Exceptional Situations and Program Reliability

WESTLEY WEIMER and GEORGE C. NECULA
University of California, Berkeley

It is difficult to write programs that behave correctly in the presence of run-time errors. Proper behavior in the face of exceptional situations is important to the reliability of long-running programs. Existing programming language features often provide poor support for executing clean-up code and for restoring invariants.

We present a data-flow analysis for finding a certain class of exception-handling defects: those related to a failure to release resources or to clean up properly along all paths. Many real-world programs violate such resource usage rules because of incorrect exception handling. Our flow-sensitive analysis keeps track of outstanding obligations along program paths and does a precise modeling of control flow in the presence of exceptions. Using it, we have found over 1,300 exception handling defects in over 5 million lines of Java code.

Based on those defects we propose a programming language feature, the compensation stack, that keeps track of obligations at run time and ensures that they are discharged. We present a type system for compensation stacks that tracks collections of obligations. Finally, we present case studies to demonstrate that this feature is natural, efficient, and can improve reliability.

Categories and Subject Descriptors: D.2.1 [**Software Engineering**]: Requirements Specifications; D.2.4 [**Software Engineering**]: Software Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging; D.3.3 [**Programming Languages**]: Language Constructs and Features; D.4.5 [**Operating Systems**]: Reliability

General Terms: Design, Languages, Reliability, Verification

Additional Key Words and Phrases: Error handling, resource management, linear types, compensating transactions, linear sagas

This research was supported in part by the National Science Foundation grants CCR-9875171, CCR-0085949, CCR-0081588, CCR-0234689, CCR-0326577, CCR-00225610, and gifts from Microsoft Research.
The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.
Authors' addresses: W. Weimer, University of Virginia, Department of Computer Science, 151 Engineer's Way, P.O. Box 400740, Charlottesville, VA 22904-4740; email: `weimer@cs.virginia.edu`; G. C. Necula, University of California, Berkeley, 783 Soda Hall, Electrical Engineering and Computer Science Department, Berkeley, CA 94720-1776, USA; email: `necula@cs.berkeley.edu`.

ACM Reference Format:

Weimer, W. and Necula, G. C. 2008. Exeptional situations and program reliability. ACM Trans. Program. Lang. Syst. 30, 2, Article 8 (March 2008), 51 pages. DOI = 10.1145/1330017.1330019 http://doi.acm.org/10.1145/1330017.1330019

1. INTRODUCTION

While software is becoming increasingly important, much of it remains unreliable. It is easier to fix software defects if they are found before deployment. It can be difficult to use testing, the traditional approach to finding defects early, to evaluate programs in exceptional situations [Sinha and Harrold 1999; Malayeri and Aldrich 2006]. For example, testing coverage metrics may require knowledge of implicit control flow from exceptional situations, and testing error handlers may require special fault-injecting test harnesses [Candea et al. 2003]. We present an analysis for finding a class of program defects that lead to resource-handling failures in exceptional situations. We also propose a new language feature, the compensation stack, to make it easier to fix such defects.

We define an *exceptional situation* to be one in which something external to the program behaves in an uncommon but legitimate manner. For example, a file write may fail because the disk is full or the operating system is out of file handle resources. Similarly, a packet send may fail because of a network breakdown. These examples represent actions that typically succeed but may fail through no fault of the requesting program.[1]

Modern languages like Java [Gosling et al. 1996], C++ [Stroustrup 1991] and C# [Hejlsberg et al. 2003] use language-level *exceptions* to signal and handle exceptional situations. The most common semantic framework for exceptions is the *replacement model* [Goodenough 1975]. Exception handlers are typically lexically scoped and may be quite labyrinthine. Language-level exceptions introduce implicit control flow, a potential source of software defects related to reliability. Our experiments show that some program failures arise from defects in the handling of multiple cascading exceptions and in the handling of multiple resources in the presence of a single exception.

Testing a program's behavior in exceptional situations can be difficult because such situations, often called *run-time errors*, must be systematically and artificially introduced. Not all run-time errors are created equal, however. We present a *fault model* that formalizes which faulty exceptional situations we are considering and when they may occur.

A program can be tested or analyzed by simulating an environment in which faults occur as dictated by the fault model. The desired faults must still be *injected* during testing while the program is running. Some have used physical techniques (e.g., pulling a network cable while the program is running to simulate an intermittent connectivity problem) [Candea et al. 2003]. Others have used special program analyses and compiler instrumentation approaches

[1]Popping an empty stack and dividing by zero are also exceptional situations: our analysis will find defects related to using language-level exceptions for such cases, but we limit our initial discussion to external conditions. See Section 3.2 for more details.

[Fu et al. 2004, 2005] to inject faults at the software or virtual machine level. These testing-based approaches still require indicative workloads and test cases.

We present a static data-flow analysis for finding program defects. The defects are reported with respect to both the fault model and also a formal partial-correctness *specification* of proper resource handling. A defect report from the analysis includes a program path, one or more run-time errors and one or more resources governed by the specification. Such a report claims that if the run-time errors occur at the given points along the program path, the program may violate the specification for the given resources. The analysis is path-sensitive and intraprocedural. It models the flow of control, including the control-flow related to the exceptions in the fault model, precisely. It abstracts away data values and only keeps track of the resources mentioned in the specification. The analysis reported no false positives in our experiments but it can miss real defects. The analysis found over 1,300 defects in over five million lines of code.

Based on that work finding defects we propose the *compensation stack*, a language feature for ensuring that simple resources and API rules are handled correctly even in the presence of run-time errors. We draw on the concepts of compensating transactions [Korth et al. 1990], linear sagas [Alonso et al. 1994; Garcia-Molina and Salem 1987], and linear types [DeLine and Fähndrich 2001] to create a model in which obligations are recorded at run-time and are guaranteed to be executed along all paths. By enforcing a certain ordering and moving bookkeeping from compile-time to run-time, we provide more flexibility and ease-of-use than standard language approaches to adding linear types or transactions. We formalize a static semantics for compensation stacks and we present case studies to show that they can be used to improve software reliability.

Aside from providing a unified overview presentation of our previously published work [Weimer and Necula 2004, 2005] on this subject, this article includes:

—A more detailed fault model (Section 3.2).
—A full description of the data-flow analysis in the presence of typestate specifications (Section 5.3), including a discussion of its strengths and weaknesses (Section 5.6).
—Additional experiments on the importance of the defects found (Section 6.3) and effects of our filtering heuristics on false positives and false negatives (Section 6.2).
—A static semantics for compensation stacks (Section 9).
—A discussion of the tradeoffs involved in using compensation stacks to manage resources (Section 9.2).

This article deals with defects in run-time error-handling code that lead to failures under the assumptions given by a fault model. We clarify our use of the relevant terms here to prevent confusion. In this article, we use *mistake* to refer to a human action that causes a software fault or defect. A *defect* refers to a flaw in the software system or program that contributes to a failure. A *failure* refers to an observed unacceptable behavior of a system, such as violating a safety policy. *Run-time error* and *exceptional situation* are used to refer to failures or environmental conditions in the software system or its components that are signaled at the language level. A *fault model* refers to a formal consideration of which run-time errors may occur. Finally, *error-handling* code deals with a signaled run-time error in order to prevent a true failure from occurring.

The rest of this article is organized as follows. We describe the state of the art in handling exceptional situations at the language level in Section 2. In Section 3, we motivate and describe simple specifications and present our fault model for linking run-time errors and language-level exceptions. We build a control-flow graph that includes exceptional control flow in Section 4. We present a static data-flow analysis that uses the fault model and the CFG in Section 5. In Section 6, we present the results of our analysis, including experiments to measure the importance of the defects found and the false positives and false negatives associated with the analysis. We discuss finalizers and destructors in Section 7 and highlight some of their weaknesses in this context. In Section 8, we propose the compensation stack as a language feature and describe our implementation. We present a static type system in Section 9 that tracks compensation stacks but not individual resources. In Section 10, we report on case studies in which we apply compensation stacks to run-time error-handling in real programs in order to improve reliability. Section 11 describes a number of important areas of related work and we conclude in Section 12.

2. HANDLING EXCEPTIONAL SITUATIONS IN PRACTICE

An IBM survey [Cristian 1982, 1987] reported that up to two-thirds of a program may be devoted to handling exceptional situations. We performed a similar survey, examining a suite of open-source Java programs ranging in size from 4,000 to 1,600,000 lines of code (see Figure 9). We found that while exception handling is a lesser fraction of all source code than was previously reported, it is still significant.

We found that between 1% and 5% of program text in our survey was comprised of exception-handling `catch` and `finally` blocks. Between 3% and 46% of the program text was transitively reachable from `catch` and `finally` blocks, which often contain calls to cleanup methods. For example, if a `finally` block calls a `cleanUp` method, the body of the `cleanUp` method is included in this count. While it is possible to handle run-time errors without using exceptions and to use exceptions for purposes other than run-time error handling, common Java programming practice links the two together. Sinha and Harrold [2000] found that on average 8.1% of methods contained exception-handling constructs, while the JESP tool [Ryder et al. 2000] found that 16% of methods contain some kind of exception handling. Later work [Sinha et al. 2004] found patterns associated with complex implicit control flow in all of the subject programs it studied. These broad numbers suggest that handling run-time errors is an important part of modern programs and that much effort is devoted to it.

```
01: Connection cn;
02: PreparedStatement ps;
03: ResultSet rs;
04: try {
05:   cn = ConnectionFactory.getConnection(/* ... */);
06:   StringBuffer qry = ...; // do some work
07:   ps = cn.prepareStatement(qry.toString());
08:   rs = ps.executeQuery();
09:   ... // do I/O-related work with rs
10:   rs.close();
11:   ps.close();
12: } finally {
13:   try {
14:     cn.close();
15:   } catch (Exception e1) { }
16: }
```

Fig. 1. Ohioedge CRM Exception Handling Code *(with defect)*.

In general, the goal of an exception handler is program-specific and situation-specific within that program. For example, a networked program may handle a transmission exception by attempting to resend a packet, while a file-writing program may handle a storage exception by asking the user to specify an alternate destination for the data. We will not consider such high-level policy notions of correctness. Instead, we will consider more generic low-level policies related to resource handling and correct API usage.

2.1 Exception Handling Example

We begin with a motivating example showing how run-time error-handling failures can occur in practice. Consider the Java language code in Figure 1, taken from Ohioedge CRM [SourceForge.net 2003], the largest open-source customer relations management project. This program uses language features designed to handle run-time errors (i.e., it uses nested `try` blocks and `finally` clauses rather than checking and returning error codes, as one might in a C-language program), but many problems remain. `Connections`, `PreparedStatements` and `ResultSets` are resources associated with an external database. Our specification of correct behavior, which we will formalize later, requires the program to `close` each allocated resource.

In some situations, the exception handling in Figure 1 works correctly. If a run-time error occurs on line 6, the run-time system will signal an exception, and the program will `close` the open `Connection` on line 14. However, if a run-time error occurs on line 9, the resources associated with `ps` and `rs` may not be freed.

One common solution is to move the `close` calls from lines 10 and 11 into the `finally` block, as shown in Figure 2. This approach is insufficient for at least two reasons. First, the `close` method itself can raise exceptions (as indicated by the fact that it is surrounded by `try-catch` and by its type signature), so an exceptional situation while closing `rs` on line 12 might leave `ps` dangling.

```
01: Connection cn;
02: PreparedStatement ps;
03: ResultSet rs;
04: try {
05:   cn = ConnectionFactory.getConnection(/* ... */);
06:   StringBuffer qry = ...; // do some work
07:   ps = cn.prepareStatement(qry.toString());
08:   rs = ps.executeQuery();
09:   ... // do I/O-related work with rs
10: } finally {
11:   try {
12:     rs.close();
13:     ps.close();
14:     cn.close();
15:   } catch (Exception e1) { }
16: }
```

Fig. 2. Revised Ohioedge CRM Exception Handling Code *(with defect)*.

```
01: Connection cn;
02: PreparedStatement ps;
03: ResultSet rs;
04: cn = ConnectionFactory.getConnection(/* ... */);
05: try {
06:   StringBuffer qry = ...; // do some work
07:   ps = cn.prepareStatement(qry.toString());
08:   try {
09:     rs = ps.executeQuery();
10:     try {
11:       ... // do I/O-related work with rs
12:     } finally {
13:       rs.close();
14:     }
15:   } finally {
16:     ps.close();
17:   }
18: } finally {
19:   cn.close();
20: }
```

Fig. 3. Nested Try-Finally Ohioedge CRM Exception Handling Code *(correct)*.

The code in Figure 2 may also `close` an object that has never been created. If a run-time error occurs on line 6 after `cn` has been created, control will jump to line 12 and invoke `rs.close()`. Since `rs` has not yet been allocated, this will signal a "`method invocation on null object`" exception and control will jump to the catch block in line 15, with the result that `cn` is never `close`d.

Using standard language features there are two common ways to address the situation. The first, shown in Figure 3, involves using nested `try-finally` blocks. One block is required for each important resource that is dealt with simultaneously. This approach has a number of software engineering

```
Connection cn = null;
PreparedStatement ps = null;
ResultSet rs = null;
try {
  cn = ConnectionFactory.getConnection(/* ... */);
  StringBuffer qry = ...; // do some work
  ps = cn.prepareStatement(qry.toString());
  rs = ps.executeQuery();
  ... // do I/O-related work with rs
} finally {
  if (rs != null) then try { rs.close(); } catch (Exception e) { }
  if (ps != null) then try { ps.close(); } catch (Exception e) { }
  if (cn != null) then try { cn.close(); } catch (Exception e) { }
}
```

Fig. 4. Run-Time Check Ohioedge CRM Exception Handling Code *(correct)*.

disadvantages, most notably that the code becomes confusing. For example, programs in our survey (see Figure 9) commonly use three to five important resources simultaneously but programmers are rarely willing to use three to five nested `try-finally` blocks.

The second standard approach uses sentinel values or run-time checks. In Figure 4, the database objects are initialized to the sentinel value `null`. This approach has the advantage that one `try-finally` statement can handle any number of simultaneous resources. Unfortunately, such bookkeeping code often contains defects in practice (see Section 6). For example, if the guarded code contains control-flow, that control-flow must be duplicated in the `finally` clause.

2.2 Exception Handling Example Summary

The Ohioedge CRM code is typical and highlights a number of important observations. First, the programmer is aware of the safety policies: `close` is common. Second, the programmer is aware of exceptional situations: language-level exception handling (e.g., `try` and `finally`) is used prominently. Third, there are many paths where exception handling is poor and resources may not be dealt with correctly. Finally, fixing the problem typically has software engineering disadvantages: the distance between any resource acquisition and its associated release increases, and extra control flow used only for exception-handling must be included. In addition, if another procedure wishes to make use of `Connections`, it must duplicate all of this exception handling code. This duplication is frequent in practice: the Ohioedge source file containing the above example also contains two similar procedures that contain the same defects. Developers have cited this required repetition to explain why exception handling is sometimes ignored [Brown and Patterson 2003]. In general, correctly dealing with N resources requires N nested `try-finally` statements or a number of run-time checks (e.g., checking each variable against `null` or tracking progress in a counter variable). Handling such exceptional situations is complicated and error-prone in practice.

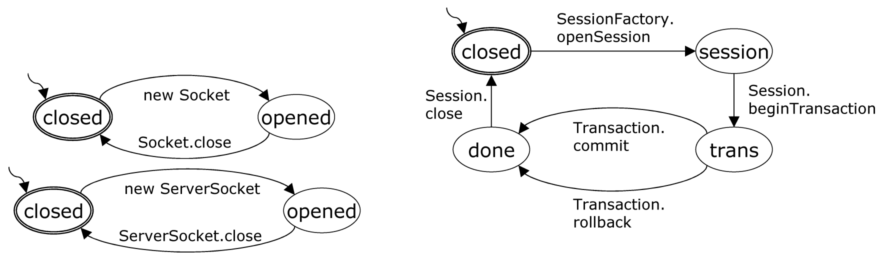


Fig. 5. A manually-derived specification for Java `Socket` resources (left) and `hibernate` `Session` resources (right).

In the next section, we will use this intuition for what often goes wrong to help formalize a partial model for program behavior.

3. SPECIFICATIONS AND EXCEPTIONAL SITUATION FAULT MODELS

A specification and a fault model are used together by our static analysis to find program defects. Intuitively, the specification describes what the program must do (e.g., free every resource it acquires) and the fault model describes what can go wrong (e.g., division by zero, network congestion).

3.1 Specifications

A specification is a formal description of correct program behavior. In this article, we will consider only *partial* specifications: those that describe a particular aspect of a program's behavior, such as how it handles certain resources. Full functional specifications are often written in specification languages that are based on formal logic [Abrial et al. 1980]. We will use lighter-weight partial specifications based on finite state machines [Ball and Rajamani 2001a; DeLine and Fähndrich 2001].

We use finite state machines (FSMs) to formalize how how programs should manipulate certain important resources and interfaces. The FSM edge labels represent program events that manipulate the internal states of resources. For example, one event may correspond to the creation of a resource and another may correspond to the disposal of a resource. We associate one FSM specification with every dynamic instance of such a resource: each resource instance is tracked separately. Resources acquired in loops are still individually governed by the specification.[2] Each FSM must end the program in an accepting state or the program is said to violate the specification for that resource instance. In addition, the program violates the specification if if the FSM ever makes an illegal transition.

Figure 5 shows a simple safety specification for Java `Socket` objects. Both `Sockets` and `ServerSockets` are based on file handles and should be freed [Campione et al. 2000]. We formally represent a specification using the

[2]The specification is separate from the means of enforcing or verifying it. It may be difficult to verify that every resource acquired in a loop is handled correctly, but it is easy to specify.

standard five-tuple finite state machine $\langle \Sigma, S, s_0, \delta, F \rangle$ [Hopcroft et al. 2000]. The first specification in Figure 5 would be given as:

$$\begin{aligned}
\Sigma &= \{\text{new Socket, Socket.close}\}\\
S &= \{\text{closed, opened}\}\\
s_0 &= \text{closed}\\
\delta &= \{\langle \text{closed, new Socket}\rangle \mapsto \text{opened},\ \ \langle \text{opened, Socket.close}\rangle \mapsto \text{closed}\}\\
F &= \{\text{closed}\}
\end{aligned}$$

Figure 5 also shows a more complicated example of a finite state machine specification [Weimer and Necula 2005] governing the use of sessions and transactions in the `hibernate` object persistence framework [Hibernate 2004]. Correct usage involves opening a session via the factory, beginning a transaction, either committing or aborting the transaction, and then closing the session. The majority of our standard library policies were simple two-state, two-edge "must call `A` after calling `B`" policies [Engler et al. 2000; Reimer et al. 2004]. Full finite state machines are used instead of such `A`–`B` pairs in order to specify more complicated resource interactions.

Our standard set of specifications [Weimer and Necula 2004] for using Java library resources includes the `socket` policy above as well as similar ones for `Streams` [O'Hanley 2005], file handles, and `JDBC` database connections.

As a concrete example, the Oracle9*i* JDBC Developer's Guide and Reference makes clear the results of violating the `JDBC` policy: "The drivers do not have finalizer methods.... If you do not explicitly close your `ResultSet` and `Statement` objects, serious memory leaks could occur. You could also run out of cursors in the database." [Perry et al. 2002] Running out of database cursors lowers transactional throughput not just for the ill-behaved client but for all other clients sharing that database. Programmers are typically very concerned with closing these objects as quickly as possible.

Users of our approach can add their own application-specific specifications by listing s_0, δ and F (since Σ and S can be inferred) or by using a subset of the syntax of an FSM specification language such as SLIC [Ball and Rajamani 2001b].

3.2 Exceptional Situation Fault Model

Our fault model applies to the operation of software and details which exceptional situations might arise. A *fault* can be officially defined as "an accidental condition that causes a functional unit to fail to perform its required function." [General Services Administration 1996] In our model a fault occurs when a method fails to adhere to a resource specification because of the implicit control flow resulting from a checked language-level exception.

Fault models are often related to the specification that is being checked. For example, a security specification related to remote buffer-overrun vulnerabilities may assume that an attacker has control over all packets that are received over the network [Wagner et al. 2000], and thus that those packet contents may take on any value. In reality, an attacker may only control some of the incoming packets, but a program that is robust in the worst-case scenario is also robust under lighter attacks. We will similarly adopt a worst-case fault model based on the assumption that many unlikely or uncommon scenarios will eventually befall a program that is running for a long time.

We wish to observe program behavior in the presence of real-world exceptional situations like network connectivity problems or database access violations. Typically, however, we have access only to the program source code and cannot mechanically simulate such exceptional situations (e.g., by running the program for a time and then pulling a plug). Thus, we need a way to bridge the gap between real world events and software-level artifacts like exception handles. Previous work by Candea et al. [2003] has found such a connection: "all faults we injected at the network level (e.g., severing the TCP connection), disk level (e.g., deleting the file), memory (e.g., limiting the JVM's heap size), and database (e.g., shutting DBMS down) resulted [in a checked exception being signaled at the language-level]."

The Java programming language features two types of exceptions: *checked* and *unchecked* [Gosling et al. 1996]. *Unchecked* exceptions typically describe defects in program logic (e.g., dereferencing a null pointer). We do not include *unchecked* exceptions in our fault model, and thus do not report defects related to them, for two reasons. The primary reason is that defect reports related to hypothetical program executions that include unchecked exceptions are more likely to be false positives. Programs typically have invariants and checks to ensure that such unchecked exceptions do not occur. The presence and correctness of such a check is statically undecidable, however, and our fault model is designed to work with lightweight intraprocedural analyses for defect detection. We view the elimination of false positives as more important than the completeness of the analysis, and thus do not consider unchecked exceptions.

The second reason we avoid unchecked exceptions is that they do not always have associated handling behavior. For example, our fault model does not consider run-time errors related to memory safety (e.g., array bounds-check violations or null-pointer dereferences), and many techniques already exist to ensure memory safety (e.g., Gay and Aiken [1998], Hauswirth and Chilimbi [2004], Necula et al. [2002b] and Tofte and Talpin [1997]). If desired, our fault model could be refined to include unchecked exceptions for which an enclosing handler exists. It would also be simple to extend the fault model with entire classes of unchecked exceptions (e.g., by treating integer division as a "method" that either returns normally or raises a divide-by-zero exception). In addition, the compensation stacks we will propose in Section 8 help to guard resources and restore invariants in the presence of both checked and unchecked exceptions. For the purposes of this presentation, however, unchecked exceptions are not part of the fault model.

Checked exceptions, on the other hand, capture our notion of exceptional situations that are beyond the program's control but that must be dealt with. Among other things, they signal failures related to network connectivity, database transactions, and physical storage, as well as security violations. The Java Type System [Gosling et al. 1996] requires that programmers either catch and handle all declared checked exceptions that they might encounter or annotate their code on a per method basis with a list of exceptions that might propagate to the caller. Checked exceptions are part of the contract associated with a Java interface.

In our fault model, any invoked method can either terminate normally or signal any of its declared checked exceptions. We expect the program to adhere to the specifications for its important resources (e.g., `Sockets`, `ResultSets`) even if a method signals, for example, a checked `SecurityException`. This fault model allows for multiple "back-to-back" faults, in which a method invocation inside a `catch` or `finally` block raises an exception.

As a corner case, our fault model forgives all defects when the programmer explicitly aborts the program. A call to `java.lang.System.exit` terminates the program and does not flag any defects even if some of the resources have not been properly handled. Programmers are typically not careful about cleaning up resources when they abort a program. We are primarily interested in finding defects in the exception handling of long-running programs and a call to `exit` almost invariably means that the programmer has given up on salvaging the situation.

Explicit `throw` statements are considered by our fault model. A program that allocates a resource, throws an exception and never cleans up the resource violates its specification under the assumptions of the fault model.

Many programs rely on proprietary third-party libraries for which neither the source code nor the byte code are readily available to outside analyses.[3] Examples include programs that link against proprietary database libraries and strict subsets of commercial programs that are made publicly available. Our analysis formally requires complete interfaces. In practice, however, we have found it useful to employ a heuristic when only partial information is available.

When we do not have the declared list of checked exceptions for a method we either reject the program or adopt a fault model such that our static analysis avoids reporting spurious warnings. We assume that an unknown method can only signal an exception for which there is a lexically enclosing `catch` clause. For example:

```
public int foo(int b) throws IOException {
  try { b = Mystery(); /* unknown method */ }
  catch (MysteryException e) { b = 0; }
  return b;
}
```

If we do not have the declaration for the `Mystery` method in the above code, we will assume that an invocation of the `Mystery` method can either terminate normally or signal a `MysteryException`. We do not assume that either it can signal an `IOException` even though the enclosing method `foo` declares that it may propagate such exceptions.

Our fault model assumes that language-level exceptions can arise from the declared checked exceptions of an invoked method and from explicit throw statements. Under our model, every call to a method (regardless of the context or

[3]Non-obfuscated bytecode is just as good as source code for our purposes.

argument values) can either raise one of its declared exceptions or terminate normally. This fault model may fail to expose real defects but will avoid making hasty conclusions about the presence of defects in the program.

4. BUILDING THE CONTROL FLOW GRAPH

Our fault-finding analysis is detailed in Section 5; this section presents our treatment of control flow in the presence of language-level exceptions. Our control-flow graph (CFG) construction is standard [Aho et al. 1986] except for our handling of method invocations (including constructor calls, etc.) and our handling of `finally`. This done based on the fault model; our goal in constructing the CFG is to admit feasible paths and behaviors under the fault model's assumptions about exceptions.

We assume a unique `start` node associated with the method entry point and a unique `end` node through which all normal and exceptional control paths must pass before exiting the method (e.g., Necula et al. [2002]): this intuitively amounts to replacing the method `body` with `try {start; body;} finally {end;}`. For an interprocedural analysis it would be necessary to model a CFG with multiple exit nodes, one per propagated exception type [Sinha and Harrold 2000].

Following the fault model, a method invocation node has an edge leading to the subsequent statement as well as zero or more edges representing possible exceptional situations. To determine these edges, we consider in turn each checked exception declared by the method. For each such exception, we inspect each lexically enclosing `catch` clause and determine if the type of the raised exception is a subtype of the caught exception. If it is, we add an edge from the method to that catch clause. If it is not, we consider the next `catch` clause. If there are no more enclosing `catch` clauses, then the exception can propagate out of the enclosing method and we add a control flow edge to the `end` node of the CFG.

`Finally` clauses are the second complication in our CFG construction. We must record how control reaches a `finally` block to determine where control flows after that block. In a `try`-`finally` statement, the `finally` clause is executed *either* if the `try` clause terminates normally *or* if the `try` clause signals an exception. If the `try` clause does not signal an exception, control flows normally after the `finally` block. If the `try` clause signals an exception, that exception is normally "re-signaled" after the `finally` clause is executed. However, if the body of the `finally` clause itself signals a new exception or executes a `return` statement (or a `continue` or `break` statement associated with a loop outside the `finally`), that new control flow overrides the "pending" exception.

We enumerate each possible path through the example code in Figure 6 to illustrate our combined handling of exceptions and `finally`. For this example we assume that `SecurityExceptions`, `IOExceptions` and `NetworkExceptions` all derive directly from a base class `Exception`. In Figure 6, the columns 2–4 record which exceptions were signaled by the methods `A`, `B` or `C`. A hyphen indicates that the method invocation was not reached (and thus could not raise an

```
public void A() throws SecurityException, IOException;
public void B() throws NetworkException;
public void C() throws SecurityException;
public void D();  // no exceptions
public void E();  // no exceptions
public void F();  // no exceptions

  try {
    try { A(); }
    catch (IOException io) { B(); }
    finally {
      C();
      // control can either transfer to the D() call two lines down
      // or an exception can be raised here ...
    }
    D();
  } catch (SecurityException sec) { E(); }
  catch (Exception e) { F(); }
```

Path Number	`A` Exception	`B` Exception	`C` Exception	Path
1	none	-	none	`A.CD..`
2	none	-	`Security`	`A.C.E.`
3	`Security`	-	none	`A.C.E.`
4	`Security`	-	`Security`	`A.C.E.`
5	`IO`	none	none	`ABCD..`
6	`IO`	none	`Security`	`ABC.E.`
7	`IO`	`Network`	none	`ABC..F`
8	`IO`	`Network`	`Security`	`ABC.E.`

Fig. 6. Example code involving exceptions and `finally`.

exception) and `none` indicates that the method terminates without raising an exception.

Of the eight control flow paths through the code in Figure 6, only the first is possible if there are no exceptions. Since neither `A` nor `C` signaled an exception, control passes from the associated finally block to the next statement: `D`. In all of the other paths the situation is more complicated. Path #2 demonstrates that a `finally` block can itself signal an exception. Path #3 shows that if a `try` clause raises an exception the `finally` clause must re-raise that exception. Path #4 illustrates a common information-masking complaint about exceptions: without additional information it is not possible to tell at `E` whether the exception was raised by `A` or `C`. In path #5, the exception signaled by `A` is caught and is thus not resignaled after `C`. Path #6 shows that a `finally` clause can signal an exception even after a `catch` clause has caught or handled one. In Path #7 the exception handler at `B` itself signals an exception which is resignaled after `C`. Since `NetworkException` is not a subtype of `SecurityException`, the `catch`-clause at `E` is not appropriate and control transfers to `F`. Finally, in path #8, everything that can go wrong does and `B`'s `NetworkException` is masked by `C`'s `SecurityException`, so control transfers to `E` instead of `F`. The `JSR` (jump to subroutine) Java bytecode instruction was designed to implement this behavior [Lindholm and Yellin 1997]. While `try-catch-finally` is conceptually simple, it has the most complicated execution description in the language

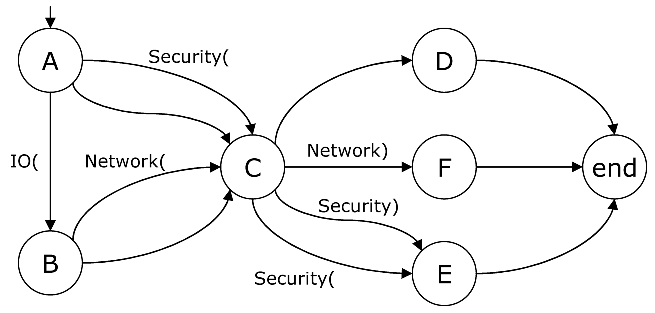


Fig. 7. A control flow graph for the example code involving exceptions and finally.

specification [Gosling et al. 1996] and requires four levels of nested "if"s in its official English description. In short, it contains a large number of corner cases that programmers often overlook.

Applying the CFG construction algorithm given above to the code in Figure 6 yields a CFG similar to the one in Figure 7. Blank edges represent normal control flow. Labeled edges represent exceptional control flow and are labeled with the associated exception. If the graph is interpreted directly it includes some infeasible paths. For example, `A-C-F-end` is possible in the graph but is not possible in the original code because it involves the `finally` block at `C` propagating a `Network` exception that was never signaled along that path. We do not want our analysis to report defects along infeasible paths.

One solution is to duplicate every `finally` block once for each exception it could propagate. This is similar to the common Java compilation technique of "inlining JSRs". We chose not to adopt that solution because we are interested in a more scalable analysis.

The solution we use involves a variant of context-free language reachability [Reps et al. 1995], as shown in Figure 7. In this framework, a path through the CFG is only valid if is described by a certain context-free language. Context-free reachability is typically used with a language of balanced parentheses to obtain a precise context-sensitive data-flow analysis by matching up method invocations and returns [Reps et al. 1995]. Here we use left parentheses to indicate "normal" or "originally-signaled" exceptions and right parentheses for exceptions that are resignaled after a `finally` block.

The language is more complicated than a simple nested "$\{^n\}^n$" of balanced parentheses because it allows both "{", representing an exception that is not re-signaled after a `finally`, and "{}}", representing an exception that is re-signaled after multiple `finally` blocks. For example, consider the code:

```
try {
  try         { throw new A(); }
  finally     { if (p == 1) throw new B(); }
} finally { p = 0; }
```

If `p==0` on entry, the corresponding string will be "$\{_A\}_A\}_A$". If `p==1` on entry, the corresponding string will be "$\{_A\{_B\}_B$". Both strings represent valid paths. Reps et al. allowed unbalanced parentheses to represent a path with a deeper or shallower call stack at the end than at the beginning; we use unbalanced parentheses to capture propagated and overridden exceptions. In our implementation, we compute our path-sensitive data-flow analysis via model-checking and state-space exploration. As a result, we effectively compute the CFL inclusion check by maintaining an explicit stack of pending exceptions to re-signal.

In addition to exceptions, `finally` clauses also interfere with `return`, `break` and `continue` statements in a similar manner. For example, if a `return` statement is executed inside the `try` block of a `try-finally` statement the return value is remembered and the `finally` block is executed. If the `finally` block terminates normally the pending `return` is "re-signaled". If the `finally` block signals an exception (or executes a `return` statement of its own, etc.) it overrides the pending `return`. We implement `return` as a special kind of *pseudo-exception* that can only be "caught" by the end of a method body. `Break` and `continue` statements are handled similarly except that more types of control flow (e.g., `while` loops) can "catch" a `break` or `continue` pseudo-exception and such pseudo-exceptions do not override normal pending exceptions if the associated loop occurs within the `finally`. This approach is similar to that of Chatterjee et al. [2001], in which data-flow elements associated with `finally` can have many different forms but the CFG construction remains normal.

The CFG construction presented here is strictly less general than the factored control-flow graph approach of Choi et al. [1999], and our data-flow analysis could be modified to work in their context. We consider a much smaller set of exception-throwing instructions: method invocations and instructions that throw checked exceptions. Their approach uses a modified dominance relation and keeps large basic blocks. Large basic blocks are useful for compiler optimizations and many analyses, but we are specifically concerned with what happens along exception control flow edges and not with improving the program. As a result, we found it more natural to retain a standard CFG structure in which an instruction dominates all of its successors in the same basic block.

5. DEFECT-FINDING DATA-FLOW ANALYSIS

The goal of our analysis is to find a path from the start of the method to the end where a resource governed by the safety policy is not in an accepting state. The analysis uses the control flow graph constructed according to the fault model as well as the formal specification. The analysis may spuriously report correct code as having defects and may fail to report real defects.

5.1 Analysis Summary and Motivation

The analysis is path-sensitive because we want to consider control flow and because the abstract state of a resource (e.g., "opened" or "closed") can change from program point to program point. We have chosen to take a fully static approach to avoid the problems of test case generation and the unavailability of third-party libraries. The analysis is intraprocedural for efficiency since we track separate execution paths. This leads to false positives, but a set of filtering heuristics (see Section 5.4) does eliminate false positives for our subject programs (see Section 6). Those heuristics may also mask real defects, however. The analysis abstracts away data values, keeping instead a set of outstanding resource states with respect to the specification as per-path data-flow facts. This abstraction can also lead to false positives and false negatives, but stylized usage patterns allow us to eliminate the false positives in practice. At join points we keep data-flow facts separate if they have distinct sets of resources.[4] We report a violation when a path leaves a method with a resource that is not in an accepting state.

5.2 Analysis Details

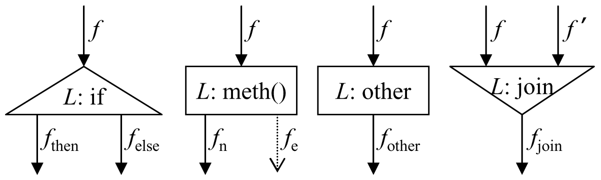
Our analysis considers each method body in turn, symbolically executing all code paths, paying special attention to control flow, exceptions and the specification.

Given the control-flow graph, our flow-sensitive data-flow analysis [Fink et al. 2006; Kildall 1973; Das et al. 2002; Engler et al. 2000; Reimer et al. 2004] finds paths along which programs violate the specification (typically by forgetting to discharge obligations) in the presence of run-time errors. We abstract away data values, and retain as symbolic data-flow facts a path through the program and a multiset of outstanding resource safety policy states for that path. That is, rather than keeping track of which variables hold important resources we merely keep track of a set of acquired resource states. We begin the analysis of each method body with an empty path and no obligations (i.e., resources governed by the specification that are not in an accepting state). If a data-flow fact at the end of method contains outstanding obligations, we term it a *violation* and report it.

The analysis is parametric with respect to a single specification $\langle \Sigma, S, s_0, \delta, F \rangle$ (see Section 3.1). If a specification contains multiple state machines we check against each one independently; it is simple to extend this algorithm to multiple simultaneous specifications. Given such a safety policy we must still determine what state information to propagate on the graph and give flow and grouping functions. Much like Fink et al. [2006], the ESP [Das et al. 2002], Metacompilation [Engler et al. 2000], and SABER [Reimer et al. 2004] projects, we combine a degree of symbolic execution with data-flow and keep state associated with multiple distinct paths that pass through the same program point.

Each path-sensitive data-flow fact f is a pair $\langle \mathcal{T}, P \rangle$. The first component $\mathcal{T}$ is a multiset of specification states. So for each $s \in \mathcal{T}$ we have $s \in S$. We use a multiset because it is possible to have multiple outstanding obligations with respect to a single type of resource. The second component P is a *path* or list of program points L between the start of the method and the current CFG edge. The path P is used to report potential violations.

[4]In the analysis presented, keeping two states will usually yield a violation later. We present the general join so that if the analysis abstraction is made more precise (e.g., if it captures correlated conditionals) the join will work unchanged.



$$
\begin{array}{lcl}
f_{\text{then}} & = & \mathsf{extend}(f, L) \\
f_{\text{else}} & = & \mathsf{extend}(f, L) \\
f_{\text{n}} & = & \begin{cases} \mathsf{extend}(f, L) & \text{if meth} \notin \Sigma \\ \mathsf{extend}(\langle \{s'\} \cup \mathcal{T}, P\rangle, L) & \text{if } f = \langle \mathcal{T} \cup \{s\}, P\rangle \text{ and } \delta(\langle s, \text{meth}\rangle) = s' \\ \mathsf{extend}(\langle \{s\} \cup \mathcal{T}, P\rangle, L) & \text{else if } f = \langle \mathcal{T}, P\rangle \text{ and } \delta(\langle s_0, \text{meth}\rangle) = s \\ \emptyset & \text{otherwise (indicates a policy violation)} \end{cases} \\
f_{\text{e}} & = & \begin{cases} \mathsf{extend}(\langle \{s'\} \cup \mathcal{T}, P\rangle, L) & \text{if } f = \langle \mathcal{T} \cup \{s\}, P\rangle \text{ and } \delta(\langle s, \text{meth}\rangle) = s' \\ & \text{and } s' \in F \\ \mathsf{extend}(f, L) & \text{otherwise} \end{cases} \\
f_{\text{other}} & = & \mathsf{extend}(f, L) \\
f_{\text{join}} & = & \begin{cases} \mathsf{extend}(\mathsf{shorter}(f, f'), L) & \text{if } f = \langle \mathcal{T}, P\rangle \text{ and } f' = \langle \mathcal{T}, P'\rangle \\ \mathsf{extend}(f, L) \cup \mathsf{extend}(f', L) & \text{otherwise} \end{cases}
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{extend}(\langle \mathcal{T}, P\rangle, L) & = & \{\langle \mathcal{T}, P \bullet L\rangle\} \\
\mathsf{shorter}(\langle \mathcal{T}, P\rangle, \langle \mathcal{T}, P'\rangle) & = & \begin{cases} \langle \mathcal{T}, P\rangle & \text{if } |P| \leq |P'| \\ \langle \mathcal{T}, P'\rangle & \text{otherwise} \end{cases}
\end{array}
$$

Fig. 8. Analysis flow functions.

5.3 Flow Functions

The analysis is defined by flow functions that are determined by the safety policy and are given in Figure 8. The four main types of control flow nodes are branches, method invocations, other statements and join points. Because our analysis is path-sensitive and does not always fully merge data-flow facts at join points, each flow function technically takes a single incoming data-flow fact and computes a set of outgoing data-flow facts. However, in all of the non-join cases the outgoing set is a singleton set. When an edge does contain a non-trivial set of data-flow facts the appropriate flow function is applied element-wise to that set.

We handle normal and conditional control flow by abstracting away data values: control can flow from an `if` to both the `then` and the `else` branch (assuming that the guard does not raise an exception) and our data-flow fact propagates directly from the incoming edge to both outgoing edges. We write $\mathsf{extend}(f, L)$ to mean the singleton set containing fact f with location L appended to its path.

A method invocation may terminate normally, represented by the f_n edge in Figure 8. If the method is not one of the important events in our safety policy (i.e., meth $\notin \Sigma$) then we propagate the symbolic state f directly. If the method is part of the policy and the incoming data-flow fact f contains a state s that could transition on that method we apply that transition and then append the label L. This is similar to the way tracked resources are handled in the Vault type system [DeLine and Fähndrich 2001].

The third possibility for a method involves creating a new important resource. For example, the first time `new Socket` occurs in a path we create a new instance of the specification state machine to track the program's use of that `Socket` object. This case and the previous case could be ambiguous if a constructor function like `new Socket` has a separate meaning somewhere else in the specification. We have never seen such a policy in practice and technically require that any outgoing edge label from the start state occur only at the start state (e.g., $\forall\langle s_0, e\rangle \in \text{Domain}(\delta).\ \forall\langle s', e'\rangle \in \text{Domain}(\delta).\ e = e' \rightarrow s' = s_0$).

The final case for a method invocation indicates a potential defect in the program. In this case, we have an important event but the analysis is not tracking any resource in a state for which that event is valid. With our simple two-state, two-event safety policies these violations almost always represent "double closes". With more complicated policies they can also represent invoking important methods at the wrong time (e.g., trying to write to a closed `File` or trying to accept on an un-bound `Socket`). When we encounter such a path, we report it and stop processing it to avoid cascading defect reports.

A method invocation may also raise a declared exception, represented by the f_e edge in Figure 8. Note that unlike the successful invocation case and as per our fault model, we do not typically update the specification state in the outgoing data-flow fact. This is because the method did not terminate successfully and thus presumably did not perform the operation to transform the resource's state. However, as a special case, we allow an attempt to "discharge an obligation" or move a resource into an accepting state to succeed even if the method invocation fails. Thus, we do not require that programs loop around `close` functions and invoke them until they succeed. Since no programs we have observed do so, it would create spurious defect reports. The check $s' \in F$ requires that the result of applying this method would put the object in an accepting state.

The join function tracks separate paths through the same program point provided that they have distinct multisets of specification states. Our join function uses the *property simulation* approach [Das et al. 2002] to grouping sets of symbolic states. We merge facts with identical obligations by retaining only the shorter path for defect reporting purposes (modeled here with the function $\text{shorter}(s_1, s_2)$). We may visit the same program point multiple times to analyze paths with different sets $\mathcal{T}$.

To ensure termination, we stop the analysis and report a defect when a program point occurs twice in a single path with different obligation sets (e.g., if a program acquires obligations inside a loop). For the safety policies we considered, that never occurred. We did encounter multiple programs that allocated and freed resources inside loops, but the (lack of) run-time error handling was always such that an exception would escape the enclosing loop.

For each $f = \langle\mathcal{T}, P\rangle$ that goes in to the end node of the CFG, if $\exists s \in \mathcal{T}.\ s \notin F$ the analysis reports a candidate violation along path P. In addition, it is possible to report violations earlier in the process (e.g., double closes).

Program	Lines of Code	methods with defects: Library Policy	methods with defects: Mined Policy	paths with defects per library safety policy: Database	paths with defects per library safety policy: `File`	paths with defects per library safety policy: `Stream`
`javad 2000`	4k	**1**		0	0	1
`javacc 3.0`	13k	**4**		0	36	0
`jtar 1.21`	17k	**5**		0	7	4
`jatlite 3.5.97`	18k	**6**		0	4	0
`toba 1.1c`	19k	**6**		0	1	20
`osage 1.0p10`	20k	**3**		15	0	0
`jcc 0.02`	26k	**0**		0	0	0
`quartz 1.0.6`	27k	**17**		46	5	20
`infinity 1.28`	28k	**14**	**4**	0	165	1
`ejbca 2.0b2`	33k	**31**		0	39	117
`ohioedge 1.3.1`	40k	**15**		23	5	0
`jogg 1.1.3`	47k	**7**		0	11	2
`staf 2.4.5`	55k	**12**		0	76	0
`hibernate 2.0b4`	57k	**13**	**93**	34	6	19
`jaxme 1.54`	58k	**6**		1	12	0
`axion 1.0m2`	65k	**15**	**45**	1	61	5
`hsqldb 1.7.1`	71k	**18**	**35**	22	8	13
`cayenne 1.0b4`	86k	**7**	**18**	2	27	6
`sablecc 2.17.4`	99k	**3**	**0**	0	0	6
`jboss 3.0.6`	107k	**40**	**94**	134	5	53
`mckoi-sql 1.0.2`	118k	**37**	**69**	37	6	190
`portal 1.8.0`	162k	**39**		99	20	13
`pcgen 4.3.5`	178k	**17**		0	120	0
`compiere 2.4.4`	230k	**322**		715	10	9
`aspectj 1.1`	319k	**27**		0	50	48
`ptolemy2 3.0.2`	362k	**27**	**72**	0	504	46
`eclipse 2/26/07`	3.0M	**203**		0	453	486
total	5.3M	**895**	**430**	1129	1631	1059
`eclipse 5/25/03`	1.6M	**126**		0	181	252

Fig. 9. Run-time error-handling defects by program and policy. The "Defects" columns indicate the total number of distinct *methods* that contain violations of various policies. The "Database", "`File`", and "`Stream`" columns give the total number of acyclic control-flow *paths* within those methods that violate the given policy. Results from an older version of `eclipse` are provided for comparison purposes but are not included in the "total" row.

5.4 Defect Report Filtering

Finally, we use heuristics as a post-processing step to filter candidate violations. The analysis as presented finds intraprocedural violations of the policy with respect to the fault model, but it may also point out spurious warnings. A spurious defect report is called a *false positive*. Based on a random sample of two of our benchmarks, 30% of the defect reports produced by our analysis as presented before are false positives. We believe that number to be unacceptably high. Based on an exhaustive analysis of the false positives reported by this analysis, we designed three simple filtering rules, which eliminate all false positives in our twenty-seven benchmark programs (see Figure 9) and two case studies (see Section 10).

When a violation $\langle \mathcal{T}, P \rangle$ is reported, we examine its path P.

(1) Conditional. Every time the path passes through a conditional of the form `t = null` we look for a state $s \in \mathcal{T}$ where $s \notin F$ and s represents an object of type `t`. If we find such a state we remove it from $\mathcal{T}$. This addresses the very common case of checking for `null` resources:

```
if (sock != null) try { sock.close(); } catch (Exception e) {}
```

We assume that a path in which `t` was verified to be `null` does not make any further use of it. Since our analysis ignores data values, it would report a false positive in such cases.

(2) Field. We examine L for assignments of the form `field = t`. For each such assignment we remove one non-accepting state of type `t` from $\mathcal{T}$. When important resources are assigned to object fields, that object sometimes contains a separate "cleanup" method that is charged with releasing those resources. We assume that such a cleanup method always exists and is called later; we may thus miss real defects.[5] Such cleanup methods are common in our experience. For example, the `SessionImpl` class of the `hibernate` program (see Section 6) features a method called `cleanup()` that sets one Boolean field to false and then invokes the `clear()` method of the objects associated with nine other fields.

(3) Return. If L contains a `return t`, we remove one non-accepting state of type `t` from $\mathcal{T}$. Methods with such `return` statements are effectively wrappers around the standard library constructors and the obligation for handling the resource falls to the caller. While we did observe many such constructor wrappers we did not observe any "destructor" wrappers, so we do not similarly remove obligations based on values passed as function arguments. If our analysis were interprocedural we would not need this filtering rule.

If the set $\mathcal{T}$ has been depleted so as to contain only states $s \in F$, there is no candidate violation and nothing is reported. These three simple filters eliminate *all* false positives we encountered but could cause this analysis to miss real defects. Based on a random sample of two of our benchmarks as well as an in-depth analysis of false positives on a version of `eclipse`, applying these three filters causes our analysis to miss between 5% and 10% of the real defects (see Section 6.2).

5.5 Usability

We have implemented our analysis in a tool that reports potential error-handling defects in Java programs with respect to our fault model and various safety policies. The tool comes with a predefined set of standard library policies. Users can also define additional policies by giving the policy a unique name and listing the resources and transitions involved. For simple two-state safety policies, this reduces to listing the acquire and release functions; larger

[5]We could refine this heuristic by searching for a "cleanup" method that calls the appropriate `field.close()`, but we would still need to prove that the cleanup method is eventually called later, possibly in another method, which is beyond the scope of our lightweight intraprocedural analysis.

policies are defined as a table of transitions. For example, the `Socket` portion of the policy in Figure 5 is represented as a small text file:

```
Socket                         // policy name
simple_policy                  // is a simple two-state policy
java/net/Socket.java Socket    // acquire function
java/net/Socket.java close     // release function
```

More verbose syntax is available for specifying compilation units declaring multiple classes or for concepts such as "or any subclass of this class." The tool is invoked on a set of policy names and Java files.

The analysis is exponential in the worst case but quite efficient in practice. For example, performing this analysis, including parsing, typechecking and printing out the resulting defect paths, took 104 seconds and 46 MB of memory on a 1.6 GHz machine for the `hibernate` program. That program has 7,038 methods, totaling 57,580 lines of code, with an average method size of 8.18 lines and a largest method size of 350 lines. As a second example, `eclipse 3.2.2` involves 23,801 methods and three million lines of code. The average method size is 13.34 lines and the largest method is 1,649 lines; the total analysis time including parsing was 350.8 seconds on a 3.6GHz Pentium4.

Each defect report is given its own context listing its compilation unit and the relevant resources (policies) and exceptions involved. The path P is given using source-code line and column offsets; calls to methods, thrown exceptions and filtering heuristic information are included in the path list. Additional information, such as a ranking of exceptions by the number of times they occur in defect reports and the number of times they are handled successfully, is also given.

5.6 Analysis Weaknesses

The analysis as presented includes a number of sources of false positives and false negatives. First, even before the analysis itself is used, our fault model (see Section 3.2) does not consider implicit control flow paths from unchecked exceptions (such as `DivisionByZero`). The analysis may thus miss defects along such paths. Second, the filtering heuristics can introduce false negatives by masking legitimate defect reports. Third, the intraprocedural nature of the analysis can lead to false positives, as the analysis will flag both the caller and the callee procedure if a resource is passed between them.[6] Finally, abstracting data values can create both false positives and false negatives. For example, the analysis will not report a defect associated with creating two resources and then freeing the second one twice.

A number of simple improvements could be made to the analysis without reducing its efficiency. For example, the analysis might be combined with an alias analysis so that data values are not abstracted away completely but instead dealt with by equivalence classes of aliases [Das et al. 2002; Fink et al.

[6]Our intraprocedural analysis can also have false negatives if a called method throws an exception that is not declared in its interface, but such situations are not part of our fault model (which includes only declared checked exceptions).

Program	Application Policy Acquire and Release Events
infinity	infinity/gui/WindowBlocker setBlocked false infinity/gui/WindowBlocker setBlocked true
hibernate	hibernate/SessionFactory openSession hibernate/Session close
axion	axiondb/tools/BatchSqlCommandRunner BatchSqlCommandRunner axiondb/tools/BatchSqlCommandRunner close
hsqldb	hsqldb/lib/StringInputStream StringInputStream hsqldb/lib/StringInputStream close
cayenne	cayenne/conn/PoolManager PoolManager cayenne/conn/PoolManager dispose
jboss	jboss/ejb/BeanLock sync jboss/ejb/BeanLock releaseSync
mckoi-sql	mckoi/database/SimpleTableQuery SimpleTableQuery mckoi/database/SimpleTableQuery dispose
ptolemy2	ptolemy/kernel/util/Workspace getWriteAccess ptolemy/kernel/util/Workspace doneWriting

Fig. 10. Example simple two-state application-specific policies.

2006]. As a second example, for method bodies without loops, the analysis might take a symbolic execution-style approach and associate predicates with data-flow facts to avoid false positives associated with correlated conditionals. Such an improvement might obviate the need for the Conditional filtering heuristic.

Despite these weaknesses, the analysis as presented is certainly applicable to general Java programs that use standard library and program-specific resources in the presence of language-level exception handling, especially since most such programs treat resources in a highly idiomatic manner. We discuss the analysis results in the next section.

6. POOR HANDLING ABOUNDS

In this section we apply the analysis from Section 5 and the specifications from Section 3.1 to show that many programs have defects in their handling of exceptional situations. We consider a diverse body of twenty-seven Java programs totaling over five million lines of code. Most of the programs were taken from the Sourceforge open source program repository [SourceForge.net 2003]. The programs include databases, business software, networking applications and software development tools.

Figure 9 shows results from this analysis. The “Defects” column shows the number of methods that violate at least one policy. We applied our three library policies to all of the programs.[7] The largest standard library policy used in our experiments has 11 edges and 8 states and describes the JDBC database interface. In addition, for nine of the programs, totaling 1 million lines of code, a total of 69 program-specific policies (found via specification mining [Weimer and Necula 2005]) were also available. Figure 10 shows indicative examples of some two-state application-specific policies used.

[7]We also applied the `Socket` policy from Figure 5 and found 14 paths with violations in 4 of the programs. Since the number of `Socket` violations is low when compared to the other policies we will not discuss them directly.

In the larger programs, much of the application logic did not interact with our library policies. For example, in `eclipse` and `ptolemy2` only 10% of the source files mentioned resources covered by those safety policies. We find many more defects in `ptolemy2` when using `ptolemy2`-specific specifications, and thus more frequently occurring, mined policies.

Figure 9 includes every defect report that was not filtered out using the heuristics from Section 5.4. All of the methods with defects were manually inspected to verify that they contained at least one defect. This inspection assumed that a method could raise any of its declared exceptions (i.e., it used the same fault model discussed in Section 3.2). The heuristics eliminate all false positives that the analysis would report on these programs. Thus, from the perspective of our fault model, there are zero false positives in Figure 9. The heuristics cause some false negatives; we present detailed results about them in Section 6.2.

All paths in Figure 9 arose in the presence of exceptions the program did not handle correctly. More than half of these paths featured some sort of exception handling (i.e., the exception was caught), but the resource was still leaked. This demonstrates that existing exception handlers often contain defects.

The most common problematic exception was the Java `IOException`: it occurred somewhere in 597 of the defects paths and was the final, uncaught exception in 474 of them. The `SQLException` was a close second, occurring in 877 paths and going uncaught in 114 of them. Although the `SQLException` occurred on more paths, we rank by counting unique methods with errors. There were more individual methods with bad handling of `IOException`, while a smaller number of methods that involved `SQLException` tended to feature many hidden control-flow paths involving that exception. While `IOExceptions` are more problematic in absolute terms, it appears to be more difficult to handle `SQLExceptions` correctly. The `SecurityException` was third with 86 mentions and 68 uncaught instances. These numbers show that programs have some sort of error handling (e.g., `SQLExceptions` are caught) but that the handling code itself is not always correct.

A single path may violate multiple safety policies: for example, along an exceptional path the program might forget to close a `Socket` and a `ResultSet`. Since the leftmost policy in Figure 9 is presumed to be more important, such cases are categorized in favor of it. To give one example, of the 59 possible defect paths reported in `hibernate`, 34 involved violating multiple policies along a single path with up to 4 forgotten resources at once. All of the defects that we report could have been fixed with flags or nested `try-finally` statements. However, fixing those 32 paths from `hibernate` would require four nested `try-finally` statements or multiple separate flags. Defects that cross safety policies argue strongly for the need to have an error-handling mechanism that supports multiple resources in sequence.

6.1 Sample Defects

We briefly illustrate two indicative defects found by our analysis. In many cases, language-level exception handling is omitted, as in this example from `axion`'s

ObjectBTree class:

```
01: public void read() throws IOException, /* ... */ {
02:   File idxFile = getFileById(getFileId());
03:   // ...
04:   FileInputStream fin = new FileInputStream(idxFile);
05:   ObjectInputStream in = new ObjectInputStream(fin);
06:   // ...
07:   in.close();
08:   fin.close();
09: }
```

This happens even though the throws annotation on line 1 and extant handling in other methods mean that the programmer is aware of the possibility of run-time errors. Such examples show that it would be useful to have an automatic mechanism that does the right thing in common cases with no programmer intervention.

We also encountered instances of try-finally statements that protect some, but not all, operations, as in this example from eclipse 3.2.2's JNIGeneratorApp class (all comments added for emphasis):

```
01: void output(byte[] bytes, String fileName) throws IOException {
02:   FileInputStream is = null;
03:   try {
04:     is = new FileInputStream(fileName);
05:     if (compare(new ByteArrayInputStream(bytes),
06:                 new BufferedInputStream(is))) return;
07:   } catch (FileNotFoundException e) { }     // ignored exception
08:   finally {
09:     try { if (is != null) is.close(); }
10:     catch (IOException e) {}                // ignored exception
11:   }
12:   FileOutputStream out = new FileOutputStream(fileName);
13:   out.write(bytes);                         // no try
14:   out.close();                              // no finally
15: }
```

Care is taken to deal with run-time errors that occur on lines 3–11 when is is created and used the first time, but writing to out on line 13 is done without an enclosing try-finally. Successfully opening fileName for reading on line 4 does not guarantee that opening it for writing on line 12 and writing to it on line 13 will succeed. Such examples show that it would be useful to have a mechanism for fine-grained control of some error handling but automatic behavior for others. Additional examples of detected errors can be found in Weimer and Necula [2004].

Condition	Field	Return	False Positives	False Positive %	Defects Found	Defects Found %
Yes	Yes	Yes	0	0%	203	94.4%
Yes	Yes	–	39	15.8%	207	96.3%
Yes	–	Yes	12	5.5%	204	94.9%
–	Yes	Yes	58	21.7%	209	97.2%
Yes	–	–	51	19.7%	208	96.7%
–	Yes	–	102	32.3%	214	99.5%
–	–	Yes	71	25.3%	210	97.7%
–	–	–	115	34.9%	215	100%

Fig. 11. Number of false positives reported and defects found on `eclipse 3.2.2` as a function of filtering heuristics used.

6.2 Filtering Experiments

The defects that we are prevented from seeing can be as interesting as those that are reported. We conducted a series of experiments to determine how many false positives were eliminated by our filtering heuristics and how many false negatives were created by those same heuristics. Figure 11 shows the result of turning off various combinations of our filtering heuristics when applying our analysis to `eclipse 3.2.2 2/26/07`. Thus the Return heuristic eliminates at least 39 false positives (or 34% of the total potential false positives), the Field heuristic eliminates at least 12 false positives (or 10%), and the Condition heuristic eliminates at least 58 (or 50%).

Note that a candidate defect report can contain properties related to multiple heuristics and can thus be filtered by either one. For example, there are 12 false positives without Field and 58 false positives without Condition, but if both are eliminated then there are 71 (not 70) false positives total. There is thus one potential defect report that involves both storing the object to a field and also checking the object against `null`. The total number of false positives that could be filtered by more than one rule is small (6 out of 115, or 5%).

Figure 11 shows potential incremental benefits in terms of finding more defects. In this experiment the false negative rate for using the three heuristics was 5.5%. If the user is willing to accept a 35% false positive rate, the number of real defect reports dropped by the filtering rules drops to zero. Disabling the Condition seems to be a reasonable point of compromise: the false negative rate drops below 3% and the false positive rate is just above 20%. It is our opinion that software ships with known defects [Liblit et al. 2003] and that making the tool and analysis easy to use by having no false positives is more important than finding 5% to 10% more defects. In a verification or safety-critical setting, however, removing sources of false negatives would be of paramount importance.

6.3 The Importance of Detected Defects

Even if a defect is not a false positive and represents an actual violation of the policy, it may not be worth the development organization's time to fix the defect. Defects that are perceived as unlikely to affect real users often go unfixed at many points in the development cycle because of the perceived dangers of code churn and because there are enough "dangerous" defects to fix to keep programmers occupied.

We report resource leaks along paths that contain one or more run-time errors. We must thus demonstrate that these defects are a serious problem "in the real world". Unfortunately, a thorough evaluation of the importance of a defect and the frequency of its occurrence at runtime is beyond the scope of this work and is typically situation-specific. Aspects such as the performance or security impact of a defect or the cost of fixing it can be difficult to measure quantitatively. We present some evidence to suggest that the defects we report are important.

One of the authors of `ptolemy2` ranked the defects that we found on his own five point scale. For that program, 11% of the reported defects were in tutorials or third-party, and thus unimportant, code; 44% of them rated a 3 out of 5 for taking place in "little used, experimental code"; 19% of them rated a 4 out of 5 and were "definitely a bug in code that is used more often"; and 26% of them rated a 5 out of 5 and were "definitely a bug in code that is used often." The 45% of the defects that rated a 4 or 5 were fixed immediately. The granularity for "code that is used often" was module-level (e.g., "engine" vs. "testing"), and thus even a defect in an important module might lie on an uncommon path. The author claimed that for his long-running servers resource leaks were a problem that forced them to reboot every day as a last-ditch effort to reclaim resources. We cannot claim that this breakdown generalizes, but it does provide one concrete example.

We also performed a so-called *time travel* experiment to determine whether the defects found by our analysis were important enough to fix. The direct experiment of finding defects, reporting them to developers and then counting how many are fixed is difficult to perform, especially in the open-source community.

We used archival copies and version control systems to obtain a snapshot of `eclipse 2.0.0` from July 2002 as well as a snapshot of `eclipse 3.0.1` from September 2004. We then ran our analysis on `eclipse 2.0.0` and chose at random 100 of defects reported. Without reporting any of the defects to `eclipse` programmers we then looked for those defects in `eclipse 3.0.1` to see if they had been fixed by the natural course of `eclipse` development. In our case, 43% of the defects found by our tool in `eclipse 2.0.0` had been fixed by `eclipse 3.0.1`. Between those versions `eclipse` underwent many refactorings, so manual inspection was necessary because defective code had often moved from one class to another. Given our stated goal of improving software quality by finding and fixing defects before a product is released, this number is important and helps to validate our analysis, our fault model and our specifications. Combined with our zero false positive rate it suggests that using our analysis is worthwhile because almost half of the defects it reports would have to be fixed later anyway.

It is difficult to obtain numbers indicating what fraction of the defects reported were later fixed for various defect-finding research projects. Our two experiments suggest that 44–45% of the defects we report are considered real by developers. As one external datapoint, the FindBugs project [Hovemeyer and Pugh 2004] produced 300 warnings when applied to a 350,000-lines-of-code Java financial application and the development team considered 17% of them to be real defects.

7. DESTRUCTORS AND FINALIZERS

Before proposing a new language feature to simplify the programming of resource reclamation in the presence of exceptions, we must consider the advantages and disadvantages of existing approaches. Based on the defects found by our analysis, we claim that `try-finally` blocks are ill-suited for handling certain classes of resources in the presence of run-time errors. A more detailed characterization of the defects found by our analysis can be found in previous work [Weimer and Necula 2004]. In essence, however, exceptions create hidden control-flow paths that are difficult for programmers to reason about [Sinha and Harrold 2000; Sinha et al. 2004; Gupta et al. 2000; Choi et al. 1999; Robillard and Murphy 2003].

Destructors and finalizers are other existing programming language features that can help programs deal with resources in the presence of run-time errors.

A *destructor* is a special method associated with a class. Destructors are typically used with the language C++ [Stroustrup 1991] but are also present in other languages like C# [Hejlsberg et al. 2003]. When a stack-allocated instance of that class goes out of scope, either because of normal control flow or because an exception was raised, the destructor is invoked automatically. Destructors are tied to the dynamic call stack of a program in the same way that local variables are. Destructors thus provide guaranteed cleanup actions for stack-allocated objects even in the presence of exceptions. However, for heap-allocated objects the programmer must still remember to explicitly delete the object along all paths. We would like to extend destructors: rather than one implicit stack tied to the call stack, programmers should be allowed to manipulate first-class collections of obligations.

In addition, we believe that programmers should have guarantees about managing objects and actions that do not have their lifetimes bound to the call stack (such objects are common in practice—see, for example, Gay and Aiken [1998]). In many domains, multiple stacks are a more natural fit with the application. For example, a web server might store one such stack for each concurrent request. If the normal request encounters an exceptional situation and must abort and release its resources, there is generally no reason that another request cannot continue. Destructors can be invoked early, but would typically have to include a flag to ensure that actions are not duplicated when it is called again. We believe such bookkeeping should be automatic. Destructors are tied to objects and there are many cases where a program would want to change the state of the object, rather than destroying it. We shall return to that consideration in Section 8.2.

A *finalizer* is another special method associated with a class. Finalizers are typically used with Java [Gosling et al. 1996] but are also present in other languages like C# [Hejlsberg et al. 2003]. A finalizer is invoked on an instance of a class when that instance is about to be reclaimed by the garbage collector.

The garbage collector is not guaranteed to find any particular piece of garbage and is not guaranteed to find garbage in a certain order or time-frame. Compared to pure finalizers, most programmer-specified error handling must be more immediate and more deterministic. Finalizers are arguably well-suited to resources like file descriptors that must be collected but need not be collected right away. However, even that apparently-innocuous use of finalizers is often discouraged because programs have a limited number of file descriptors and can easily "race" with the garbage collector to exhaust them [O'Hanley 2005]. In contrast, the elements of the "Database" policy from Section 3.1 should be released as quickly as possible, making finalizers an awkward fit for performance reasons. For example, the Oracle9*i* documentation specifically states that finalizers are not used and that cleanup must be done explicitly. As a second example, the SABER project [Reimer et al. 2004] classified as a defect the use of finalizers to close database connections in a large financial application in the field.

We want a mechanism that is well-suited to being invoked early, and while finalizers can be called in advance they suffer from the same disadvantages as destructors in that regard. Like destructors, finalizers can be invoked early but doing so typically requires additional bookkeeping.

More importantly, finalizers in Java come with no order guarantees [Gosling et al. 1996]. For example, a `Stream` built on (and referencing) a `Socket` might be finalized after that `Socket` if they are both found unreachable in the same garbage collection pass. For example, in this code if the `Stream` finalize method is called first, the `Socket` will be `closed` twice:

```
class Socket {
  /* ... */
  void finalize() { this.close(); }
}
class Stream {
  Socket s ;
  /* ... */
  void finalize() { s.close(); this.close(); }
}
```

If the arbitrary cleanup actions above were to be handled by finalizers on dependent objects, the natural "trick" of adding an extra pointer field to the `child` object pointing to the `parent` object to ensure that the `child` action is called before the `parent` action would not be sound. Thus we desire an error handling mechanism that can strictly enforce such dependencies and provide a more intuitive ordering for cleanup actions. In addition, finalizers must be asynchronous (and may be so even in single-threaded programs), which complicates how they must be written. While such dependencies could be encoded in a finalizer system, we did not observe such a system in any of the programs we examined in Section 6.

Finally, note that Java programmers do not make even a sparing use of finalizers to address these problems. Some Java implementations do not implement finalizers correctly [Boehm 2003], finalizers are often viewed as unpredictable or dangerous, and the delay between finishing with the resource and having the finalizer called may be too great. In all of the code surveyed in Section 6, there were only 13 user-defined finalizers (`hibernate` had 4; `osage` had 3; `jboss` and `eclipse` had 2 each; `javad` and `aspectj` had 1 each). One might also hope that standard libraries would make use of finalizers, but this is not always the case. The GNU Classpath 0.05 version of the Java Standard Library does not use finalizers for any of the resources governed by the safety policies in Section 6. Sun's JDK 1.3.1_07 does use them, but only in some situations (e.g., for database connections but not for sockets). While other or newer libraries may well use finalizers for all such important resources, one cannot currently count on the library to do so in a platform-independent manner. We would like to make something like finalizers more useful to Java programmers by making them easier to use and giving them destructor-like properties such as support for scoping based on the dynamic call stack.

The results in Section 6 argue that language support is necessary: a better `Socket` library will not help if `Sockets`, databases, and user-defined resources must be dealt with together. Using exception handling to deal with important resources is difficult. In the next section, we will describe a language mechanism that makes it easy to do the right thing: all of the defects presented in this paper could have been avoided using our proposed language extension. The analysis presented in Section 5 could verify programs that use our mechanism in an intraprocedural manner (e.g., using the `methodScopedStack` approach described in Section 8).

8. COMPENSATION STACKS

Based on existing defects and coding practices, we propose a language extension that lets program actions and interfaces be annotated with *compensations*, which are closures containing arbitrary code. At run-time, these compensations are stored in first-class stacks. *Compensation stacks* can be thought of as generalized destructors, but we emphasize that they can be used to execute arbitrary code and not just call functions upon object destruction.

8.1 Compensations

Our compensation stacks are an adaptation of the database notions of *compensating transactions* and *linear sagas* [Garcia-Molina and Salem 1987]. A compensating transaction semantically undoes the effect of another transaction after that transaction has committed. A saga is a long-lived transaction seen as a sequence of atomic actions $a_1 \cdots a_n$ with compensating transactions $c_1 \cdots c_n$. This system guarantees that either $a_1 \cdots a_n$ executes or $a_1 \cdots a_k c_k \cdots c_1$ executes. Note that the compensations are applied in reverse order. We have found the compensation stack model to be a good fit for saga-style run-time error handling, although there are other techniques for controlling long-running multi-step processes (e.g., Dayal et al. [1990]). Many program actions require that multiple resources be handled in sequence.

Our system links actions with compensations, and guarantees that if an action is taken, the program cannot terminate without executing the associated compensation. Compensation stacks are first-class objects that store closures. They may be passed to methods or stored in object fields. The Java language syntax is extended to allow arbitrary closures to be pushed onto compensation stacks. These closures are later executed in a last-in, first-out order. Closures may be run "early" by the programmer, but they are usually run automatically when a stack-allocated compensation stack goes out of scope or when a heap-allocated compensation stack is finalized. Relying on a finalizer to deal with a compensation stack opens the door to many of the timing problems associated with finalizers (see Section 7). However, since the finalizer is called on the compensation *stack* and not on an individual compensation, the compensations within a single stack will still be called in last-in-first-out order. Multiple compensation stacks that are garbage collected at the same time may be finalized in any order, but within any stack the compensations will be run in order. Programmers should not use finalizers for nested compensation stacks that are being used to achieve a "nested transaction" effect.

If a compensating action raises an exception while executing, the exception is logged but compensation execution continues.[8] When a compensation terminates (either normally or exceptionally), it is removed from the compensation stack.

Compensation stacks behave like destructors, deallocating resources based on lexical scoping, but they are also first-class collections that can be put in the heap and that make use of finalizers to ensure that their contents are eventually executed. They are as convenient as destructors when lexical and lifetime scoping coincide and are flexible enough to handle resources when they do not. Executing some compensations early is important and allows the common programming idiom where critical shared resources are freed as early as possible along each path.

In addition, the program can explicitly discharge an obligation without executing the corresponding compensation code (presumably based on outside knowledge not directly encoded in the safety policy). In our system, ignoring the compensation code requires an explicit function call, which can be viewed as a programmer declaring that the compensation should be run only to handle a run-time error. The programmer need only write code for the small number of non-run-time error paths in this manner, rather than the large number of implicit control-flow paths associated with run-time errors. Additional compensation stacks may be declared to create a "nested transaction" effect.

[8]Neither Java finalizers nor POSIX cleanup handlers propagate such exceptions. LISP's `unwind-protect` may not execute all cleanup actions if one raises an exception. In analogous situations, C++ aborts the program. Since our goal is to keep the program running and restore invariants, we choose to log such exceptions. Ideally, critical compensations would contain their own internal compensation stacks for error handling. A second option would be to have the type system statically verify that a compensation cannot raise an exception. In the particular example of Java, this solution is not desirable. First, it would require checking *unchecked* exceptions, which is non-intuitive to most Java programmers. Second, most compensations can, in fact, raise exceptions (e.g., `close` can raise an `IOException`).

8.2 Compensation Stack Implementation

We implemented compensation stacks using a source-level transformation tool for Java programs. This entails defining a `CompensationStack` class, adding support for closures (as in the Pizza project [Odersky and Wadler 1997]), and adding convenient syntactic sugar for lexically-scoped compensation stacks. Users of the tool write Java files with the compensation stack syntax described below and use a variant of `javac` that saves the original file, applies the transformation, compiles the resulting pure Java file, and then replaces it with the original source. This approach has the advantage of transparency to the user: normal build processes can be used. It has the disadvantage that debugging the resulting program is more difficult as bytecode line-number information related to compensation stacks will not match the original source code. For example, single-stepping through the code that manages a compensation stack would be obscure.

Consider again the client code from Figure 1. The first step in our approach is to annotate the interface of methods that acquire important resources. For example, we would associate with the action `getConnection` the compensation `close` at the interface level so that all uses of `Connections` can be affected. Consider this code:

```
public Connection getConnection() throws SQLException {
  /* ... do work ... */
}
```

We would change it so that a `CompensationStack` argument is required. The syntax `compensate { a } with { c } using (S)` corresponds to executing the action `a` and then pushing the compensation code `c` on the stack `S` if `a` completed normally. The modified definition follows:

```
public Connection getConnection(CompensationStack S)
  throws SQLException {
  compensate {
    /* ... do work ... */
  } with {
    this.close();
  } using (S);
}
```

Note that this sort of interface change is not required by our approach. It is entirely possible for each client to use a local `CompensationStack`, as in:

```
  compensate { c = getConnection(); }
  with       { c.close(); }
  using (S) ;
```

We believe it may be easier to modify the method signature once, since there are presumably many more uses than definitions. Any annotation requirement, however, limits the applicability of our approach.

As we mentioned in Section 7, this mechanism has the advantages of early release and proper ordering over just using finalizers. Not all actions and compensations must be associated at the function-call level; arbitrary code can be placed in compensations. After annotating the database interface with compensation information, the client code might look like this:

```
Connection cn;
PreparedStatement ps;
ResultSet rs;
CompensationStack S = new CompensationStack();
try {
  cn = ConnectionFactory.getConnection(S, /* ... */);
  StringBuffer qry = ...; // do some work
  ps = cn.prepareStatement(S, qry.toString());
  rs = ps.executeQuery(S);
  ... // do I/O-related work with rs
} finally {
  S.run();
}
```

As the program executes, closures containing compensation code are pushed onto the `CompensationStack S`. Compensations are recorded at run-time, so resources can be acquired in loops or other procedures. Before a compensation stack becomes inaccessible, all of the associated compensations must be executed. A particularly common use involves lexically scoped compensation stacks that essentially mimic the behavior of destructors. We add syntactic sugar allowing a keyword (e.g., `methodScopedStack`) to stand for a compensation stack that is allocated at the beginning of the enclosing scope and `finally` executed at the end of it. In addition, we optionally allow that special stack to be used for omitted compensation stack parameters. We thus arrive at a simple, six-line version of the original client code:

```
Connection cn;
PreparedStatement ps;
ResultSet rs;
cn = ConnectionFactory.getConnection(/* ... */);
StringBuffer qry = ...; // do some work
ps = cn.prepareStatement(qry.toString());
rs = ps.executeQuery();
... // do I/O-related work with rs
```

All of the release actions are handled automatically, even in the presence of run-time errors. An implicit `CompensationStack` based on the method scope is being used and the resource-acquiring methods have been annotated to use such stacks.

Compensations can contain arbitrary code, not just method calls. For example, consider this code fragment adapted from Brown and Patterson [2003]:

```
try {
  StartDate = new Date();
  try {
    StartLSN = log.getLastLSN();
    ... // do work 1
    try {
      DB.getWriteLock();
      ... // do work 2
    } finally {
      DB.releaseWriteLock();
      ... // do work 3
    }
  } finally {
    StartLSN = -1;
  }
} finally {
  StartDate = null;
}
```

We might rewrite it as follows, using explicit `CompensationStacks`:

```
CompensationStack S = new CompensationStack();
try {
  compensate { StartDate = new Date(); }
  with       { StartDate = null; } using (S);
  compensate { StartLSN = log.getLastLSN(); }
  with       { StartLSN = -1; } using (S);
  ... // do work 1
  compensate { DB.getWriteLock(); }
  with       { DB.releaseWriteLock();
               ... /* do work 3 */ } using (S);
  ... // do work 2
} finally {
  S.run();
}
```

Resource finalization and state changes are handled by the same mechanism and benefit from the same ordering. The assignments to `StartLSN` and `StartDate` as well as "`work 3`" are examples of state changes that are not method invocations. This also has the advantage that "undo" code is close to its "do" counterpart.

Traditional destructors are tied to objects, and there are many cases where a program would want to change the state of the object rather than destroying it. Destructors could be used here by creating "artificial objects" that are stack-allocated and perform the appropriate state changes on the enclosing object. However, such a solution would not be natural. For example, the program from which the last example was taken had 17 unique compensations (i.e., error-handling code that was site-specific and never duplicated) with an average length of 8 lines and a maximum length of 34 lines. Creating a new artificial object for each unique bit of error-handling logic would be burdensome, especially since many of the compensations had more than one free variable (which would generally have to be passed as extra arguments to the helper constructor). Nested `try-finally` blocks could also be used but are error-prone (see Section 2.1 and Section 6).

In practice, it is sometimes useful to run compensations "early" rather than in a strict last-in-first-out order. At run-time, each compensation is associated with an identifier j. This identifier is returned by the `compensate-using` expression. It can be retained by the programmer to run that compensation early or it can be ignored. In addition, the programmer can specify a that a particular identifier should be used, using syntax like:

```
compensate { sock = new Socket(); } with { sock.close(); }
using (compStack) id (sock);
```

This allows the programmer to use `compStack.runEarly(sock)` to run the compensation associated with the `sock` object. In our implementation compensation stacks are maintained as doubly linked lists that are also indexed by a hash table. The `runEarly()` method uses the hash table to provide keyed access to compensations. We treat instances of `runEarly()` as annotations that the corresponding socket can be executed outside of the normal last-in-first-out order.

Programmers should follow certain guidelines when programming with compensation stacks. For example, the intended lifetime of a compensation stack should just exceed the intended lifetime of all of its compensations. If a `methodScopedStack` is used to hold compensations for a resource that is allocated with the intent of being used by the callers of that method, the resource will be released prematurely when that method terminates. Similarly, local resources tracked by a global compensation stack might not be released early enough. We recommend that compensation stacks only be passed from callers to callees. If a method allocates a resource that is intended to be used by its callers, the caller should pass in a compensation stack to hold that resource's compensation. This is a more general instance of the `getConnection()` example above: a constructor or allocator that returns a resource should have its interface modified so that it accepts a compensation stack. In general, correctly associating resources and compensation stacks is akin to the problem of associating memory allocations with regions [Gay and Aiken 1998].

Previous approaches to similar problems can be vast and restrictive departures from standard semantics (e.g., linear types or transactions) or lack support for common idioms (e.g., running or discharging obligations early). We designed this mechanism to integrate easily with new and existing programs, and we needed all of its features for our case studies. Using compensation stacks, we found it easy to avoid the mistakes that resulted in defects that were reported hundreds of times in Section 6. In the common case of a lexically scoped linear saga of resources, the run-time error-handling logic needs to be written only once with an interface, rather than every time a resource is acquired. In more complicated cases (e.g., storing compensations in heap variables and associating them with long-lived objects) extra flexibility is available when it is needed.

e ::=	$\textsf{skip}$	no-op
|	e_1 ; e_2	sequencing
|	$\textsf{if} * \textsf{then}\ e_1\ \textsf{else}\ e_2$	non-deterministic choice
|	$\textsf{while} * \textsf{do}\ e$	non-deterministic looping
|	$\textsf{let}\ c_i = \textsf{new CompStack() in}\ e$	compensation stack creation
|	$\textsf{compensate}\ a_j\ \textsf{with}\ b_j\ \textsf{using}\ c_i$	compensation stack use
|	$\textsf{store}\ c_i$	store a stack in memory (address not modeled)
|	$\textsf{let}\ c_i = \textsf{load in}\ e$	load a stack from memory (address not modeled)
|	$\textsf{run}\ c_i$	discharge all of a stack's obligations
|	$\textsf{runEarly}\ a_j\ \textsf{from}\ c_i$	discharge one obligation early

Fig. 12. A simple expression language with compensation stacks.

9. COMPENSATION STACK STATIC SEMANTICS AND DESIGN

We provide a simple static type system for the correct use of explicitly declared compensation stacks. This allows us to highlight the differences between our system and a full linear type system for tracking resources. It also provides a framework in which to describe the ordering guarantees provided by our system. The theoretical developments in this chapter formalize how we track compensation stacks but not individual compensations at run-time, and thus avoid resource-allocation defects.

9.1 Static Semantics

Figure 12 shows a simple expression language involving compensation stacks. Normal program variables (e.g., integers) and objects (e.g., `Sockets`) are abstracted away. The join points after the non-deterministic conditional and loop suffice to model arbitrary control flow. We use c_i to refer to a particular compensation stack, b_j to refer to particular single compensation, and a_j to refer to the action associated with compensation b_j.

The compensation expressions are as described in Section 8.2. Each static let $c_i =$ new CompStack() in e in the program is annotated with a fresh i (which can be thought of as the line number on which it occurs) for bookkeeping purposes. Unless c_i is stored in memory, its scope is limited to e in the sense that all of is compensating actions must be executed before the end of e. The store c_i expression represents storing a compensation stack in a global variable and setting a finalizer to run its compensations. Perhaps the most important detail is that run c_i and runEarly a_j from c_i remove compensations from stacks after executing them at run-time. Compensations are removed from stacks even if those stacks are stored in memory or global variables. Thus, there is no danger of "double-frees" or "free-and-then-finalize-and-then-free" in calling run c_i multiple times.

Each compensation stack in this system is similar to a tracked resource in a linear type system [DeLine and Fähndrich 2001]. Whenever a compensation stack is in scope, we know statically at what location i it was allocated (or loaded). The typing rules for compensation stacks are orthogonal to the typing rules for normal program variables and objects. The goal of the type system is to reject programs in which it cannot be guaranteed that all compensations will be executed.

$$\frac{}{C, D \vdash \mathsf{skip} : C, D}\ \mathit{skip} \qquad \frac{C_1, D_1 \vdash e_1 : C_2, D_2 \quad C_2, D_2 \vdash e_2 : C_3, D_3}{C_1, D_1 \vdash e_1\ ;\ e_2 : C_3, D_3}\ \mathit{seq}$$

$$\frac{C_1, D_1 \vdash e_1 : C_2, D_2 \quad C_1, D_1 \vdash e_2 : C_3, D_3 \quad C_2 \cup D_2 = C_3 \cup D_3}{C_1, D_1 \vdash \mathsf{if} * \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : (C_2 \cup C_3), (D_2 \cap D_3)}\ \mathit{if}$$

$$\frac{C_1, D_1 \vdash e : C_2, D_2 \quad C_1 \cup C_2 = C_2 \cup D_2}{C_1, D_1 \vdash \mathsf{while} * \mathsf{do}\ e : C_1 \cup C_2, D_1 \cap D_2}\ \mathit{while}$$

$$\frac{C_1, D_1 \cup \{i\} \vdash e : C_2, D_2 \quad D_3 = D_2 \setminus \{i\}}{C_1, D_1 \vdash \mathsf{let}\ c_i =\ \mathsf{new\ CompStack()}\ \mathsf{in}\ e : C_2, D_3}\ \mathit{let}$$

$$\frac{i \in C}{C, D \vdash \mathsf{compensate}\ a_j\ \mathsf{with}\ b_j\ \mathsf{using}\ c_i : C, D}\ \mathit{compC}$$

$$\frac{D_2 = D_1 \setminus \{i\}}{C, D_1 \vdash \mathsf{compensate}\ a_j\ \mathsf{with}\ b_j\ \mathsf{using}\ c_i : C \cup \{i\}, D_2}\ \mathit{compD}$$

$$\frac{C_2 = C_1 \setminus \{i\}}{C_1, D \vdash \mathsf{store}\ c_i : C_2, D \cup \{i\}}\ \mathit{storeC} \qquad \frac{i \in D}{C, D \vdash \mathsf{store}\ c_i : C, D}\ \mathit{storeD}$$

$$\frac{C_1 \cup \{i\}, D_1 \vdash e : C_2, D_2 \quad C_3 = C_2 \setminus \{i\}}{C_1, D_1 \vdash \mathsf{let}\ c_i =\ \mathsf{load}\ \mathsf{in}\ e : C_3, D_2}\ \mathit{loadC}$$

$$\frac{C_1 \cup \{i\}, D_1 \vdash e : C_2, D_2 \quad D_3 = D_2 \setminus \{i\}}{C_1, D_1 \vdash \mathsf{let}\ c_i =\ \mathsf{load}\ \mathsf{in}\ e : C_2, D_3}\ \mathit{loadD}$$

$$\frac{C_2 = C_1 \setminus \{i\}}{C_1, D_1 \vdash \mathsf{run}\ c_i : C_2, D \cup \{i\}}\ \mathit{runC} \qquad \frac{i \in D}{C, D \vdash \mathsf{run}\ c_i : C, D}\ \mathit{runD}$$

$$\frac{i \in C \cup D}{C, D \vdash \mathsf{runEarly}\ a_j\ \mathsf{from}\ c_i : C, D}\ \mathit{early}$$

Fig. 13. Expression language static semantics.

A compensation stack, which can potentially store un-executed compensations, is only allowed to go out of scope if it is stored in a global variable or if we can prove statically that all of its compensations have been executed. We approximate this by requiring that run c_i or store c_i occur after the last compensate a_j with c_j using c_i before c_i goes out of scope. Our typing judgment maintains two *disjoint* sets: C, a set of “active” compensation stacks that may have unexecuted compensations, and D, a set of “inactive” compensation stacks on which all compensations have been executed or stored in memory. Together, C and D contain all in-scope compensation stacks. Adding a compensation to a inactive stack makes it active. Thus, we propose an effect type system for compensation stacks.

The form of our typing judgment is $C, D \vdash e : C', D'$. This judgment says that expression e typechecks in the context of the set of active compensation stacks C and the set of unused stacks D and that after executing the expression the set of active stacks will be C' and the set of unused stacks will be D'.

Figure 13 shows the typing rules for the language in Figure 12. Note that whenever we write a set difference $C \setminus \{i\}$ we require $\{i\} \in C$. The *seq* rule shows that this is a flow-sensitive type system for compensation stacks. The *if* rule is presented in slightly more generality than is warranted by our simple language. Recalling the invariant that $C \cap D = \emptyset$, at the join point of the conditional the resulting active set C_3 contains all of the stacks that might possibly be active after either branch and the inactive set D_3 contains all of the stacks that are definitely inactive after both branches. The rule is thus conservative. The $C_2 \cup D_2 = C_3 \cup D_3$ requirement prevents the program from creating a new compensation stack on one branch of the conditional. This is impossible in our simple example language, because newly-created compensation stacks have local scope, but is possible in our Java implementation.

The *while* rule is also conservative. If the loop body can make a stack active, we assume that it does. If the loop body can make a stack inactive, we assume that it does not (and thus a well-typed program will have to run those compensation stacks again later, even if they are empty).

The *let* rule introduces a new compensation stack and requires that it be inactive as it goes out of scope. The *comp* rules are simple since managing the stacks is deferred to run-time. Adding a compensation to a inactive stack makes it active and compensations can only be added to valid stacks that are currently in-scope.

The *store* rules simulate the user storing a compensation stack in a global variable and consigning ultimate care of it to the garbage collector. When it is finalized the run-time system will execute any remaining compensations associated with it. The *storeD* rule for storing a inactive stack is provided for completeness. There is rarely a reason to store a stack with no outstanding obligations.

The *loadC* and *loadD* rules are similar to the *let* rule except that the stack c_i need not be inactive as it goes out of scope at end of e. Instead, the program is obligated to discharge any compensations in heap-stored compensation stacks at some later point (e.g., in a different block containing let $c_i =$ load in e or via a finalizer). Compensation stacks stored in memory are not tracked as precisely as those stored in local variables or method arguments. If all of the obligations in c_i have been discharged (i.e., if $i \in D_2$ and the *loadD* rule is used) then nothing remains to be done. If some of the compensations in c_i remain outstanding (i.e., if $i \in C_2$ and the *loadC* rule is used) then the outgoing C_3 will not contain i: the active and inactive sets in these judgments refer only to compensation stacks that are not stored in memory.

The *run* rules execute all remaining compensations in the given stack and ensure that it is inactive. The *run* and *store* rules are the only way to move a stack from the active set C to the inactive set D, so every stack must pass through a *run* or *store* rule at least once just before going out of scope.

The *early* rule models our syntax for allowing the user to optionally execute certain compensations early, if desired. The runEarly a_j from c_i expression does remove the compensation b_j associated with a_j from stack c_i if b_j was present. Our run-time tracking of compensations thus prevents "double frees". However, our static type system tracks compensation stacks, not individual compensations. This has the benefit of making the system tractable, since the number of compensation stacks is much smaller than the number of compensations. Note that b_j does *not* have to be the topmost compensation on the stack c_i. If the *early* rule is used, compensations on a particular stack will not exhibit a strict last-in first-out order. The *early* rule is provided as an escape hatch for programmers and is not intended to be used commonly; we treat it as an annotation that running b_j early is safe.

In any event, if b_j, the particular compensation associated with a_j, has already been executed or is otherwise no longer on the appropriate stack, nothing happens at run-time. Since we do not track individual compensations, we cannot know if the last outstanding compensation has been discharged by this rule, so the *early* rule cannot make a stack inactive.

We say that a program e typechecks if $\emptyset, \emptyset \vdash e : \emptyset, \emptyset$. Our system can be viewed as a linear type system for sets of resources rather than a linear type system for individual resources. A program containing a loop that allocates resources and puts obligations to deallocate them on a stack c_i can be statically type-checked provided that run c_i occurs after any compensations are added to c_i on all paths containing c_i before it goes out of scope. Similarly, programs in which only one branch of a conditional adds an obligation to a compensation stack are handled naturally. We also expect that it will be easier to avoid creating aliasing of compensation stacks than it is to avoid creating aliases of individual resources (e.g., in the same way that it is easier to manually allocate and destroy regions of objects then it is to manually use `malloc` and `free` for individual objects).

We do not discuss method calls and returns here. An annotation system similar to the one described in Vault [DeLine and Fähndrich 2001] suffices: each function type specifies its requirements for compensation stacks and how it transforms them (e.g., requiring two active stack arguments and ensuring that the first one is inactive when it returns). The type system is also amenable to the standard extension for handling exceptions (i.e., extend the judgment to produce one pair C, D representing normal termination and another pair C', D' for exceptional termination).

9.2 Compensation Stack Weaknesses

Modifying a program to use compensation stacks is an invasive transformation that may be difficult to reason about. One major concern is that the transformation is not always semantics-preserving. At one level this is intentional: the transformed program typically calls cleanup code exactly once per resource, rather than zero times (e.g., a resource leak) or multiple times (e.g., a double free). Inasmuch as the transformation fixes defects, it must fail to preserve the exact program semantics. However, a `finally` block in the original program may not execute at the same point as a compensation in the transformed program. If there are dependencies between statements in the `finally` block and statements after it, the original and transformed program may behave differently. For example, in the `startLSN` code in Section 8.2, in which a `finally` block sets `startLSN` to -1, a statement after that finally block may expect `startLSN` to be -1. If the assignment is placed in a compensation it may not be executed immediately, and later code that depends on `startLSN`'s value will not behave correctly.

The danger of out-of-order compensations is very real, but is mitigated by the idiomatic way in which Java programs handle cleanup. For example, none of the defects presented in Section 6.2 and none of the error handling in either of the case studies discussed in the next section feature dependencies between the cleanup code and subsequent statements in that method. We do not believe such out-of-order problems would be common in practice. They are certainly possible, however, especially for programmers unfamiliar with the new compensation semantics.

Beyond truly convoluted cases of exception handling, programming with compensation stacks may not necessarily be simpler than using traditional exception handling. The reduction from "`try { s = new Socket(); } finally { s.close(); }`" to "`s = new Socket();`" with implicit stacks and annotated interfaces adds an optional argument whose flow must be tracked. We believe that tracking specialized implicit value flow (to correctly handle resources using compensation stacks) may be easier than tracking the implicit control-flow of exceptions (to correctly handle resources using `try`), but this will not always be the case. A software project that already has a strong `try-finally-if-close` discipline for releasing resources will not gain from adopting our approach.

The use of compensation stacks, which are essentially lists of function pointers, may complicate static analyses such as control flow analysis or data slicing more than the original implicit control flow from exceptions did [Sinha and Harrold 2000]. In this regard, compensation stacks are as difficult to analyze as finalizers, but would occur frequently in the transformed program.

The use of compensation stacks also adds a new language feature that does not have standard language semantics. The compensation stack model of building up obligations which are later discharged may be unintuitive to programmers. As a result, training may be required before they can be used in a less error-prone fashion than standard `try-finally` blocks.

A simpler "logging" approach might modify the program to record resource acquisitions and releases at run-time. The program could then be fixed if any resource leaks are detected in the log. Such a logging approach has many potential advantages and disadvantages over compensation stacks: (1) there is no danger of executing cleanup actions out of order; (2) subsequent analyses and debugging efforts are not complicated; (3) no features with nonstandard semantics are added; (4) only defects that appear at run-time will potentially be fixed; and (5) the run-time monitoring can be disabled if desired, either program-wide or for a specific region.

We view addressing defects that have not yet been identified using the test set as an advantage for compensation stacks. Formulating test cases and test metrics for exceptional situations can be difficult [Sinha and Harrold 1999; Malayeri and Aldrich 2006]. If the use of compensations does not introduce additional defects from out-of-order execution, then additional error-handling defects are fixed early. The logging approach would need an indicative test set in order to find defects, just as other dynamic analyses do (e.g., Savage et al. [1997]).

In addition, as we shall argue in the next section, the run-time overhead of using compensation stacks is low, even if they are always on. The logging approach has the advantage of being able to reduce that overhead. However, if the program is evolved and maintained over time, the compensation stacks may prove an advantage. For example, if a second `Socket` is added to a system, the logging approach must re-enable logging and test both the resource acquire and the resource release. A system already using compensation stacks continues to pay the run-time overhead and must test the resource acquire, but the resource release using the compensation stack mechanism has presumably already been verified (especially when using the annotated interface approach from Section 8.2).

Compensation stacks can complicate subsequent analyses. However, the compensation syntax makes it clear what code is the compensation. For compensation stacks not stored in the heap, that code can only be executed at `run()` or `runEarly()` method invocations. In the common case of implicitly scoped compensation stacks, the disruption of control is regular: if action a_j is reached, then compensation b_j will be executed just before the end of the method. Analyses could be slightly extended to handle control-flow graphs with simple compensation stacks in a similar spirit to way that they are extended for factored control-flow graphs [Choi et al. 1999]. Ultimately, however, this is a weakness of a compensation stack approach, and uses of compensations can confuse static analyses.

We believe that the dangers of run-time overhead and out-of-order execution are minimal in practice and that being able to correct error-handling defects for which no test cases have been developed is an advantage. However, there are many situations in which the logging approach described here would be a better fit, and users of compensation stacks should be aware of their weaknesses.

10. CASE STUDIES

We hand-annotated two programs to show that the run-time overhead is low and that existing programs can be rapidly modified to use compensation stacks. Guided by the data-flow analysis in Section 5, the programs were modified so that their existing run-time error handling made use of compensation stacks; no truly new run-time error handling was added (even when inspection revealed it to be missing) and the behavior was otherwise unchanged. In the common case this amounted to removing an existing `close` call (and possibly its guarding `finally`) and using a `CompensationStack` instead (possibly with a method that had been annotated to take a compensation stack parameter). Maintaining the stacks and the closures takes time, but that overhead was dwarfed by the I/O latency in our case studies. As a micro-benchmark example, a simple program that creates hundreds of `Sockets` and connects each to a website is 0.7% slower if a compensation stack is used to hold the obligation to close the `Socket`.

The subject for the first case study, Aaron Brown's undo-able email store [Brown and Patterson 2003], can be viewed as an SMTP and IMAP proxy that uses database-like logging. The original version was 35,412 lines of Java code. Annotating the program took about four hours and involved updating 128 sites with code to use compensations as well as annotating the interfaces for some standard library methods (e.g., `sockets` and databases). The resulting program was 225 lines shorter (about 1%) because redundant run-time error-handling code and control-flow were removed. The program contains non-trivial error handling, including one five-step saga of actions and compensations and one three-step saga. Single compensating actions ranged from simple `close` calls to 34-line code blocks with internal exception handling and synchronization. The annotated program's performance was almost identical to the original on fifty micro-benchmarks and one example workload (all provided by the original author). Performance was measured to be within one standard deviation of the original; the overhead associated with keeping track of obligations at run-time was dwarfed by I/O and other processing times. For example, the annotated program took 289.1 seconds to complete the example workload compared to 286.9 seconds for the original average over five trials with a standard deviation of 12.4 seconds.

Compensations were used to handle every request answered by the program. By changing a method invocation in some insufficiently-guarded cleanup code to always raise one of its declared exceptions in both versions of the program, we were able to cause the unmodified version of the program to drop all SMTP requests. The version using compensations handled that cleanup correctly (i.e., without leading to a failure) and proceeded normally. While this sort of targeted fault injection is hardly representative, it does show that the defects we are addressing with compensations can have an impact on reliability.

The subject for the second case study, Sun's `Pet Store 1.3.2` [Sun Microsystems 2001], is a web-based, database-backed retailing program. The original version was 34,608 lines of Java code. Annotations to 123 sites took about two hours. The resulting program was 168 lines smaller (about 0.5%). Most error-handling annotations centered around database `Connections`. Using an independent workload [Chen et al. 2002; Candea et al. 2003], the original version raises 150 exceptions from the `PurchaseOrderHelper`'s `processInvoice` method over the course of 3,900 requests. The exceptions signal run-time errors related to `RelationSets` being held too long (e.g., because they are not cleared along with their connections on some paths) and are caught by a middleware layer which restarts the application.[9] The annotated version of the program raises no such exceptions: compensation stacks ensure that the database objects are handled correctly. The average response times for the original program (over multiple runs) is 52.06 milliseconds (ms), with a standard deviation of 100 ms. The average response time for the annotated program is 43.44 ms with a standard deviation of 77 ms. The annotated program is more consistent, and, because less middleware intervention was necessary, the program-and-middleware system was 17% faster.

In these case studies we were able to rapidly annotate existing programs to use compensation stacks. The resulting programs did not suffer an undue performance overhead. Finally, the checks ensure that cleanup code is invoked correctly along all paths through the program.

[9]While updating a purchase order to reflect items shipped, the `processInvoice` method creates an `Iterator` from a `RelationSet Collection` that deals with persistent data in a database. Unfortunately, the transaction associated with the `RelationSet` has already been completed.

11. RELATED WORK

Related work falls into five broad categories: analyses in the presence of exception, approaches to cleaning up resources, type systems, reliability and exception handling, and transactional models.

11.1 Analyses in The Presence of Exceptions

Sinha and Harrold [1999] present multiple testing criteria for programs that use exception-handling constructs, as well as describing how to compute testing requirements using those constructs. Defects in exception-handling code can be hard for programmers to detect: testing metrics that make it clear that certain exceptional control-flow paths are not being examined would make it easier to find and fix such defects. In later work [Sinha and Harrold 2000], they describe the effects of language-level exception handling on techniques such as control dependence, data-flow and control flow analysis. Their observations, such as noting that data-flow facts must be propagated along exceptional control-flow paths, apply to our data-flow analysis. More recently [Sinha et al. 2004], they use such static and dynamic analyses to guide software development and maintenance tasks in the presence of implicit control flow from polymorphism or exceptions.

Malayeri and Aldrich [2006] have proposed a lightweight system for specifying exceptions. They infer information and take advantage of user annotations, but they assume that whole-program analyses are undesirable. Since we are not concerned with evolving a program over time and verifying its compliance, we place less emphasis on annotation and more on defect reports without false positives.

Gupta et al. [2000] support optimization in the presence of exceptions via static and dynamic analyses that allow some exception-causing instructions to be ignored while applying certain optimizations. Their analysis determines the portion of the program state that can be discarded if an exception occurs and has been used successfully in the Jalapeño Java compiler [Burke et al. 1999]. Our fault model is concerned only with method dispatches that raise exceptions: they consider a much broader class of *potential exception-throwing instructions*. The Jex tool [Robillard and Murphy 2003] and others (e.g., Chang et al. [2001]) have investigated the flow and reach of exceptions: exceptions can often end up escaping module boundaries with unintended consequences. Determining which exceptions can reach which program points is essential for ensuring reliability in systems with language-level exception handling. In our analysis we do not examine what happens to an exception after it causes the program to violate a specification.

The *factored control-flow graph* (FCFG) model of Choi et al. [1999] efficiently represents programs with exceptional control flow without losing precision. In an FCFG, exception-throwing instructions do not force the end of a basic block. This changes the dominance relation: an instruction does not necessarily dominate all subsequent instructions in its basic block. Program analyses on an FCFG must be extended to recognize this change. They explicitly support the sort of global data-flow analysis we present in Section 5. Since we do not consider as many potentially exception-throwing instructions (e.g., we do not consider unchecked exceptions), and do not need large basic blocks for program optimization, we construct a more direct CFG with a standard dominance relation. Their FCFG model is strictly more general than ours, and our analysis could be performed in their framework.

The SABER project [Reimer et al. 2004] uses a small set of pattern-like rules to detect defects in Java programs. It is similar to the Metacompilation [Engler et al. 2000] and ESP [Das et al. 2002] projects. The scope of SABER is quite broad, and includes defects related to, for example, incorrectly storing objects and incorrectly implementing special paired methods. SABER has specific handling for "must call `X` after `Y`" rules, and our "must call `close()` after `new`" analysis in Section 5 is a particular instance of that. One of the SABER case studies analyzes closing database connections instead of using finalizers. One primary difference is that SABER's filtering rules are less aggressive than ours: 15 of the 94 defect reports from their four case studies were false positives. Beyond that, SABER is more general than the analysis we present in Section 5. Our additional checks for `ResultSets` and `Statements` could be codified as SABER rules.

Bruntink et al. [2006] analyze the exception handling of an industrial software system, paying special attention exception raising and logging. In their system exceptions are propagated by return codes, and they present a formal fault model for such an environment.

Fink et al. [2006] present a flow-sensitive, context-sensitive typestate verifier for Java programs. Their analysis handles the same sorts of specifications that we do but is much more precise and integrates aliasing information. However, their work does not address the issue of exceptional control-flow directly.

11.2 Cleaning Up Resources

Beyond destructors and finalizers there are a number of existing approaches that are similar in spirit to our compensation stacks.

Common Lisp's "`unwind-protect` *body cleanup*" syntax behaves like `try-finally` and ensures that *cleanup* will be executed no matter how control leaves *body*. To handle a common case, the macro "`with-open-file` *stream body*" opens and closes *stream* automatically as appropriate. Since Lisp comes with first-class functions and macros, `unwind-protect` can be used more conveniently than Java's `try-finally` with respect to duplicate and unique run-time error handling. However, it still suffers from many of the same limitations (e.g., no easy way to discharge obligations early, one nesting level per resource, one global stack). In Scheme "`dynamic-wind` *before work after*" and `call-with-open-file` serve similar purposes, although `dynamic-wind` is complicated by the presence of continuations (e.g., the dynamic extent of *work* may not be a single time period).

The POSIX thread library (IEEE 1003.1c-1995) provides a per-thread cancellation cleanup stack (`pthread_cleanup_push` and `_pop`). The cleanup routines are executed when the thread exits or is canceled. However, the cleanup stack is not a first-class object, so cleanup code must be associated with the thread and not with an object. In addition, only the most recently-added cleanup code can be executed early or removed from the stack. Also, those two actions may only be taken inside the same lexical scope as their corresponding `push`. The stack uses C-style function pointers, so general run-time error-handling (like that of `undo` in Section 10) requires the creation of separate functions. Finally, the mechanism can only be used safely in "deferred cancellation mode" because performing the action and pushing the cleanup code are not done atomically with respect to thread cancellation. Our `compensate-with` expression handles this issue in Java, where thread cancellation is signaled via exceptions.

The `Cleanup Stack` programming convention is used by C++ programs that run on the Symbian embedded OS. The Symbian OS is typically used for cell phones and other environments where memory is a particularly scarce resource and every effort is made to keep track of and release it. A Symbian `Cleanup Stack` keeps track of local pointers to memory and frees them automatically if some intermediate computation terminates with an exception [van der Wal 2002]. There is a single global `Cleanup Stack` and only one type of resource (i.e., explicitly managed memory) is supported. In addition there is no support for freeing memory early along some paths.

The GNU Debugger `gdb` uses `cleanups` as "a structured way to deal with things that need to be done later" [Stallman et al. 2002], `Cleanups` are executed when `gdb` commands are finished, when an exceptional situation occurs, or on explicit request. A `cleanup` is a chain of function pointers and arguments. `Cleanup` chains do not support arbitrary closures and can be awkward when more than one local variable must be referenced by the postponed action. In addition, their default execution behavior is somewhat tied to `gdb`'s top-level command loop.

11.3 Type Systems

Flow-sensitive type systems check many of the same safety properties that our system enforces. The key difference is that a strong type system will reject a program that cannot be statically shown to adhere to the safety policy, whereas our system will use run-time instrumentation to ensure compliance.

DeLine and Fähndrich [2001] propose the Vault language and static linear type system for enforcing high-level software protocols. Vault represents a different point in the design space, with more powerful properties but a more difficult programming model. It can verify that operations are performed on resources in a certain order (e.g., that `open` is called before `read`), while we cannot. It can also ensure that an operation is in a thread's computational future (e.g., that an `open`ed resource is `close`d by the end of the method). Vault's *keys* represent the right to perform certain operations on objects. Keys can be in various states (e.g., open or closed). Vault's variant keys (e.g., special objects that are either empty or contain a key) can be used to free an object early on one path and free it later on another. These variants require the programmer to make an explicit run-time check to determine if the key has already been freed. Our system handles this aspect slightly more naturally by performing that check automatically. On the other hand, our system lacks stateful keys.

Vault does not support arbitrary polymorphic lists of keys. In Vault, placing a resource in a list makes it anonymous. We can place arbitrary compensations relating to different resources in the same compensation stack.

Perhaps the greatest drawback of Vault is that it requires much of the program to adhere to a linear type system. Linear type systems are generally considered to be difficult to work with, and structuring a program to fit a linear type system is often a herculean task. Later work [Fähndrich and DeLine 2002] extends the Vault type system with additional features that ease the burden of programming with linear types, but aliasing can still be difficult. However, our basic approach cannot be modeled in a standard linear type system. In our approach, the compensation stack holds a reference to the tracked resource while the program continues to manipulate that same resource. In a standard linear type system, such aliasing cannot be allowed.

11.4 Reliability and Run-Time Error Handling

Quite a bit of attention from a number of research communities has been devoted to issues of run-time error handling in long-running processes and general software systems. Broadly speaking, expressive systems for signaling and handling run-time errors are considered integral to the reliability of large-scale software systems.

Alonso et al. [2000] believe that poor support for exception handling is a major obstacle for large-scale and mission-critical systems.

Hagen and Alonso [2000] claim that exception handling must be separated from normal code if processes are to be reused like libraries. This separation is similar to our goal of annotating interfaces with compensation information.

Dony [2001] describes an object-oriented exception handling system where all exception handlers have a dynamic call-stack scope. Dony's form of `unwind-protect` is similar to our approach, although it offers no support for discharging obligations early or for a first-class handling of the current set of pending obligations.

Miller and Tripathi [1997] note that requirements of object-oriented design, such as specialization and evolution, can conflict with language-level exception handling mechanisms. Similarly, Cargill [1994] argues that without extraordinary care exceptions actually diminish the overall reliability of software. The hard part of exception handling is not raising exceptions but writing the support code so that exceptional situations are handled correctly. Our technique is particularly well-suited to handling the matched acquire-free behavior in his presentation.

Valetto and Kaiser [2002] note that adaptation to run-time errors usually involves several conditional or dependent activities that may fail; the linear saga model we support is rich enough to capture many dependent activities.

Cardelli and Davies [1999] present a language for writing programs with an explicit notion of failure. We have a less holistic notion of run-time errors but have an easier time integrating with existing code.

Demsky and Rinard [2003] allow defects in key data structures to be repaired at run-time based on specifications. Their technique works at the level of data structures and not at the level of program actions, and it may be viewed as addressing an orthogonal problem. For example, their approach does not lend itself naturally to I/O-based repairs and ours does not handle logical defects in compensation code.

The VINO operating system [Seltzer et al. 1996] uses software fault isolation and lightweight transactions to address problems like resource hoarding in user-defined kernel extensions. This form is similar to our approach in that an interface has been annotated with compensations that are called if a fatal exceptional situation occurs. However, in VINO there is only one compensation stack per extension, and it is not a first-class object. In addition, there is no support for nested transactions without defining additional extensions.

11.5 Transactions

Database transactions provide a strong and well-founded approach to run-time error handling [Gray 1981]. However, many find the consistency and durability of transactions to be too heavyweight for most programming purposes (e.g., Alonso et al. [2000], Liskov and Scheifler [1983], and Dayal et al. [1990]). For example, Java programs that want transactional support for certain pieces of data (e.g., e-commerce applications updating inventory tables) often make explicit calls to an external database (as in the "Database" policy of Section 3.1). For variables internal to the program, however, other measures are more appropriate.

Restructuring a program to make use of transactions can be a large, invasive change. Borg et al. [1989] describe a checkpointing system that allows unmodified programs to survive hardware failures. Essentially, every system call is intercepted and logged. Others [Schmuck and Wyllie 1991; Lowell and Chen 1998; Shapiro et al. 1999] provide similar services. Our compensation annotations are a much less drastic change to the program semantics than the incorporation of transactions.

In addition, these transaction techniques address an orthogonal run-time error handling issue. In Borg et al.'s system, a process with a defect that acquires a lock twice and deadlocks on initialization will continue to deadlock no matter how many times it is recovered. Lowell et al. [2000] formalize this point by noting that the desire to log all events actually conflicts with the ability to recover from all run-time errors. Such systems are very good at masking hardware failures and quite poor at masking software failures; Lowell et al. suggest that 85–95% of application defects cause crashes that would not be prevented by a failure-transparent operating system. Our technique hopes to address such defects, but it is less automatic.

Many researchers have found that advanced transactional concepts fit closely with language-level run-time error handling [Dan et al. 1998; Liu et al. 2001]. One such concept, the compensating transaction, semantically undoes the effects of another transaction *after* that transaction has been committed [Korth et al. 1990]. Designing a full compensating transaction that completely undoes the effects of a previous action is often difficult. Our system relaxes this requirement by limiting compensations to certain actions (e.g., resource allocation) and by associating compensations with interfaces so they need only be defined once. Alonso et al. [1994] consider the notion of *linear sagas* [Garcia-Molina and Salem 1987] in a similar context. Our system is slightly more general than a pure linear saga [Korth et al. 1990] and more closely resembles a form of nested or interleaved linear sagas.

12. CONCLUSION

Software reliability remains an important and expensive issue. This work presents an approach for addressing a certain class of software reliability problems associated with exceptional situations and language-level error handling.

First, we presented a static data-flow analysis for finding defects in how programs deal with important resources in the presence of exceptional situations. To find defects in programs we formalized some initial specifications of how a program should acquire and release resources. To find defects in exceptional situations we defined a particular fault model to describe what exceptional situations could crop up. The analysis itself was designed to scale well to large programs. We introduced three simple filtering rules to make the analysis easier to use by eliminating false positives. The analysis found over 1,300 methods with defects in almost five million lines of Java code.

Second, given those resource-handling defects in exceptional situation we designed a language feature to make it easier for programmers to avoid making such mistakes. We proposed that programmers keep track of important obligations at run-time in special compensation stacks. We provide a static semantics for compensation stacks to highlight their differences from previous approaches like pure linear type systems. In two case studies we showed that compensation stacks can be rapidly applied to existing Java programs and that they introduce minimal overhead.

Using specifications and our fault model we can analyze programs to find defects. Once defects have been located we can provide programmers with an easy-to-use tool for addressing them. All of this can be done rapidly, before the program is deployed. We believe this work can help to make software more reliable in the presence of exceptional situations.

ACKNOWLEDGMENTS

We thank Aaron Brown for an explanation of and a workload for his `undo` program, as well as for fruitful discussions about run-time error handling. We thank Mark Brody, George Candea and Tom Martell for generously providing their infrastructure and their workload generator for `Pet Store`. We thank Christopher Hylands Brooks for an insightful discussion of run-time error handling in general and `ptolemy2` in particular. Finally, we thank the anonymous referees for going above and beyond the call of duty in offering constructive criticism on earlier drafts.

REFERENCES

ABRIAL, J.-R., SCHUMAN, S. A., AND MEYER, B. 1980. Specification language. In *On the Construction of Programs*. 343–410.

AHO, A., SETHI, R., AND ULLMAN, J. 1986. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, MA.

ALONSO, G., HAGEN, C., AGRAWAL, D., ABBADI, A. E., AND MOHAN, C. 2000. Enhancing the fault tolerance of workflow management systems. *IEEE Concurr. 8*, 3 (July), 74–81.

ALONSO, G., KAMATH, M., AGRAWAL, D., ABBADI, A. E., GUNTHOR, R., AND MOHAN, C. 1994. Failure handling in large-scale workflow management systems. Tech. Rep. RJ9913, IBM Almaden Research Center, San Jose, CA. Nov.

BALL, T. AND RAJAMANI, S. K. 2001a. Automatically validating temporal safety properties of interfaces. In *SPIN 2001, Workshop on Model Checking of Software*. Lecture Notes in Computer Science, vol. 2057. Springer-Verlag, New York. 103–122.

BALL, T. AND RAJAMANI, S. K. 2001b. SLIC: A specification language for interface checking (of C). Tech. Rep. MSR-TR-2001-21, Microsoft Research.

BOEHM, H.-J. 2003. Destructors, finalizers and synchronization. In *Proceedings of the Symposium on Principles of Programming Languages*. ACM, New York.

BORG, A., BLAU, W., GRAETSCH, W., HERRMANN, F., AND OBERLE, W. 1989. Fault tolerance under UNIX. *ACM Trans. Comput. Syst. 7*, 1 (Feb.).

BROWN, A. AND PATTERSON, D. 2003. Undo for operators: Building an undoable e-mail store. In *USENIX Annual Technical Conference*.

BRUNTINK, M., VAN DEURSEN, A., AND TOURWÉ, T. 2006. Discovering faults in idiom-based exception handling. In *ICSE '06: Proceeding of the 28th International Conference on Software Engineering*. ACM, New York. 242–251.

BURKE, M., CHOI, J., FINK, S., GROVE, D., HIND, M., SARKAR, V., SERRANO, M., SREEDHAR, V., SRINIVASAN, H., AND WHALEY, J. 1999. The jalapeno dynamic optimizing compiler for Java. In *Proceedings of the ACM 1999 Java Grande Conference* (San Francisco, CA). ACM, New York. 129–141.

CAMPIONE, M., WALRATH, K., AND HUML, A. 2000. *The Java Tutorial*. Addison-Wesley, Reading, MA.

CANDEA, G., DELGADO, M., CHEN, M., AND FOX, A. 2003. Automatic failure-path inference: A generic introspection technique for internet applications. In *Proceedings of the IEEE Workshop on Internet Applications* (San Jose, CA). IEEE Computer Society Press, Los Alamitos, CA.

CARDELLI, L. AND DAVIES, R. 1999. Service combinators for web computing. *Softw. Eng. 25*, 3, 309–316.

CARGILL, T. 1994. Exception handling: A false sense of security. C++ *Report 6*, 9.

CHANG, B.-M., JO, J.-W., YI, K., AND CHOE, K.-M. 2001. Interprocedural exception analysis for Java. In *SAC '01: Proceedings of the 2001 ACM Symposium on Applied Computing*. ACM Press, New York. 620–625.

CHATTERJEE, R., RYDER, B. G., AND LANDI, W. 2001. Complexity of points-to analysis of Java in the presence of exceptions. *IEEE Trans. Software Eng. 27*, 6, 481–512.

CHEN, M. Y., KICIMAN, E., FRATKIN, E., FOX, A., AND BREWER, E. 2002. Pinpoint: Problem determination in large, dynamic Internet services. In *Proceedings of the International Conference on Dependable Systems and Networks*. IEEE Computer Society, Press, Los Alamitos, CA. 595–604.

CHOI, J.-D., GROVE, D., HIND, M., AND SARKAR, V. 1999. Efficient and precise modeling of exceptions for the analysis of Java programs. In *PASTE '99: Proceedings of the 1999 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. ACM, New York. 21–31.

CRISTIAN, F. 1982. Exception handling and software fault tolerance. *IEEE Trans. Comput. 31*, 6, 531–540.

CRISTIAN, F. 1987. Exception handling. Tech. Rep. RJ5724, IBM Research.

DAN, A., DIAS, D. M., NGUYEN, T., SACHS, M., SHAIKH, H., KING, R., AND DURI, S. 1998. The Coyote project: Framework for multi-party e-commerce. In *Proceedings of ECDL*. Lecture Notes in Computer Science, vol. 1513. Springer-Verlag, New York. 873–889.

DAS, M., LERNER, S., AND SEIGLE, M. 2002. ESP: Path-sensitive program verification in polynomial time. *SIGPLAN Notices 37*, 5, 57–68.

DAYAL, U., HSU, M., AND LADIN, R. 1990. Organizing long-running activities with triggers and transactions. In *Proceedings of ACM SIGMOD* (Atlantic City, NJ). ACM, New York. 204–214.

DELINE, R. AND FÄHNDRICH, M. 2001. Enforcing high-level protocols in low-level software. In *Programming Language Design and Implementation*. 59–69.

DEMSKY, B. AND RINARD, M. C. 2003. Automatic data structure repair for self-healing systems. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, New York.

DONY, C. 2001. A fully object-oriented exception handling system. In *Advances in Exception Handling Techniques*. Lecture Notes in Computer Science, vol. 2022. Springer-Verlag, New York. 18–38.

ENGLER, D., CHELF, B., CHOU, A., AND HALLEM, S. 2000. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the Symposium on Operating Systems Design and Implementation*.

FÄHNDRICH, M. AND DELINE, R. 2002. Adoption and focus: Practical linear types for imperative programming. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*. ACM, New York.

FINK, S., YAHAV, E., DOR, N., RAMALINGAM, G., AND GEAY, E. 2006. Effective typestate verification in the presence of aliasing. In *ISSTA '06: Proceedings of the 2006 International Symposium on Software Testing and Analysis*. ACM, New York. 133–144.

FU, C., MILANOVA, A., RYDER, B. G., AND WONNACOTT, D. 2005. Robustness testing of Java server applications. *IEEE Trans. Softw. Eng. 31*, 4, 292–311.

FU, C., RYDER, B., MILANOVA, A., AND WANNACOTT, D. 2004. Testing of Java web services for robustness. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*.

GARCIA-MOLINA, H. AND SALEM, K. 1987. Sagas. In *Proceedings of the ACM Conference on Management of Data*. ACM, New York. 249–259.

GAY, D. AND AIKEN, A. 1998. Memory management with explicit regions. In *Prog. Lang. Des. Implement.* 313–323.

GENERAL SERVICES ADMINISTRATION. 1996. Telecommunications: Glossary of Telecommunication terms. Tech. Rep. Federal Standard 1037C, National Communications System Technology & Standards Division. Aug.

GOODENOUGH, J. B. 1975. Exception handling: issues and a proposed notation. *Commun. ACM 18*, 12, 683–696.

GOSLING, J., JOY, B., AND STEELE, G. L. 1996. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, MA.

GRAY, J. 1981. The transaction concept: virtues and limitations. In *Proceedings of the International Conference on Very Large Data Bases* (Cannes, France). ACM, New York. 144–154.

GUPTA, M., CHOI, J.-D., AND HIND, M. 2000. Optimizing Java programs in the presence of exceptions. In *ECOOP '00: Proceedings of the 14th European Conference on Object-Oriented Programming* (London, UK). 422–446.

HAGEN, C. AND ALONSO, G. 2000. Exception handling in workflow management systems. *IEEE Trans. Software Engineering 26*, 9 (Sept.), 943–959.

HAUSWIRTH, M. AND CHILIMBI, T. 2004. Low-overhead memory leak detection using adaptive statistical profiling. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

HEJLSBERG, A., WILAMUTH, S., AND GOLDE, P. 2003. *The C# Programming Language*. Addison-Wesley, Reading, MA.

HIBERNATE. 2004. Object/relational mapping and transparent object persistence for Java and SQL databases. In http://www.hibernate.org/.

HOPCROFT, J. E., MOTWANI, R., AND ULLMAN, J. D. 2000. *Introduction to Automata Theory, Languages, and Computation (2nd Edition)*. Addison-Wesley.

HOVEMEYER, D. AND PUGH, W. 2004. Finding bugs is easy. In *OOPSLA '04: Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and applications*. ACM, New York. 132–136.

KILDALL, G. A. 1973. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. ACM, New York. 194–206.

KORTH, H. F., LEVY, E., AND SILBERSCHATZ, A. 1990. A formal approach to recovery by compensating transactions. *VLDB J*. 95–106.

LIBLIT, B., AIKEN, A., ZHENG, A. X., AND JORDAN, M. I. 2003. Bug isolation via remote program sampling. In *Programming Language Design and Implementation* (San Diego, CA).

LINDHOLM, T. AND YELLIN, F. 1997. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Reading, MA.

LISKOV, B. AND SCHEIFLER, R. 1983. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Trans. Prog. Lang. Syst. 5*, 3 (July), 381–404.

LIU, C., ORLOWSKA, M. E., LIN, X., AND ZHOU, X. 2001. Improving backward recovery in workflow systems. In *Proceedings of the Conference on Database Systems for Advanced Applications*.

LOWELL, D. E., CHANDRA, S., AND CHEN, P. M. 2000. Exploring failure transparency and the limits of generic recovery. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*.

LOWELL, D. E. AND CHEN, P. M. 1998. Discount checking: transparent, low-overhead recovery for general applications. Tech. Rep. CSE-TR-410-99, University of Michigan. Nov.

MALAYERI, D. AND ALDRICH, J. 2006. Practical exception specifications. In *Advanced Topics in Exception Handling Techniques*, C. Dony, J. L. Knudsen, A. B. Romanovsky, and A. Tripathi, Eds. Lecture Notes in Computer Science, vol. 4119. Springer-Verlag, New York. 200–220.

MILLER, R. AND TRIPATHI, A. 1997. Issues with exception handling in object-oriented systems. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)*. 85–103.

NECULA, G. C., MCPEAK, S., RAHUL, S. P., AND WEIMER, W. 2002. Cil: An infrastructure for C program analysis and transformation. In *Proceedings of the International Conference on Compiler Construction*. 213–228.

NECULA, G. C., MCPEAK, S., AND WEIMER, W. 2002. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the ACM Symposium on Principles of Programming Languages*. ACM, New York. 128–139.

ODERSKY, M. AND WADLER, P. 1997. Pizza into Java: Translating theory into practice. In *Proceedings of the ACM Symposium on Principles of Programming Languages*. ACM, New York. 146–159.

O'HANLEY, J. 2005. Always close streams. In http://www.javapractices.com/.

PERRY, E. H., SANKO, M., WRIGHT, B., AND PFAEFFLE, T. 2002. Oracle9i JDBC developer's guide and reference. Tech. Rep. A96654-01 (Release 2 (9.2)), http://www.oracle.com. Mar.

REIMER, D., SCHONBERG, E., SRINIVAS, K., SRINIVASAN, H., ALPERN, B., JOHNSON, R. D., KERSHENBAUM, A., AND KOVED, L. 2004. Saber: Smart analysis based error reduction. *SIGSOFT Softw. Eng. Notes 29*, 4, 243–251.

REPS, T., HORWITZ, S., AND SAGIV, M. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, CA). ACM, New York. 49–61.

ROBILLARD, M. P. AND MURPHY, G. C. 2003. Static analysis to support the evolution of exception structure in object-oriented systems. *ACM Trans. Softw. Eng. Methodol. 12*, 2, 191–221.

RYDER, B. G., SMITH, D., KREMER, U., GORDON, M., AND SHAH, N. 2000. A static study of Java exceptions using jesp. In *CC '00: Proceedings of the 9th International Conference on Compiler Construction* (London, UK). Springer-Verlag, New York. 67–81.

SAVAGE, S., BURROWS, M., NELSON, G., SOBALVARRO, P., AND ANDERSON, T. 1997. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst. 15*, 4, 391–411.

SCHMUCK, F. AND WYLLIE, J. 1991. Experience with transactions in QuickSilver. In *Proceedings of the 13th ACM SIGOPS Symposium on Operating Systems Principles*. ACM, New York. 239–253.

SELTZER, M. I., ENDO, Y., SMALL, C., AND SMITH, K. A. 1996. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the Symposium on Operating Systems Design and Implementation* (Seattle, WA). 213–227.

SHAPIRO, J. S., SMITH, J. M., AND FARBER, D. J. 1999. EROS: A fast capability system. In *Proceedings of the Symposium on Operating Systems Principles*. 170–185.

SINHA, S. AND HARROLD, M. J. 1999. Criteria for testing exception-handling constructs in Java programs. In *Proceedings of the International Conference on Software Maintenance* (ICSM'99) (Oxford, England, UK, August 30–September 3). IEEE Computer Society, Online publication: http://computer.org/proceedings/icsm/0016/0016toc.htm, 265–276.

Sinha, S. and Harrold, M. J. 2000. Analysis and testing of programs with exception handling constructs. *IEEE Trans. Softw. Eng. 26*, 9, 849–871.

Sinha, S., Orso, A., and Harrold, M. J. 2004. Automated support for development, maintenance, and testing in the presence of implicit control flow. In *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)* (St. Louis, MO, May 15–21). ACM, New York. 336–345.

SourceForge.net. 2003. About SourceForge.net (document A1). http://sourceforge.net. Tech. rep.

Stallman, R., Pesch, R., and Shebs, S. 2002. *Debugging with GDB*. Free Software Foundation.

Stroustrup, B. 1991. *The* C++ *Programming Language (second edition)*. Addison-Wesley, Reading, MA.

Sun Microsystems. 2001. Java pet store 1.1.2 blueprint application. http://java.sun.com/blueprints/code/. Tech. rep.

Tofte, M. and Talpin, J.-P. 1997. Region-based memory management. *Inf. Comput.*

Valetto, G. and Kaiser, G. 2002. A case study in software adaptation. In *Proceedings of the ACM Workshop on Self-Healing Systems (WOSS '02)*. 73–78.

van der Wal, S. 2002. Creating the C++ auto_ptr<> utility for Symbian OS. Tech. rep., http://www.symbian.com/developer/techlib/. Aug.

Wagner, D., Foster, J. S., Brewer, E. A., and Aiken, A. 2000. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of the Networking and Distributed System Security Symposium 2000* (San Diego, CA).

Weimer, W. and Necula, G. C. 2004. Finding and preventing run-time error handling mistakes. In *OOPSLA '04: Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, New York. 419–431.

Weimer, W. and Necula, G. C. 2005. Mining temporal specifications for error detection. Lecture Notes in Computer Science, vol. 3440. Springer-Verlag, New York. 461–476.

Received December 2005; revised May 2007 and August 2007; accepted August 2007.