or not a software component will work properly when subjected to specific fault loads and workloads [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 correct 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 web services are written using components with unknown internal structure, testers need to identify vulnerabilities to system problems automatically (i.e., with the help of software tools). The testing of error recovery code in web services is necessary for ensuring the high reliability required of these systems.

Our methodology uses the tools of white-box def-use testing to aid a tester of web services in this task. There is a large body of existing work on *white-box* testing methodologies [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

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

Chen Fu (`chenfu@cs.rutgers.edu`) and Barbara Ryder (`ryder@cs.rutgers.edu`) are with the Rutgers University Department of Computer Science, Piscataway, NJ 08854

Ana Milanova (`milanova@cs.rpi.edu`) is with the Rensselaer Polytechnic Institute Department of Computer Science, Troy, NY 12180

David Wonnacott (`davew@cs.haverford.edu`) is with the Haverford College Department of Computer Science, Haverford, PA 19041

almost always left unexecuted in traditional white-box testing, because it may not be executable by merely manipulating program inputs.

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

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

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

In a previous paper [10], we gave a general overview of our methodology for testing of error recovery code, and defined appropriate coverage metrics. We presented a proof-of-concept case study in which a proxy server application was instrumented by hand, and then fault injection was performed and recorded by executing the instrumentation. In this paper we have defined and implemented a compile-time exception-catch link analysis, fully automated the program instrumentation process, and experimented with several versions of analysis on a data set of moderately-sized web service applications.

The specific contributions of this paper are:

- Design of a new compile-time exception-catch link analysis to identify error recovery code in relation to certain resource usage program points (i.e., a def-use analysis for potential exceptions involving resource usage). This analysis essentially is an interprocedural def-use dataflow analysis calculation with two new refinements: (i) performing a points-to analysis using limited context sensitivity by inlining constructors that set object fields (in order to avoid conflating objects, especially in libraries with long call chains) and (ii) using the absence of data reachability through object references to confirm the *infeasibility* of some links, by showing the corresponding interprocedural paths to be infeasible.
- Demonstration of *automatic* program instrumentation directed by our analysis, that effectively constructs a compiler-directed fault injection engine from *Mendosus* [11], an existing fault injection framework.
- Empirical validation of our methodology using several mode-rately-sized Java web service applications, including comparison of our new analysis with less precise, less costly class-based analysis adapted to find exception-flow def-uses. These studies demonstrate the appropriateness of the precision of our analysis for this task, in that on average, 84% of all exception-flow def-use links are covered by the testing.

Overview. The rest of this paper is organized as follows. In Section II we describe our coverage metric, which is a slight variant of the original metric described in [10], and give an overview of the compiler-directed fault injection methodology. In Section III, we discuss our compile-time analysis for exception-flow def-uses and its precision increasing refinements. In Section IV we report our empirical results on moderate-sized Java applications, describing the impact on the exception-flow def-uses obtained, of varying the compile-time analysis used. In Section V we describe related work. Finally, we present our conclusions.

II. Measuring Coverage of FaultHandling 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 [12]. Each `catch` block is potentially the starting point of error recovery code for a matching error/exception raised during the lifetime of the corresponding `try` block.

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

In any given program execution, each element of P could possibly produce an exception that reaches some subset of the program's `catch` blocks. By viewing fault-sensitive operations as the definition points of exceptions, and `catch` blocks as uses of exceptions, we can define a coverage metric in terms of *exception-catch (e-c) links.*

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 [10] (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 1 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.

In the terms used by traditional def-use analysis [13], fault def-catch coverage is an *all-def-uses* metric with faults counting as *def*s; exception def-catch coverage can be seen as an *all-def-uses* metric with exceptions counting as *def*s, or as a metric that is stricter than *all-uses* when faults are viewed as *def*s. **FIX: We need to check the above against the actual definitions of the terms** . For a more detailed discussion of possible coverage metrics for fault-tolerant code, see [10], [14].

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

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

The compiler uses the set of *e-c links* found to identify where to place the instrumentation that will communicate with *Mendosus* [11], the fault injection engine, during execution. This communication will request the injection of a particular fault when execution reaches the `try` block containing the vulnerable operation and will result in the recording of the execution of the corresponding `catch` block.

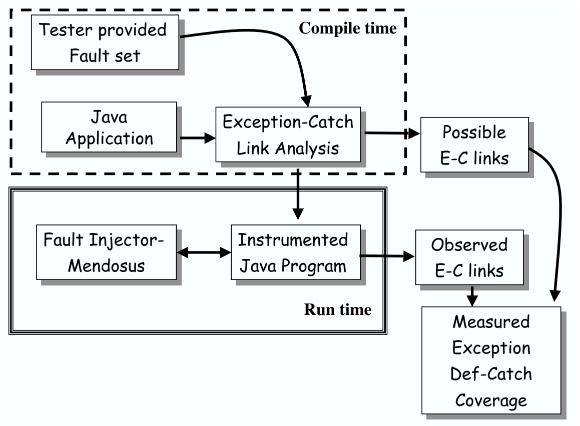

Tester provided Fault set
Compile time
Java Application
Exception-Catch Link Analysis
Possible E-C links
Fault Injector-Mendosus
Instrumented Java Program
Observed E-C links
Run time
Measured Exception Def-Catch Coverage

Fig. 1. Compiler-directed fault injection framework

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

III. Compile-time Analysis

Figure 2 illustrates the high level structure of the two-phased compile-time exception-catch link analysis which we designed to calculate *e-c links* in Java programs. **Exception-flow** analysis takes a static representation (i.e., AST) of a Java program as well as its call graph, and produces the *e-c link* set of the given program. Unlike previous exception-flow analysis [15], [16], [17] which relied on interprocedural propagation of exception types, our analysis is object-based, distinguishing between exception objects created by different `new()` statements. The **DataReach** analysis serves as a post-pass filter which uses the reference points-to graph [18], [19] of the program to discard as many infeasible *e-c links* in the set produced by exception-flow analysis as possible, so as to increase the precision of the entire analysis. Intuitively, both of these analysis phases can vary in their precision, because they effectively are parameterized by the points-to and call graph construction analysis used as their inputs. Various analysis choices are available for call graph construction [20], [21], [22] which differ in their cost and the precision of the resulting graph. The empirical results discussed in Section IV 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.

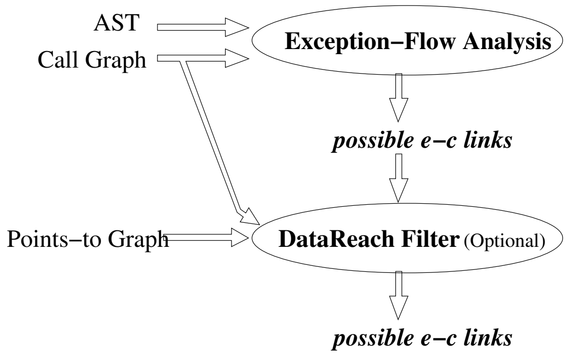


Fig. 2. Two phases of exception-catch link analysis

A. 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 [12]. We want to find the relationship between `catch` blocks and fault-sensitive operations. We use "`throw` statement" to represent all fault-sensitive operations in our discussions for simplicity; we actually mean all instructions or calls that may throw some exception, if a fault occurs.

A naive analysis that relies only on examination of user declared exception types in `catch` blocks and method signatures is too inaccurate to yield information of practical use. In part this is because the declared exception can be a supertype, subsuming many exception types that actually cannot be thrown in this context. Moreover, a method may declare that some exception may be thrown, when actually no exceptions can ever be raised; this can occur when the implementation of some method has changed, but the method declaration is not updated. Dynamic dispatch can add to the imprecision of the declared exception information. Suppose class `A` is the superclass of `B` and method `bar()` is declared in both of them, but only `A.bar()` may throw an exception of class `E` when called. If some other method `foo()` contains a call `a.bar()` for `a` of static type `A`, then `foo()` must define a handler for exception `E` or declare that it throws this exception. However if at runtime reference `a` always points to a `B` object, no exception can ever be thrown at the call site.

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

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

Exception-flow as a dataflow analysis. We define *exception-flow* as the flow of each exception object thrown per `throw` statement along the exception handing path [23] — from the `throw` statement to the `catch` block where it is handled.

According to the semantics of exception handling in Java [12], 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 [24] dataflow analysis to this problem, but there are some unique aspects of exception-flow that require special handling:

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

[2]In Java, `try` blocks can be nested within each other. Handlers are associated with exceptions in inner to outer order [12].

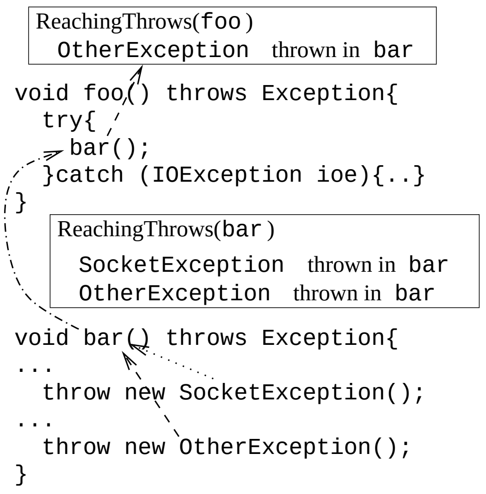
[3]Adding these annotations is not difficult for any call graph construction algorithm.

blocks may also yield unhandled exceptions, depending on the types of the respective `catch` blocks. Thus, the program representation used is a variant of a call graph, where each method node has an inner structure consisting of an edge from the entry node to each uncovered `throw` or method call, and an edge to each outermost `try-catch` block.

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

Definition (*ReachingThrows(method M)*): The set of all `throw` statements for which there exists an exception handling path [23] from the throw statement to method M, and the exceptions are not handled in method M.

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


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

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

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

Fig. 3. Example of ReachingThrows

The dataflow equations for the *ReachingThrows* problem are defined on the annotated call graph of the program.**FIX: need more detail on the finally handling -BGR** [4] We define *RT(m)*, the ReachingThrows at the entry to method m, as

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

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

These dataflow equations are consistent with the definition of a monotone dataflow analysis framework [25] and therefore, amenable to fixed-point iteration.[5]

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

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

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

B. Data reachability analysis

We want to use a fairly precise program analysis to eliminate as many infeasible interprocedural paths as possible, to reduce

[4]Under certain conditions[12], `finally`s behave like `catch`es and/or `throws`. Our algorithm handles these situations correctly, but we omit the details involving `finally`s for brevity.

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

the work that otherwise must be done by human testers when *e-c links* based on these paths cannot be covered. Using a more precise analysis for call graph construction such as points-to analysis [18], [19] helps to reduce the number of infeasible *e-c links* found. However, in practice even a very precise call graph building algorithm introduces many infeasible *e-c links*. Figure 4 is an example of typical use of the Java network-disk I/O packages. Figure 5 illustrates how infeasible *e-c links* are introduced even given a fairly precise call graph for the code. As we can see, the `try` block in `readFile` is only sensitive to disk faults and the `try` block in `readNet` is only sensitive to network faults. But exception-flow information is merged in `BufferedInputStream.fill()`[6] and propagated to both `readFile` and `readNet`; thus, two infeasible *e-c links* are introduced reducing the acheiveable runtime coverage to less than 50%.

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

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

Fig. 4. Code Example for Java I/O Usage

This inaccuracy can be resolved by using a different program representation such as a call tree [28] instead of a call graph. However, constructing a call tree by compile-time analysis is too expensive and once constructed, this representation is too large to scale appropriately. For example, to remove the infeasible *e-c links* in Figure 5, the call tree algorithm must be able to find that there are only 2 feasible call chains which share a middle segment of length 3. Separating these 2 chains would require a context-sensitive points-to analysis analogous to 4-CFA [29], [30], 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 inter-procedural paths requiring receiver objects of a specific

[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.

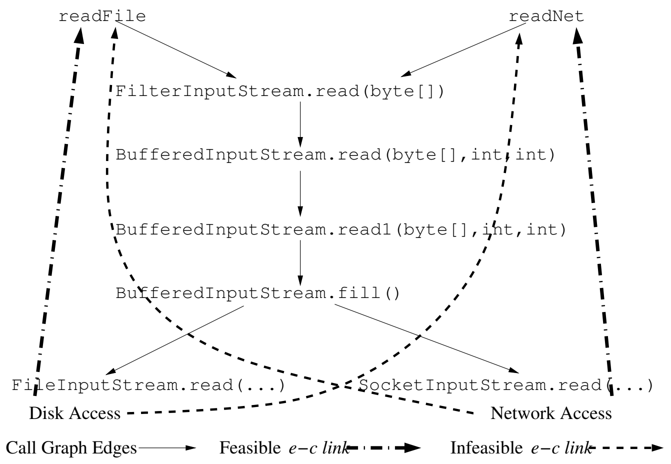

readFile
readNet
FilterInputStream.read(byte[])
BufferedInputStream.read(byte[],int,int)
BufferedInputStream.read1(byte[],int,int)
BufferedInputStream.fill()
FileInputStream.read(...)
SocketInputStream.read(...)
Disk Access
Network Access
Call Graph Edges
Feasible e−c link
Infeasible e−c link

Fig. 5. Call Graph for Java I/O Usage

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

FIX: ANA: item 1 does not seem right. Does not mention field dereferences of objects created during the lifetime of the call. BGR fixed in 2nd item; is this ok?

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

Therefore given an original call site, we can express the feasibility of a particular call path in terms of whether some data reachability is possible according to these conditions. For example, to show that the *e-c link* referred to above is infeasible, we verify that there is no object in the points-to set of the receiver of the target call site with type `SocketInputStream` that can either be created in one of the methods reachable from the original call, or reachable by transitive field loads from the receiver or the arguments of the original call site or static fields. This means that the exception-flow def-use path is infeasible. Note, we only consider object fields and static fields loaded in *methods reachable from the original call*. Clearly, we need reasonably precise points-to information [31], [18] to obtain the high-quality data reachability information.

The rest of this section describes the original data reachability algorithm from [32] using a declarative constraint formalism from [33]. In addition, it describes the imprecisions of the original data reachability algorithm. Section **??** presents a schema of successively more precise data reachability algorithms.

1) Original DataReach Algorithm: In previous work [32] we introduced a data reachablility algorithm referred to as *DataReach*. This algorithm requires as input a points-to graph [31], [18]. 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 [**?**], [**?**]. Auxiliary 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 the set of object names 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 [33]). The statement of these constraints is followed with 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 $e.m(\ldots)$ occurring 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 static call site $C.M'(\ldots)$ in M:
   $(M \in R) \Rightarrow M' \in R$
3) For each method M and for each object creation statement s_i: $\ldots = new\ o_i$ in M:
   $(M \in R) \Rightarrow o_i \in U$
4) For each method M and for each static field read statement s_i: $\ldots = C.f$ in M:
   $(M \in R) \Rightarrow Pt(C.f) \subseteq U$
5) For each method M and for each instance field read statement s_i: $\ldots = r.f$ in M:
   $(M \in R) \Rightarrow f \in F$
6) $(o \in U \wedge f \in F) \Rightarrow Pt(o.f) \subseteq U$

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, 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. Constraint 3 specifies that an object is added to set U whenever there is an object creation statement in a reachable method; similarly constraint 4 specifies that objects are added to U whenever a static field is accessed. Finally, constraint 5 collects the set of field identifiers accessed in reachable methods, and constraint 6 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 feasible (i.e., executable) path from that call site. The algorithm starts from the given call site, does a breadth-first search on the call graph 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 a fault occurs during the lifetime of the original call, then an exception may be handled by a `catch` block associated with the `try` in which the original call site is nested. In this case, there is a corresponding *e-c link* resulting from an excepting call to some method m or `throw` in method m during the lifetime of the original call. If at the target call to m, the set of possible target methods does not contain m, then the *e-c link* is spurious (i.e., it corresponds to an infeasible control-flow path); thus, there is no need for this link to be exercised.

2) Imprecision of DataReach: The original data reachability algorithm produced relatively precise results which led to an average of 85% *e-c link* coverage on an initial set of benchmarks [32]. However, it estimates U relatively conservatively; examples from several new benchmark programs reveal that in many cases this conservative estimate is not sufficient. Therefore, there is a need to investigate more precise analyses. Consider the sample set of statements in Figure 6 and DataReach analysis starting at original call $c1$ 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 analyses 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 (or in a subtype of `Hashtable`) 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 the original call $c2$ in method `Read2`, the analysis determines that both the `throw` in `Y.read` and the `throw` in `Z.read` will be handled by the `catch` block in method `Read2`. It is easy to see that the only two feasible *e-c links* are (i) between `throw new SomeIOException` and the `catch` in `Read1`, and (ii) between `throw new OtherIOException` and the `catch` in `Read2`.

C. 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 [33], 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 each for U and F. The new data reachability algorithms in our schema keep separate sets for program entities such as methods, classes and reference variables. The major differences with Tip and Palsberg's algorithms are that (i) our algorithm propagates objects rather than class types, and (ii) it is formulated on a *partial* program rather than on a whole-program. The algorithms in our schema keep specialized local information for program entities such as methods and reference variables, which makes possible increased precision for data reachability calculations. For example, consider the set of statements in Figure 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, sets U_M are kept for each method M.

This paper discusses three instantiations of the schema: one set U valid throughout the data reachability analysis (this instantiation corresponds to the original DataReach algorithm), separate sets U_M for each method M, and separate sets U_V for each reference variable V. It is possible to define an algorithm, where there is a set per each class (i.e., aggregating the method sets for all methods in that class into a single set U_C); for brevity we omit a detailed discussion of this instantiation.

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, instead of calculating one U and one F set program-wide. Analogously to [33], $ParamTypes(M)$ 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 method M at original call
  $Pt(v) \subseteq U_M$ for each actual argument v at original call and for each target M
  $U_N = \emptyset$ for each non-target method N
  $F_M = \emptyset$ for each method M

1) For each method M, each virtual call site $e.m(\ldots)$ occurring in M, each object $o \in Pt(e)$ where $StaticLookup(o, m) = M'$:
   $(M \in R) \wedge (o \in U_M) \Rightarrow$
   $$\begin{cases} M' \in R \wedge \\ MatchingObjects(ParamTypes(M'), U_M) \subseteq U_{M'} \wedge \\ MatchingObjects(ReturnType(M'), U_{M'}) \subseteq U_M \wedge \\ o \in U_{M'} \end{cases}$$
2) For each method M and for each static call site $C.M'(\ldots)$ in M:
   $(M \in R) \Rightarrow$
   $$\begin{cases} M' \in R \wedge \\ MatchingObjects(ParamTypes(M'), U_M) \subseteq U_{M'} \wedge \\ MatchingObjects(ReturnType(M'), U_{M'}) \subseteq U_M \end{cases}$$
3) For each method M and for each object creation statement s_i: $\ldots = new\ o_i$ in M:
   $(M \in R) \Rightarrow o_i \in U_M$
4) For each method M and for each static field read statement s_i: $\ldots = C.f$ in M:
   $(M \in R) \Rightarrow Pt(C.f) \subseteq U_M$
5) For each method M and for each instance field read statement s_i: $\ldots = r.f$ in M:
   $(M \in R) \Rightarrow f \in F_M$
6) $(o \in U_M \wedge f \in F_M) \Rightarrow Pt(o.f) \subseteq U_M$

Intuitively, constraints 1 and 2 refine the analogous constraints from DataReach, respectively. First, the receiver object o at a virtual call should be available in the universe for the method M enclosing the call. Second, 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. Third, 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. Constraints 3 and 4 respectively gather objects created in M, and objects that flow to M due to static field reads. Finally, constraint 5 gathers the set of instance fields that may be accessed in M and constraint 6 accounts for the computation of the transitive closure of U_M (the closure is found by traversing the points-to graph starting from the objects in U_M with respect only to fields in F_M, that is, the fields that are accessed in M).

Example. Consider the code in Figure 6. After initialization at original call $c1$ we have $U_{A.m} = \{o_1, o_2\}$. Applying constraint 3 at call `n(x)` results in objects o_1 and o_2 being added to the upper level universe of `A.n`; no objects flow back to $U_{A.m}$. Clearly, no fields are accessed in `A.m` and therefore the closure of the universe 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`. This is a simplified example which illustrates a frequently occurring situation in benchmark code.

```
abstract class X {
     void abstract read() throws IOException

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

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

```
public void Read1() {
try {
  s1: A a = new A();
  s2: Y y = new Y();
  c1: a.m(y);
  }
  catch (IOException) { ... }
}


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

Fig. 6. Imprecision of DataReach algorithm

2) Separate sets for variables (V-DataReach): Additional precision over M-DataReach can be achieved by distinguishing the object sets for each reference variable and reference object field. For this instantiation of the schema, called V-DataReach, the algorithm keeps distinct sets U_V and $U_{o.f}$ for each reference variable V and for each reference object fields $o.f$. The predicate $MethodLocal(o)$ returns true if object o does not escape the method where it is created (i.e., the lifetime of the object does not exceed the lifetime of the call); it returns false otherwise. This information can be trivially computed from a points-to graph as shown in [18].

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

- **input:** $Pt : R \to \mathcal{P}(O)$
- **initialize:** $M \in R$ for each target M at original call
  Initialize U_p for formals p of targets accordingly
  Initialize all other U_v and $U_{o.f}$ 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 $StaticLookup(o, m) = M'$:
   $(M \in R) \wedge (o \in U_e) \Rightarrow$
   $\begin{cases} M' \in R \wedge \\ U_{e_i} \subseteq U_{f_i} \text{where } f_i \text{ are the formal parameters of } M' \wedge \\ U_{M'.ret\_var} \subseteq U_l \wedge \\ o \in U_{M'.this} \end{cases}$
2) For each method M and for each static call site
   $l = C.M'(e_1, \ldots, e_n)$ in M:
   $(M \in R) \Rightarrow$
   $\begin{cases} M' \in R \wedge \\ U_{e_i} \subseteq U_{p_i} \text{where } p_i \text{ are the formal parameters of } M' \wedge \\ U_{M'.ret\_var} \subseteq U_l \end{cases}$
3) For each method M and for each reference assignment statement s_i: $l = r$ in M:
   $(M \in R) \Rightarrow U_r \subseteq U_l$
4) For each method M and for each object creation statement s_i: $l = new\ o_i$ in M:
   $(M \in R) \Rightarrow o_i \in U_l$
5) For each method M and for each static field read statement $l = C.f$ in M:
   $(M \in R) \Rightarrow Pt(C.f) \subseteq U_l$
6) For each method M, for each instance field write statement $l.f = r$ in M and each $o_i \in Pt(l)$ where $MethodLocal(o_i)$:
   $(M \in R \wedge o_i \in U_l) \Rightarrow U_r \subseteq U_{o_i.f}$
7) For each method M, for each instance field read statement $l = r.f$ in M and each $o_i \in Pt(r)$:
   $(M \in R \wedge o_i \in U_r) \Rightarrow$
   $\begin{cases} MethodLocal(o_i) \Rightarrow U_{o_i.f} \subseteq U_l \wedge \\ \not{M}ethodLocal(o_i) \Rightarrow Pt(o_i.f) \subseteq U_l \end{cases}$

Intuitively, constraints 1,2,3,4 and 5 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 as it is illustrated by the following example.

Example. Consider the set of statements in Figure 7. Starting from original call `c1: a.m(...)` in `Read1` DataReach will compute $U = \{o_1, o_2, o_3\}$. At target call `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

```
abstract class X {
  void abstract read() throws IOException

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

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

class B {
  s1: public static X xy = new Y();
  s2: public static X xz = new Z();
  ...
  }
```

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


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

Fig. 7. Imprecision of M-DataReach algorithm

o_2 are in U, they are determined to be valid receivers; therefore, the `throw SomeIOException` and the `throw OtherIOException` statements flow to the catch in `Read1`. The same situation occurs when DataReach starts from original call `c2: a.m(...)` in `Read2`. The same imprecision occurs with M-DataReach because it computes a single set $U_{A.m}$. V-DataReach is able to avoid this imprecision because it keeps separate sets U_{x1} and U_{x2} for $x1$ and $x2$ respectively.

Constraints 6 and 7 refine constraint 6 from M-DataReach. The role of the first constraint is to separate instance field writes to objects whose lifetime does not exceed the lifetime of the original call. For those objects, all field writes occur during the lifetime of the original call and the values assigned to fields can be collected from the right-hand-side of the field write statement in set $U_{o.f}$. Constraint 7 accounts for propagating field values. For objects o whose lifetime does not exceed the lifetime of the original call, the values of an accessed field f are collected from the corresponding set $U_{o.f}$. For objects whose lifetime may exceed the lifetime of the original call (i.e., the object escapes the original call, or it is created before control enters the original call) the possible field values are approximated from the global points-to solution since those fields may be set outside of the original call. The following example taken from the *HttpClient* benchmark illustrates this situation.

Example. Consider the example in Figure 11. Starting V-DataReach from original call $c1$ 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., it's 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 6 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 7 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`. 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) Conclusions: This section summarizes our algorithms. For a given program let $\mathcal{C}$ be the number of classes, $\mathcal{M}$ be the number of methods, $\mathcal{V}$ be the number of reference variables, including static fields, $\mathcal{O}$ be the number of object allocation sites, and $\mathcal{F}$ be the number of instance field identifiers.

The complexity of a data reachability analysis that fits our schema depends on the number k of U sets kept during propagation. The overall complexity can be broken into three components: (i) the complexity of generating inclusion constraints for program statements (constraints 1-4 for DataReach and M-DataReach, and 1-5 for V-DataReach), (ii) the complexity of solving the system of inclusion constraints, and (iii) the complexity of computing the field closure for sets U (constraints 5 and 6 for DataReach and M-DataReach and 6 and 7 for V-DataReach). Clearly the complexity of (i) is dominated by the time to process virtual calls which is $O(\mathcal{O} * E)$ where E is the number of call graph edges. The complexity of (ii) is $O(\mathcal{O} * k^2)$ (for each set U_i at most $\mathcal{O}$

```
class A {
  void read() throws IOException; }

class Dmy extends A {
  void read() { // do nothing. } }

class Res extends X {
  public void read() throws IOException {
    // Code reads disk files and throws
    // IOException
  } }

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

```
class M {
  void getData(A a) throws IOException {
    W w = new W(a); //o1
    w.read(); }

  void getDmy() {
    try {
  c1: getData(new Dmy()); //o2
  } catch (IOException e) {...} }


  void getRes() {
    try {
  c2: getData(new Res()); //o3
  } catch (IOException e) {...} }
}
```

Fig. 8. Imprecision of M-DataReach algorithm

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

The following table summarizes our analyses in order of growing precision and complexity:

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

IV. EMPIRICAL RESULTS

In this section we discuss the instrumentation used in our methodology, report our empirical findings, and discuss some case histories from our experiments. Initial findings on a set of four moderate-sized web server applications have been reported previously in [32]. In this paper we report the results of additional analyses applied to these programs and present extensive case studies of them. New experiments with three additional, larger applications, including one written with the *Tomcat* framework, are presented and discussed as well.

A. Instrumentation

The methodology described in Section II requires that the Java program be instrumented to report coverage of the *e-c links* exercised and to communicate with *Mendosus* to request specific faults. A detailed description of the methodology was described in our previous paper [10]; we briefly summarize it here.

The instrumentation is accomplished through method calls. For each *e-c link* (p, c), we first locate the `catch` block c and the corresponding `try` block. At the entry of the `try` block, a special method call is inserted to direct *Mendosus* to inject the fault selected at static instrumentation time. At the entry of the `catch` block, another method call is inserted to query and record the call stack encapsulated in the caught exception. The instrumentation methods called are designed so that each instrumentation point can be turned on and off by a command line option. Note that the fault must be selected so that exactly one fault-sensitive operation will fail and throw an exception. In addition, we collect the set of I/O objects created in user code during execution, in order to limit the scope of the injected faults to this set, so that I/O operations conducted by Java virtual machine instead of user code will not be affected, which include, for instance, security policy loading and class loading.

B. Experimental setup & benchmarks

We implemented Exception-flow analysis and DataReach analysis as two separate modules in the Java analysis and transformation framework Soot [19] version 2.0.1, using a 2.8GHz P-IV PC with Linux 2.4.20-13.9 and the SUN JVM 1.3.1_08 for Linux. By separating the two phases of our analysis, we were able to show the gains from adding the DataReach postpass. Soot provides a call graph builder using *Class Hierarchy Analysis* (CHA) [20], and *Spark*, a field-sensitive, flow-insensitive and context-insensitive points-to analysis (a form of 0-CFA)[30], [34], [18], [31]. We implemented another call graph builder using *Rapid Type Analysis* (RTA)[21]. The instrumentation phase is also implemented as a separate module in Soot.

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 plus use DataReach as a postpass filter.
6) InPTA-DR — Use Spark plus selective constructor inlining to provide the points-to graph and the call graph, and use DataReach as a postpass filter.
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 by Peter Sorotokin v0.6
- JNFS, a server application that runs on top of a native file system and listens to and handles requests for both read and write accesses to files. The server communicates with various clients via RMI [35]
- Muffin, a web filtering proxy server [36]
- Haboob, a simple web server based on SEDA, a staged event-driven architecture [37]
- HttpClient, an HTTP utility package from the *Apache Jakarta Project* [38]. We collected its unit tests to form a whole program to serve as a benchmark.
- SpecJVM, a standard benchmark suite[39] 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* [40] — a web based chat server. The benchmark includes the chat server and simulated client

Column 2 of Table II shows the number of user classes, with those in parentheses comprising the JDK library classes reachable from each application. The data in column 3 are the number of user methods and those in parenthesis are the JDK library methods reachable from each application. Column 4 gives the number of lines of code in user code source files, when available. 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, with the lines of code calculated using the UNIX *wc* utility. JNFS is the only multi-node application.[8] Also note that we run all the benchmarks in SpecJVM together as one Java program, because I/O module in SpecJVM is shared across all the benchmarks.

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.

[7]Selective constructor inlining and 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.

TABLE II
BENCHMARKS

Name	*Classes*	*Methods*	*LOC*	*.class Size*
FTPD	11(1407)	128(7479)	2783	39,218
JNFS	56(1664)	447(9603)	10478	175,297
Muffin	278(1365)	2080(7677)	32892	727,118
Haboob	338(1403)	1323(7432)	39948	731,413
HttpClient	252(2210)	1334(4741)	61405	1,049,784
SpecJVM	484(2161)	2489(4592)	N/A	2,817,687
VMark	307(2266)	1565(5029)	N/A	2,902,947

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

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

C. Empirical data

Table III lists the number of *e-c links* reported for each benchmark in each analysis configuration. Column 9 lists the number of links, amoung those discovered in InPTA-MDR, whose corresponding `try` block was executed by a test execution. The last column shows the number of *e-c links* actually covered for each benchmark in the fault injection testing. Table IV is the overall exception def-catch coverage for all the benchmarks derived from the data in Table III. 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 2 different calculations for the percentage *e-c links* covered. In columns 2-8 of Table IV, we use the metric described in Section II (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 IV, 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 lets us distinguish 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 III). 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 III). For the *e-c links* reported by InPTA-DR, the coverage percentage of the four smaller benchmarks was stabilized at approximately 84% with small variance. In Muffin and HttpClient, the additional precision helped cut the number of reported *e-c links* by more than half. Haboob is special because it is the only benchmark that uses a self-constructed non-blocking network library, which does not have as much polymorphism as the standard JDK library. Thus the simple PTA analysis is sufficient to analyze Haboob, as shown in Table III. 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 III).

More disappointingly, on the larger benchmarks the coverage obtained exhibited a larger variance across the programs, from 15% to 72%. Sections IV-D.2, IV-D.3 and IV-D.4 discuss the large benchmarks and describe the causes for the lack of coverage gleaned from code inspection, where possible.

Figure 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 is less than 3 minutes. DataReach is most time consuming phase of our approach, 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 III). For FTPD and Haboob, DataReach used about 50% of the total running time; for other benchmarks, it used more than 90% of the total running time. We believe that an optimized implementation of DataReach will improve overall analysis performance significantly. M-DataReach is slower than DataReach in most of the benchmarks, except SpecJVM. It takes 72% more time to finish in FTPD, 43% in Haboob, 40% in Muffin and 15% in HttpClient. While it taks 14% less time to finish in SpecJVM. And M-DataReach provides more precision on three larger benchmarks.

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

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

D. Case Studies

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. While by comparing columns 8 and 9 of Table IV, we can see that input data or tests included in these benchmarks are not sufficient to drive the programs to `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 test while the *e-c links* are not experienced[9]. 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 no static analysis is able to prune.
3) Infeasible *e-c links* that may be eliminated using context-sensitive object renaming.
4) Infeasible *e-c links* that may be eliminated using context-sensitive points-to analysis.

TABLE V
NUMBER OF UNCOVERED *e-c links* IN CATEGORY 1, 2, 3 AND 4

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

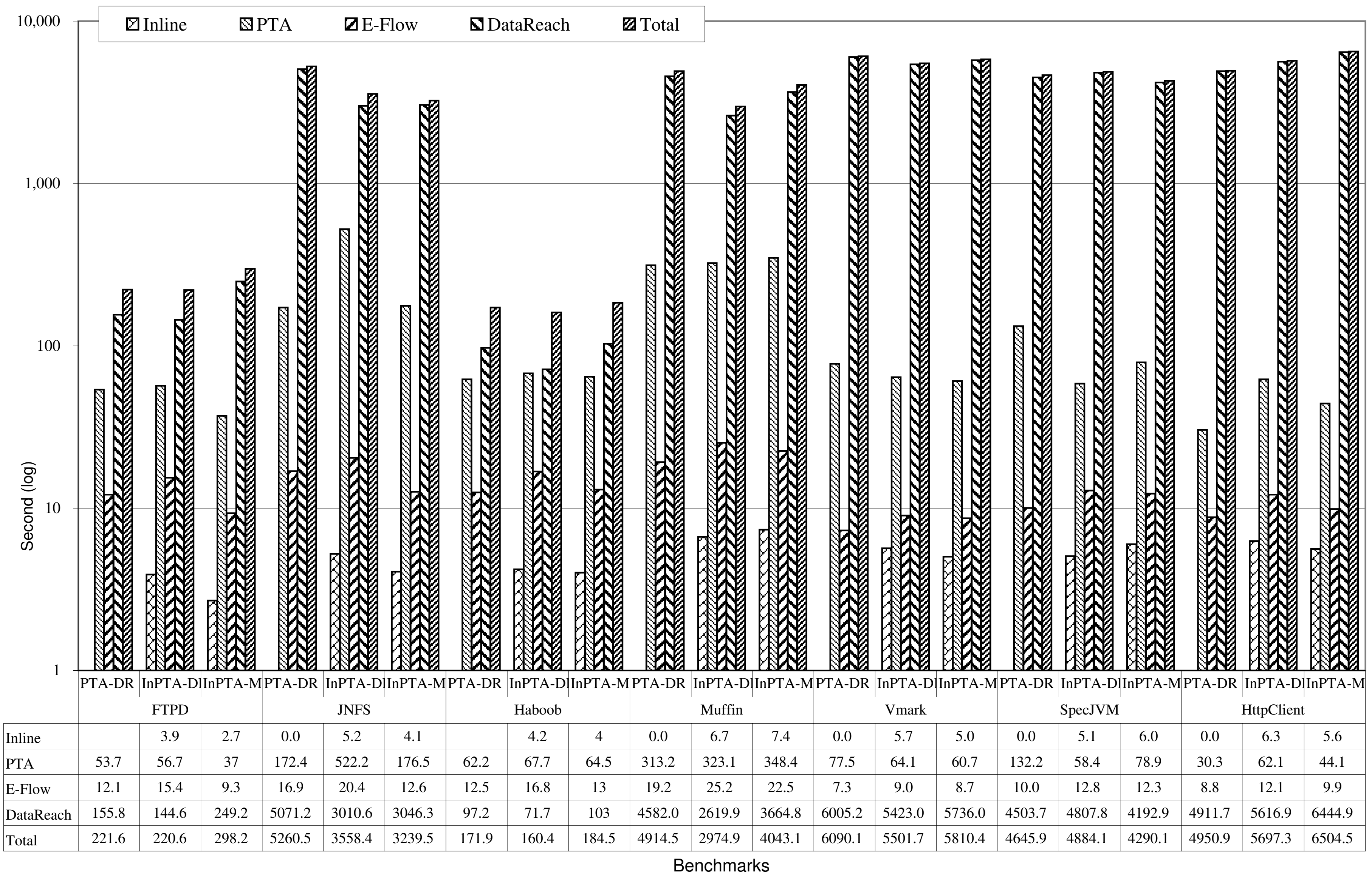
Table V shows the number of inspected *e-c links* in each of the categories for each benchmark studied, with those in parentheses showing the percentage of the number of *e-c links* in this category over the total number of inspected *e-c links* in that benchmark. The last column lists the total number of inspected *e-c links*. We will show examples extract from each benchmarks to illustrate each category in detail.

1) Muffin: There are 3 *e-c links* discovered in Muffin are in category 4. As mentioned in Section III-A, 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). The code given below is one of the possible call chains found for one of these *e-c links*.[10] There are several hundred call paths given for this single *e-c link*.

```
org.doit.muffin.Handler.processRequest()
org.doit.muffin.Https.recvReply()
org.doit.muffin.Reply.read()
org.doit.muffin.Reply.read()
java.io.SequenceInputStream.read()
```

[9]We were not successful on doing this study for VMark in detail because we don't have access to its source code .

[10]Parameters are omitted for readability.



	FTPD			JNFS			Haboob			Muffin			Vmark			SpecJVM			HttpClient		
	PTA-DR	InPTA-DI	InPTA-M	PTA-DR	InPTA-DI	InPTA-M	PTA-DR	InPTA-DI	InPTA-M	PTA-DR	InPTA-DI	InPTA-M	PTA-DR	InPTA-DI	InPTA-M	PTA-DR	InPTA-DI	InPTA-M	PTA-DR	InPTA-DI	InPTA-M
Inline		3.9	2.7	0.0	5.2	4.1		4.2	4	0.0	6.7	7.4	0.0	5.7	5.0	0.0	5.1	6.0	0.0	6.3	5.6
PTA	53.7	56.7	37	172.4	522.2	176.5	62.2	67.7	64.5	313.2	323.1	348.4	77.5	64.1	60.7	132.2	58.4	78.9	30.3	62.1	44.1
E-Flow	12.1	15.4	9.3	16.9	20.4	12.6	12.5	16.8	13	19.2	25.2	22.5	7.3	9.0	8.7	10.0	12.8	12.3	8.8	12.1	9.9
DataReach	155.8	144.6	249.2	5071.2	3010.6	3046.3	97.2	71.7	103	4582.0	2619.9	3664.8	6005.2	5423.0	5736.0	4503.7	4807.8	4192.9	4911.7	5616.9	6444.9
Total	221.6	220.6	298.2	5260.5	3558.4	3239.5	171.9	160.4	184.5	4914.5	2974.9	4043.1	6090.1	5501.7	5810.4	4645.9	4884.1	4290.1	4950.9	5697.3	6504.5

Fig. 9. Time Cost Break-down of Static Program Analysis

TABLE III

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**

TABLE IV

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%

```
java.util.zip.GZIPInputStream.read()
java.util.zip.InflaterInputStream.read()
java.util.zip.InflaterInputStream.fill()
java.io.BufferedInputStream.read()
java.io.BufferedInputStream.read1()
java.io.BufferedInputStream.fill()
java.util.jar.JarInputStream.read()
java.util.zip.ZipInputStream.read()
java.util.zip.ZipInputStream.readEnd()
java.util.zip.ZipInputStream.readFully()
java.io.PushbackInputStream.read()
java.io.FilterInputStream.read()
java.io.FileInputStream.read()
```

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

Call chains for all three *e-c links* in the second category share the same characteristics described here: they all involve the use of containers. This phenomenon is caused by context-insensitive points-to analysis, in a manner similar to the analysis imprecision for constructors discussed previously. More precise points-to analysis [41] addresses this problem by distinguishing calls by their receiver object when analyzing methods, thus producing a more sparse (and precise) points-to graph; this should reduce the call chains for a *e-c link*, or maybe even make it possible for DataReach to judge that the *e-c link* is actually infeasible. We believe that additional context sensitivity added to the points-to analysis would further improve the precision of our *e-c links*, but further experimentation is needed to confirm this hypothesis.

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

2) SpecJVM: There is no network related program in SpecJVM; therefore, we were surprised to see both disk and network I/O related *e-c links* found by our analysis. After code inspection we discovered that SpecJVM has a dedicated I/O package that is shared among all the benchmark programs. All the I/O requests are handled in this package; requests can be fulfilled by reading files either on a local disk or on a remote HTTP server. Input data is read from HTTP server when the benchmark is running as a Java applet; otherwise data is read from local disks. When the program is running as a Java applet, it is either enclosed in some web browser, or in a *Java Applet Viewer* that is provided with the Java JDK. In either case, unfortunately, we failed to set up the current implementation of the fault injection framework to perform fault injection targeted solely on the applet, without affecting the enclosing program: either the Web browser or the *Java Applet Viewer*. Thus, we could not cover the network-related *e-c links* without changing the code in the SpecJVM slightly. We discovered that `spec.harness` package maintains an `SpecBasePath` variable which is the base location of SpecJVM itself. The value of `SpecBasePath` is set to a remote URL when SpecJVM is running as a Java applet. We modified 7 lines of source code in the benchmark to keep the value of `SpecBasePath` as a URL pointing to a remote file so that I/O requests are fulfilled through network access, even when SpecJVM is running as a stand-alone Java program. This enabled the network-related *e-c links* to be covered.

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

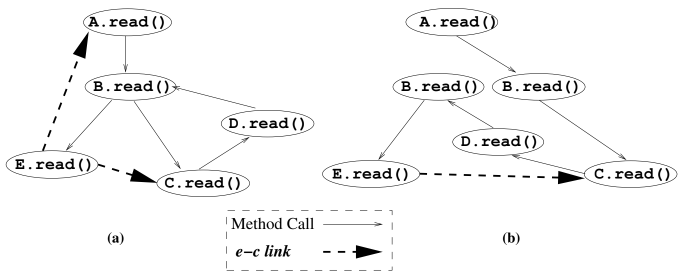


Fig. 10. Recursive Call Graph

The call chains corresponding to these 26 *e-c links*, share an 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()` that, when executed, 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 polymorphism call site. The call graph and the generated *e-c links* are shown in Figure 10 (a). The *e-c link* from `E.read()` to `A.read()` is infeasible, because the actual points-to relationship between objects in the program causes the call chain `A.read()` → `B.read()` → `E.read()` to be infeasible. Context-sensitive renaming may be able to help on this case. If we split `B.read()` according to its caller, as shown in Figure 10 (b), before performing points-to analysis and exception-flow analysis, we may be able get a better call graph and thus more precise *e-c link* information.

3) HttpClient: Control flow in methods of HttpClient is complicated. And many control flow decisions depend on values of string variables, for instance, protocol names, HTTP response code, 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 that these *e-c links* correspond to. 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. But unfortunately none of our tests fit this scenario. More carefully designed test cases and specialized HTTP responses are needed to drive the program into different control-flow paths in order to cover these 10 links.

Complicated control flow also confuses static analysis. There are 24 *e-c links* fall in category 2 (infeasible *e-c links* no static analysis is able to prune), which account for 60% of all the inspected *e-c links* in HttpClient. And below is a brief description of these *e-c links*.

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 do not necessarily involve I/O operation 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". Its hard coded in these test cases that the special HTTP connection class never uses secure protocol or any proxy server, so as to avoid real I/O operations. But it is hard (if possible) for any static analysis to recognize the infeasibility of these complicated control-flow paths and thus eliminate the corresponding *e-c links*.

Significant portion of inspected *e-c links* fall in category 2 in Muffin(43%) and SpecJVM(13%) too. All of these *e-c links* correspond to infeasible control flow paths, while the infeasibility of these paths can not be recognized by static analysis due to usage of string variables in the `if` condition expressions.

There are 6 *e-c links* of HttpClient in category 3. A simplified example is showed in Figure 11 to illustrate this case. Consider the call chain `M.getDmy` → `M.getData` → `W.read` → `Res.read`. By reading the code we can see this call chain is infeasible, because in method `M.getDmy` a `Dmy` object $o2$ is passed to the `W` object $o1$ created in method `M.getData`. So only `Dmy.read` will be called in `W.read` if we start from `M.getDmy`. But when the M-DataReach analysis examines this call chain, first $o1$ will be put into $U_{M.getData}$ when `M.getData` is processed. Then $o1$ is propagated to $U_{W.read}$. A field dereference `W.f` will be reached in method `W.read`. According to the program points-to graph, both $o2$ and $o3$ are in the points-to set of $o1.f$, thus they are both put into $U_{W.read}$. So `Res.read` is considered reachable by the analysis.

Using context-sensitive renaming before running points-to analysis may help solve this problem [41]. In the renaming phase, $o1$ will be renamed to two different objects depending the caller of its enclosing method `M.getData`: $o1_{M.getDmy}$ and $o1_{M.getRes}$. Thus only $o2$ will be in the points-to set of $o1_{M.getDmy}.f$ and only `Dmy.read` will be considered reachable given the call chain started from `M.getDmy`.

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 different parts 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*[42], 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

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

```
class M{
  void getData(A a) throws IOException{
    W w = new W(a); //o1
    w.read();
  }
  void getDmy() {
    try{
      getData(new Dmy()); //o2
    } catch (IOException e) {...}
  }
  void getRes() {
    try{
      getData(new Res()); //o3
    } catch (IOException e) {...}
  }
}
```

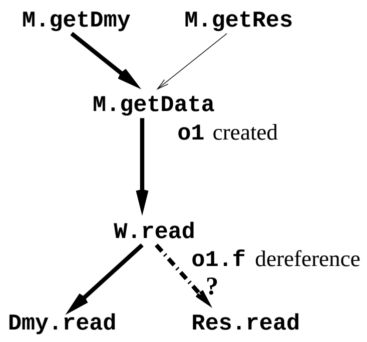


Fig. 11. New Object

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 of the *e-c links* whose `try` blocks are not reached, only three *e-c links* are related to the chat server code; the call chains corresponding to all the other *e-c links* lie 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.

V. 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.

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 [43]. 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 [13], [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 [44] 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 [45] explored an alternative approach where a fault is injected by violating a set of pre- or post-conditions in the code, which are required to be expressed explicitly in the program by the programmer. This approach used branch coverage, a program-coverage metric.

In the terminology of Hamlet's summary paper reconciling traditional program-coverage metrics and probabilistic fault analysis [46], 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 [47], which surveyed the entire software dependability process, our method can be classified as a stress-test, because it generates unlikely inputs to the program.

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 [13], [48]. 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 [10].

Sinha et. al defined an interesting and novel set of coverage metrics for testing exception constructs and gave their subsumption relations [49]. 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 [10]. 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 [13], 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 [10], 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 I, traditional fault-injection testing is performed by treating the application as a black box. Success is judged by how often the application does not crash in response to an injected fault. Other white-box, control-flow coverage metrics have been proposed by some groups for use with fault-injection testing; these correspond to previous metrics (e.g., branch, edge and basic block coverage) and have been summarized previously [10].

Analysis of exception handling. Two previous exception-flow analyses were aimed at improving exception handling in programs, for example avoiding exception handling through subsumption [15], [16]. 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 [20], [21]. 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 [16] present an interprocedural set-based [50] exception-flow analysis; only checked exceptions are analyzed. Experiments show that this is more accurate than an intraprocedural JDK-style analysis on a set of benchmarks five of which contain more than 1000 methods. Robillard et. al [15] describe a dataflow analysis that propagates both checked and unchecked exception types interprocedurally. Neither approach analyzes Java libraries unless source code is available (not the case for the JDK). They each handle a large subset of the Java language, but make the choice to omit or approximate some constructs (e.g., *static initializers*, *finally*s). Both of these analyses are more imprecise than ours, especially in their approximation of interprocedural control-flow; neither of them trace definitions of specific exception objects to their appropriate `catch` blocks[11].

Another analysis of programs containing exception handling constructs [51] calculates control dependences in the presence of implicit checked exceptions in Java. This analysis focuses on defining a new interprocedural program representation that exposes exceptional control-flow in user code. In a more recent technical report [17], Sinha et. al present an interprocedural program representation which more accurately embeds the possible intraprocedural control-flow through exception constructs (i.e., `trys`,`catchs` and `finallys`). Class hierarchy analysis is used to construct the call edges in this representation. An exception-flow analysis is defined by propagation of exception types on this representation to calculate links between explicitly thrown checked exceptions in user code and their possible handlers. It seems clear that this analysis could be extended to include implicit checked exceptions as well, assuming that the program representation could be constructed from the bytecodes of the JDK library methods, and that the fault-sensitive operations could be identified. The CHA version of our analysis seems the most similar to the analysis presented in [17]; 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 [52] designed a new intraprocedural control-flow representation, that accounted for operations that might generate unchecked exceptions called *PEIs, potentially excepting instructions*; they used this representation as a basis for safe dataflow analyses for an optimizing compiler. It is difficult to compare their representation with the others described here, because they capture different sorts of exceptions, such as *NullPointerException*, that correspond to different possibly excepting instructions.

Exceptions and compilation. Dynamic analyses have been developed to enable optimization of exception handling in programs that use exceptions to direct control-flow between methods, such as some of the Java Spec compiler benchmarks [39]). The IBM Tokyo JIT compiler [23], successfully uses a feedback-directed optimization to inline exception handling paths and eliminate `throws` in order to optimize exception-intensive programs whose performance can be improved up to 18% without affecting performance of non-intensive codes. In *LaTTe* [53], 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* [54] performs lazy exception throwing, in that it avoids creating exception objects, where possible, unless they are live on entry to their handler.

Points-to analysis. There is a wide variety of reference and points-to analyses for Java which differ in terms of cost and precision. The information computed by these analyses can be used as input to our exception-flow and data reachability analyses; clearly, the precision of the underlying analysis affects the quality of the computed coverage requirements. A detailed discussion of points-to and reference analyses and the dimensions of precision in their design spectrum appears

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

in [34]. Our partially context-sensitive points-to analysis is most closely related to the context-sensitive analyses in our previous work [26], [27]. These approaches avoid the cost of non-discriminatory context sensitivity, which seems to be impractical; they rely on techniques which preserve the practicality of the underlying context-insensitive analysis while improving precision substantially. This is achieved by effectively selecting parts of the program for which the analysis computes more precise information, either by using parameterization mechanisms as in [26], [27], 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 [55], [22], [56], [57]; these analysis algorithms implement non-discriminatorily context sensitivity.

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 [58]. 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 [58]. Souter and Pollock present a methodology (without empirical investigation) for demand-driven analysis for the detection of type infeasible call chains [59], [60]. Similarly to their work, our analysis is demand-driven as we analyze the program starting from the original call. However, our data reachability analysis propagates information in terms of objects instead of classes which will result in more precise analysis results. In addition, our work proposes a technique for summarizing the effects of callees; this problem is not addressed in [59] and [60]. 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.

VI. Conclusions

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

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

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

References

[1] J. Arlat, A. Costes, Y. Crouzet, J.-C. Laprie, and D. Powell, "Fault injection and dependability evaluation of fault-tolerant systems," *IEEE Transactions on Computers*, 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," in *Symposium on Reliable Distributed Systems*, 1999, pp. 168–177.

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

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

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

[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," in *Proc. 18th Int. Symp. on Fault-Tolerant Computing (FTCS-18)*. Tokyo, Japan: IEEE Computer Society Press, 1988, pp. 102–107.

[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 Transactions on Software Engineering*, vol. 3, no. 4, pp. 279–290, July 1977.

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

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

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

[13] S. Rapps and E. Weyuker, "Selecting software test data using data flow information," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 4, pp. 367–375, Apr. 1985.

[14] 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," Department of Computer Science, Rutgers University, Tech. Rep. DCS-TR-518, Jan. 2003.

[15] M. P. Robillard and G. C. Murphy, "Static analysis to support the evolution of exception structure in object-oriented systems," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 12, no. 2, pp. 191–221, 2003.

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

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

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

[19] M. Sable, "Soot: a java optimization framework," see `http://www.sable.mcgill.ca/soot/`.

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

[21] D. Bacon and P. Sweeney, "Fast static analysis of C++ virtual functions calls," in *Proceedings of ACM SIGPLAN Conference on Object-oriented Programing Systems, Languages and Applications (OOPSLA'96)*, Oct. 1996, pp. 324–341.

[22] D. Grove and C. Chambers, "A framework for call graph construction algorithms," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 23, no. 6, 2001.

[23] T. Ogasawara, H. Komatsu, and T. Nakatani, “A study of exception handling and its dynamic optimization in java,” in *Proceedings of ACM SIGPLAN Conference on Object-oriented Programing Systems, Languages and Applications (OOPSLA’01)*, 2001, pp. 83–95. [Online]. Available: citeseer.nj.nec.com/ogasawara01study.html

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

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

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

[27] A. Milanova, “Precise and practical flow analsis of object-oriented software,” Ph.D. dissertation, Rutgers University, 2003, also available as DCS-TR-539.

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

[29] M. Sharir and A. Pnueli, “Two approaches to interprocedural data flow analysis,” in *Program Flow Analysis: Theory and Applications*, S. Muchnick and N. Jones, Eds. Prentice Hall, 1981, pp. 189–234.

[30] O. Shivers, “Control-flow analysis of higher-order languages,” Ph.D. dissertation, Carnegie Mellon University, 1991.

[31] O. Lhoták and L. Hendren, “Scaling Java points-to analysis using Spark,” in *International Conference on Compiler Construction*, ser. LNCS 2622, 2003, pp. 153–169.

[32] C. Fu, B. G. Ryder, A. Milanova, and D. Wonnacott, “Testing of java web services for robustness,” in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, July 2004, pp. 23–33.

[33] F. Tip and J. Palsberg, “Scalable propagation-based call graph construction algorithms,” in *Proceedings of the Conference on Object-oriented Programming, Languages, Systems and Applications*, Oct. 2000, pp. 281–293.

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

[35] M. J. Radwin, “The java network file system,” see `http://www.radwin.org/michael/projects/jnfs/`.

[36] “The Muffin world wide web filtering system,” see `http://muffin.doit.org/`.

[37] M. Welsh, D. E. Culler, and E. A. Brewer, “SEDA: An architecture for well-conditioned, scalable internet services,” in *Symposium on Operating Systems Principles*, 2001, pp. 230–243. [Online]. Available: citeseer.nj.nec.com/welsh01seda.html

[38] A. S. Foundation, “Apache jarkarta project.” [Online]. Available: http://jakarta.apache.org/

[39] Specbench.org, “Spec jvm98 benchmarks.” [Online]. Available: http://www.spec.org/jvm98/

[40] V. LLC, “Volanomark.” [Online]. Available: http://www.volano.com/benchmarks.html

[41] A. Milanova, A. Rountev, and B. G. Ryder, “Parameterized object sensitivity for points-to analysis for java,” *ACM Transactions on Software Engineering Methodology*, in press, 2004.

[42] A. S. Foundation, “Apache jakarta tomcat.” [Online]. Available: http://jakarta.apache.org/tomcat/

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

[44] T. Tsai, M. Hsueh, H. Zhao, Z. Kalbarczyk, and R. Iyer, “Stress-based and path-based fault injection,” *IEEE Transactions on Computers*, vol. 48, no. 11, pp. 1183–1201, Nov. 1999.

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

[46] D. Hamlet, “Foundations of software testing: dependability theory,” in *Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of software engineering*. ACM Press, 1994, pp. 128–139. [Online]. Available: http://doi.acm.org/10.1145/193173.19540

[47] D. Tang and H. Hecht, “An approach to measuring and assessing dependability for critical software systems,” in *In Proceedings of the Eighth International Symposium on Software Reliability Engineering*, Albuquerque, NM, Nov. 1997, pp. 192–202.

[48] P. Frankl and E. Weyuker, “An applicable family of data flow testing criteria,” *IEEE Transactions on Software Engineering*, vol. 14, no. 10, pp. 1483–1498, Oct. 1988.

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

[50] N. Heintze, “Set-based analysis of ml programs,” in *Proceedings of the ACM Conference on Lisp and Functional Programmig*, 1994, pp. 306–317.

[51] S. Sinha and M. J. Harrold, “Analysis and testing of programs with exception-handling constructs,” *IEEE Transactions on Software Engineering*, vol. 26, no. 9, pp. 849–871, September 2000.

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

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

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

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

[56] R. O’Callahan, “The generalized aliasing as a basis for software tools,” Ph.D. dissertation, Carnegie Mellon University, 2000.

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

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

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

[60] ——, “Characterization and automatic identification of type infeasible call chains,” *Information and Software Technology*, vol. 44, no. 13, pp. 721–732, October 2002.