Exceptional Situations And Program Reliability

by

Westley R. Weimer

B.A. (Cornell University) 1999
M.S. (University of California, Berkeley) 2003

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:
Professor George C. Necula, Chair
Professor Rastislav Bodík
Professor Leo Harrington

Fall 2005

Exceptional Situations And Program Reliability

Abstract

Exceptional Situations And Program Reliability

by

Westley R. Weimer

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor George C. Necula, Chair

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 dataflow analysis for finding a certain class of mistakes made during exceptional situations. We also present a specification miner for automatically inferring partial notions of what programs should be doing. Finally, we propose and evaluate a new language feature, the compensation stack, to make it easier to write solid code in the presence of run-time errors.

We give a dataflow analysis for finding a certain class of exception-handling mistakes: those that arise from 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 800 exception handling mistakes in almost 4 million lines of Java code. The analysis is unsound and produces false positives, but a few simple filtering rules suffice to remove them in practice. The remaining mistakes were manually verified. These mistakes cause sockets, files and database handles to be leaked along some paths.

Specifications are necessary in order to find software bugs using program verification tools. We give a novel automatic specification mining algorithm that uses information about exception handling to learn temporal safety rules. Our algorithm is based on the observation that programs often make mistakes along exceptional control-flow paths, even when they behave correctly on normal execution paths. We show that this focus improves the miner's effectiveness at discovering specifications beneficial for bug finding. We present quantitative results comparing our technique to four existing miners. We highlight assumptions made by various miners that are not always borne out in practice. Additionally, we apply our algorithm to existing Java programs and analyze its ability to learn specifications that find bugs in those programs. In our experiments, we find filtering candidate specifications to be more important than ranking them. We find 430 bugs in 1 million lines of code. Notably, we find 250 more bugs using per-program specifications learned by our algorithm than with generic specifications that apply to all programs.

We present a characterization of the most common causes of those bugs and discuss the limitations of exception handling, finalizers and destructors. Based on that characterization we propose a programming language feature, the compensation stack, that keeps track of obligations at run time and ensures that they are discharged. Finally, we present

case studies to demonstrate that this feature is natural, efficient, and can improve reliability; for example, retrofitting a 34,000-line program with compensation stacks resulted in a 0.5% code size decrease, a surprising 17% speed increase (from correctly deallocating resources in the presence of exceptions), and more consistent behavior.

Professor George C. Necula
Dissertation Committee Chair

Dedication

A lungo il mio cuore di tali ricordi ha voluto colmarsi!

Come un vaso in cui le rose sono state dissetate:

Puoi romperlo, puoi distruggere il vaso se lo vuoi,

Ma il porfumo delle rose sarà sempre tutt'intorno.

Lang, lang soll die Erinnerung in meinem Herzen klingen!

Gleich einer Vase, drin Rosen sich einst tränkten:

Lass sie zerbrechen, lass sie zerspringen,

Der Duft der Rose bleibt immer hängen.

Mon coeur est brûlant rempli de tels souvenirs

Comme un vase dans lequel des roses ont été distillées:

Tu peux le briser, tu peux détruire le vase si tu le désires,

Mais la senteur des roses sera toujours là.

Muito, muito tempo seja meu coração preenchido com tais lembranças!

Tal qual o vaso onde rosas foram uma vez destiladas:

Pode quebrar, pode estilhaçar o vaso se o desejas,

Mas perdurará para sempre o aroma das rosas perfumadas.

Długo, długo moje serce przepełnione było takimi wspomnieniami!

Były jak waza, w której kiedyś róże destylowały:

Możesz sprawić by pekła, możesz gruchotać waze jeśli chcesz,

Ale zapach róż bedzie wciaż czuć dookoła.

Contents

List of Figures

Acknowledgments

I thank my advisor George Ciprian Necula. George is not only a source of good ideas but also a destroyer of bad ones. Many of my less-tenable schemes were put forever to rest on his whiteboard. George and his family (Simona, Deanna and Sylvia) made my life as a grad student that much brighter with dinners, sailing trips and get-togethers.

On the academic side, I thank Ras Bodik, Glenn Ammons and Dave Mandelin for insightful discussions and for helping me to experiment on their `Strauss` tool. I thank Dawson Engler for enlightening discussions about his technique, z-ranking, and expected results. I thank John Whaley for providing me with the `joeq` source code and pointers for running his miner. I thank Rajeev Alur for giving me a number of examples of `JIST` in action.

I thank the Berkeley/Stanford Recovery-Oriented Computing Project and the Berkeley Center for Hybrid and Embedded Software Systems. I found attending retreats and speaking with people in those projects to be invaluable. I thank Aaron Brown for providing an explanation of and workload for his `undo` program, as well as for fruitful discussion about Java error handling. I thank Mark Brody, George Candea and Tom Martell for generously providing their infrastructure and their workload generator for `Pet Store`. I thank Christopher Hylands Brooks for an insightful discussion of error handling in general and `ptolemy2` in particular. I thank William Kahan for discussions of floating-point exception handling and comments on this document. Scott McPeak was also kind enough to point out mistakes in this document.

Finally, I would like to thank a number of people for enlightening non-technical (i.e., friendly) discussions while I was a graduate student: Evan Chang, Jason Compton, Jeremy Condit, Simon Goldsmith, Sumit Gulwani, Matt Harren, Ranjit Jhala, Iain Keddie, David Liben-Nowell, Scott McPeak, Ana Ramírez Chang, Shivani Saxena, Andrew Shum, Tachio Terauchi, Kiri Wagstaff, and Donna Weimer.

How glorious it is—and also how painful—to be an exception.

Louis Charles Alfred de Musset
French writer (1810-1857)

Chapter 1

Introduction

Software is increasingly important but much of it remains unreliable. It is much easier to fix software defects if they are found before the software is deployed. It is difficult to use testing, the traditional approach to finding defects early, to evaluate programs in exceptional situations. We present an analysis for finding a class of program mistakes related to such exception situations. We also present an algorithm for inferring what the program should be doing in those circumstances. Finally, we propose a new language feature, compensation stacks, to make it easier to fix such mistakes.

1.1 The Cost of Software Reliability

The NIST calculated the 2002 U.S. annual economic cost of software errors to be \$59.5 billion (or 0.6 percent of the gross domestic product). The report claims that more than a third of that cost could be eliminated by enabling “earlier and more effective identification and removal of software defects.” [NIS02]

Once a piece of software has been shipped or deployed it can be from two to thirty times more expensive to fix a bug. Those figures are somewhat conservative and some sources suggest that a factor of one hundred is more reasonable. For example, in one company surveyed by Rex Black, the "response to field failures was to fly a programmer to the client's site, along with sufficient tools to fix the bug, and keep him there until the problem was fixed, which was typically about a week. Last-minute airfare, hotel costs, meals, and car rental added about $2,000 to the $4,000 cost associated with the programmer's lost time." [Bla02] An internal source who asked not to be named suggested that the general cost for a software defect averaged over IBM's software division was $10,000. Thus a compelling case can be made for the importance of finding defects early.

1.2 Testing

Testing is the traditional approach to finding software defects before the software is deployed. Testing typically involves running the program on a predetermined workload or test case and evaluating the result. The result may be compared against a reference that is known to be correct or it may merely be inspected to show the absence of some catastrophic failure. A bad result usually means that the testing has found a bug.

Testing is very popular. An oft-quoted rule of thumb is that at least fifty percent of a commercial software project's budget is devoted to testing. Unfortunately, finding indicative test cases is difficult. Selecting good test cases a priori has been compared to baby-proofing a house in preparation for a child's arrival. Invariably the child will find some way to get in to trouble that the parents failed to forsee.

Errors involving exceptional situations are particularly difficult to catch with conventional testing. The complete input to a program consists not just of the values entered by the user or found in files but also of the state of the local machine and other "environmental" concerns. Typically a test case only specifies the values that would be entered by the user or found in such files. For example, a program may have a bug that only surfaces when the local disk is full. No simple test case on an expensive testing server with plenty of free space will reveal such a bug. However, end users with more modest machines may legitimately run out of space and encounter the defect. Similarly, networked programs that depend on local environmental factors like congestion and reachability are notoriously difficult to test in advance.

Testing programs that are intended to run for a long time is also difficult. Resource leaks and other API violations that usually do not matter in a program that is started and terminated within a few seconds can bring down a longer-running program over time. For example, a program that leaks a megabyte of virtual address space every thirty minutes will still take around three months to exhaust the address range of a 32-bit machine. Until such a program finally runs out of resources it will typically respond normally and correctly to requests. A software developer can rarely spare the testing resources to keep a fixed version of a program that is running for such a long time. Many companies producing highly-available server software have taken a "live with leaks" attitude and make special provisions to reboot their machines (and thus start with a blank slate of unleaked resources) every twenty-four hours.

Gradual resource leaks in long-running programs can often be seen as a special kind of failure in handling unexpected situations. Typically, if a server answers multiple requests per second and leaks resources on most requests the leak will be noticed rapidly during testing. If, however, the server only leaks resources when processing certain rare requests (e.g., requests from users with network connectivity problems or requests involving items in a high-contention portion of the inventory database) the leak will usually escape immediate detection. The occasional requests that trigger a leak can be viewed as an exceptional situation.

1.3 Exceptional Situations

In this context an *exceptional situation* is one in which something external to the program behaves in an uncommon but legitimate manner. For example, a request to write a file may legitimately fail because the disk is full or because the underlying operating system is out of file handle resources. Similarly, a request to send a packet reliably may fail because of a network breakdown between the source and the destination. A request to commit a database transaction may fail because of opportunistic concurrency control or other locking considerations. A request to allocate memory may fail because the operating system or virtual machine is out of memory. All of the above examples represent actions that typically succeed but may occasionally fail through no fault of the requesting program.

Testing a program's behavior in exceptional situations is difficult. Exceptional situations, often called *faults* or *run-time errors*, must be systematically and artificially introduced while the program is executing. The program and its intended context help to determine a fault model, which governs the appropriate kind and number of faults. For example, a program may be expected to degrade gracefully if 10% of network send requests fail but may not be expected to make forward progress if all network send requests fail. A text editor may only care about recovering from user interface or file system faults while a robust database may be expected to be ironclad in almost all circumstances. Finding the right fault model is important.

Once the fault model has been established the 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 error) [CDCF03]. Others have used special program analyses and compiler instrumentation approaches [FRMW04] to inject faults at the software or virtual machine level. These approaches are still based on testing, however, and require indicative workloads and test cases.

1.4 Toward Reliability in Exceptional Situations

We theorized that difficulties in testing code under exceptional situations and in understanding fault models would mean that many programs had latent bugs related to their handling of such exceptional situations.

Our approach to improving software reliability involves fixing defects and facilitating the writing of defect-free code. In order to make it easier to write or rewrite such code we must characterize why the mistakes are being made. Given such an understanding we can propose features or analyses that handle or verify the complicated and error-prone portions of the process. In order to apply such technology to existing code we must automatically find existing defects. Finding a defect related to an exceptional situation involves formalizing both what can legitimately go wrong and what the program should have been doing. The former is the fault model, the later is typically called a specification. While some specifications are universal, most are program-specific. Thus we must be able to determine specifications for a program by analyzing that particular program. If we can determine that the program fails to do the right thing during a legitimate situation (i.e., that it violates the specification with respect to the fault model) we have found a defect.

We will discuss a number of analyses and techniques to achieve those goals. In Chapter 2, we present a static dataflow analysis for locating places where a program violated a safety policy with respect to a fault model. In Chapter 3 we present a specification mining algorithm for automatically inferring candidate specifications from programs. In Chapter 4 we propose new programming language features that make it easy to fix the class of defects discovered by our analyses. For all of these we present empirical results to support our claims.

Putting it all together, we have a multi-step process for addressing software reliability concerns related to exceptional situations. Given an existing program we apply an analysis to the program, our fault model and some generic specifications. Given those three components the analysis yields potential defects. In addition, we analyze the program in order to determine locally-important specifications and use those to find potential defects. Once the defects have been located we provide tools and language features for easily removing the defects.

Beyond the primary goal of improving software reliability we have a number of secondary goals. It is often said that program analyses can either prove big things about small programs or prove small things about big programs. We believe that any analysis we develop should scale to large, real-world programs (i.e., should work on millions of lines of code rather than just toy examples) and we are willing to sacrifice precision in a controlled manner in order to achieve that goal. We also want any tools or techniques we propose to be easy to use, especially in terms of the time or effort it takes in order to see an improvement. Programmers certainly make cost-benefit comparisons when evaluating potential tools, but we have found that a notion of “activation energy” is also important: if it takes too long to get any benefit, even a large benefit, the tool will be discarded. Thus we aim to avoid making programmers sift through hundreds of lines of output in order to find a single useful piece of information. In addition, we do not want to require that programmers annotate their code or otherwise spend time making it ready for our techniques. Ideally we should be able to consider a new program and find real defects in it without requiring the programmer to sift through the results or guide the process.

In the `finally`, you protect yourself against the exceptions, but you don't actually handle them. Error handling you put somewhere else. ... But you make sure you protect yourself all the way out by deallocating any resources you've grabbed, and so forth. You clean up after yourself, so you're always in a consistent state.

Anders Hejlsberg, Lead C# Architect

Chapter 2

Finding Defects

This chapter builds up to a static dataflow analysis that can locate software errors in a program with respect to a fault model and a specification of correct behavior. The analysis examines each method in turn and keeps track of resources governed by the safety specification along all paths, but especially along paths related to the exceptional situations allowed by the fault model. We provide one such fault model and three such specifications based on manual inspection of a large code base. A simpler form of the analysis presented here was previously discussed in an earlier work [WN04].

2.1 Handling Exceptional Situations At The Language Level

Modern languages like Java [GJS96], C++ [Str91] and C# [HWG03] use a language-level featured called *exceptions* to facilitate signaling and handling exceptional situations. The most common semantic framework for exceptions is the *replacement model* [Goo75]. The program or an underlying library will *signal* or *raise* an exception and interrupt the normal flow of control in order to indicate the presence of an exceptional situation. In the replacement model the result of a computation that is interrupted by an exception is replaced by the result of evaluating the nearest enclosing appropriate *exception handler*. An exception handler is conceptually similar to a subroutine and may itself signal or handle exceptions.

Exception handlers are typically lexically scoped. In Java the syntax for an basic exception handler is the `try-catch` block:

```
try {
  boo();
} catch (Exception exc) {
  minsc();
}
```

If `boo()` terminates normally the `catch` block is never executed. If `boo()` signals an exception, `minsc()` is executed with the variable `exc` containing information about the particular exception (e.g., what caused it). Within a particular context a signaled exception that has no handler is called an *uncaught exception*. If `minsc()` signals an exception control passes to the nearest enclosing exception handler:

```
try {
  try {
    boo();
  } catch (Exception exc1) {
    minsc();
  }
} catch (Exception exc2) {
  imoen();
}
```

In this example, if `boo()` raises an exception it is handled by `minsc()` as `exc1`. If `minsc()` raises an exception it is handled by `imoen()` as `exc2`. The two exceptions need not be directly related. For example, if `boo()` is related to a networked e-commerce application, `exc1` might be a network timeout. The exception handler `minsc()` might take that information and attempt to write a log record to the disk. In the process `boo()` might discover that the disk is full and be unable to proceed. The second handler `imoen()` deals with the full-disk scenario in some other manner (e.g., by displaying a message on the console or by trying to free up space).

Languages that support `try-catch` exception handling almost invariably also support a mechanism for executing important code in all cases. The Java syntax for this feature is the `finally` block:

```
try {
  boo();
} catch (Exception exc) {
  minsc();
} finally {
  edwin();
}
```

In this example if `boo()` terminates normally, `edwin()` is executed. On the other hand, if `boo()` raises an exception then `minsc()` is executed and then `edwin()` is executed. There are two important corner cases to consider. First, if `minsc()` raises an exception, `edwin()` is still executed. Second, if `boo()` raises an exception *and* `edwin()` raises an exception, the exception from `edwin()` will be propagated to the nearest enclosing handler.

Lexical nesting allows exception handlers to become quite labyrinthine. Complicated exception handling is difficult for programmers to reason about and to code correctly. As a result, it will prove to be a source of software defects related to reliability. In particular, programs tend to make mistakes when attempting to handle multiple cascading exceptions.

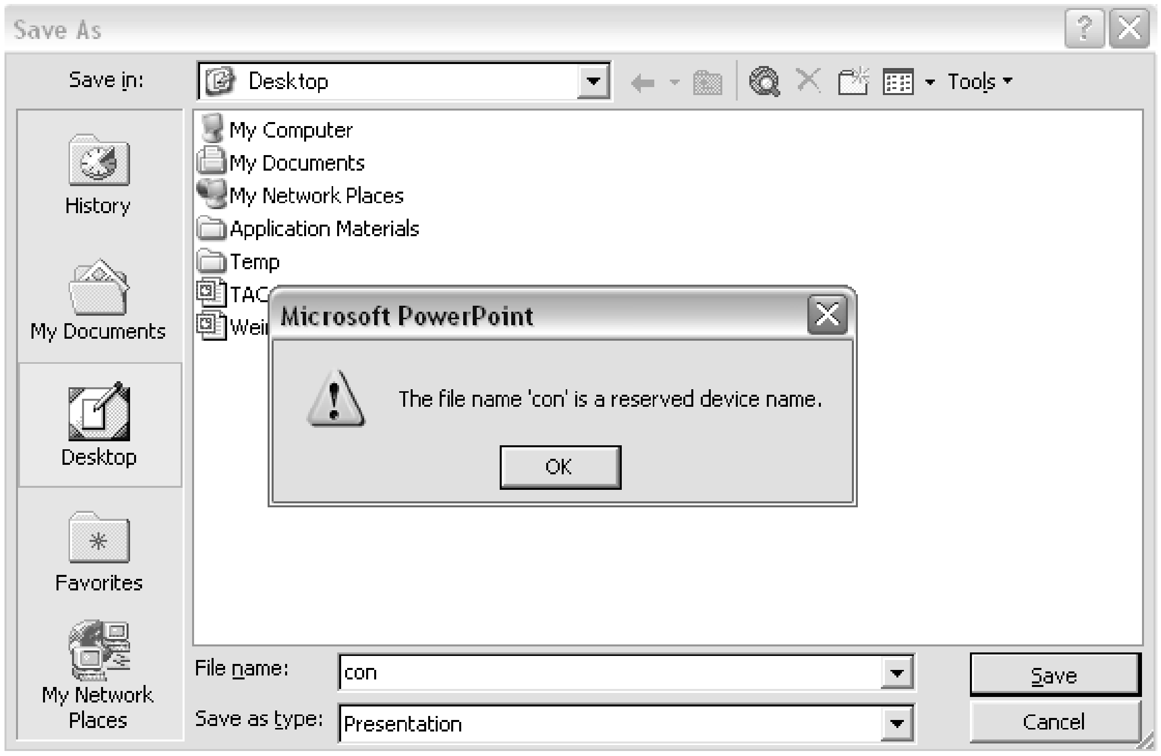
2.2 Handling Exceptional Situations In Practice

An IBM survey [Cri87] reported that up to two-thirds of a program may be devoted to error handling and exceptional situations. We were initially skeptical and performed a similar survey on more modern programs. We examined a suite of open-source Java programs ranging in size from 4,000 to 1,600,000 lines of code and 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 experiments 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 errors without using exceptions and to use exceptions for purposes other than error handling, common Java programming practice links the two together.

Aside from programs specifically designed from the ground up for reliability (e.g., Brown's database-like `undo` [BP03]), these proportions grow with program size and age. That is, smaller and younger programs have less code devoted to exception handling. These broad numbers suggest that error handling is an important part of modern programs and that much effort is devoted to it.

Despite the importance of handling exceptional situations and the programmer effort devoted to it, we will demonstrate that poor handling abounds. In order to claim that a program is making a mistake, however, we must first specify what it should be doing.



The file name 'con' is a reserved device name.

Figure 2.1: Microsoft PowerPoint exception-handling dialog box.

2.3 Proper Exception Handling

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. A file-writing program may handle a storage exception by asking the user to specify an alternate destination for the data. A security-conscious program may respond to an access violation exception by attempting to acquire additional credentials.

Figure 2.1 shows a dialog box displayed by Microsoft PowerPoint when the user attempts to save a file using the name `con`. For legacy reasons the name `con` refers to the console device and is not a valid filename for user data under Microsoft Windows. A GUI program that attempts to write to a file named `con` may receive an error from the operating system (e.g., via the `open(2)` or `write(2)` system calls). In modern languages like Java the interface with the operating system is handled by an abstraction layer that looks for errors reported by the operating system and signals exceptions when they occur. In this particular example Microsoft PowerPoint displays a warning dialog box that implicitly asks the user to choose another filename. Other handling options were available. For example, it could also have automatically renamed the file by appending "`.ppt`".

We will not consider high-level policy notions of correctness like whether the desired program behavior is to display a dialog box or to rename the file. Similarly, we will not consider the particular details of the actions performed by the exception handler (e.g., whether the message is spelled "file name" or "filename" or whether there are two buttons or one on the dialog box). Such specifications of proper exception handling behavior are too high-level for our purposes.

Instead, we will consider more generic low-level notions of correctness. To continue our example, regardless of whether PowerPoint displays a dialog box or renames the file it should not crash. In addition, it should not lose the user's work or prevent the user from saving that work somewhere else. Faulty exception handling, however, could result in just that scenario.

Common exception handling mistakes could easily cause PowerPoint to be unable to save further files. In modern operating systems, programs like PowerPoint are only allowed to access a limited number of files at once. When a file is opened the operating system returns a special file “handle” associated with it to the program. Each program has a maximum number of outstanding file handles and must eventually return them to the operating system before opening more. Normally programs like PowerPoint open a file, save the data, and then close the file. In the case of the `con` file, however, the program may well acquire a file handle associated with the file name `con` (it is legal to open the console device file) but will be unable to save the data. The exception handler can display a dialog box or rename the file and try again, but in all cases it should close the file handle associated with `con`. If it forgets to do so PowerPoint will eventually “run out” of file handles and will be unable to open any new files (e.g., to save the user’s work later).

While this particular example may be contrived (e.g., it is unlikely that an interactive user will just happen to pick a long string of reserved file names: `con`, `aux`, `prn`, `nul`, etc.) it encapsulates all of the concepts in a large class of exception handling mistakes. First, the program has some important resources (in this case, file handles) that are involved in operations that may legitimately fail in exceptional situations. Those important resources must be treated correctly (in this case, must be closed and returned to the operating system). Regardless of any application-specific logic, the program should treat those resources correctly even in exceptional situations. A short interactive session with PowerPoint can tolerate a few leaked file handles but a webserver answering hundreds of requests per second that mishandles an important resource whenever, for example, the webpage `con` is requested (or the username `con` is used, etc.) will quickly crash.

The next section describes how exception handling looks from the programmer’s perspective.

```
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: }
```

Figure 2.2: Ohioedge CRM Exception Handling Code

2.4 Exception Handling Example

We begin with a motivating example that shows how the mistakes described in Section 2.3 can occur in practice. Consider the code in Figure 2.2, taken from Ohioedge CRM, the largest open-source customer relations management project [Sou03]. This program uses language features to facilitate exception handling (i.e., nested `try` blocks and `finally` clauses), but many problems remain. `Connection`s, `PreparedStatement`s and `ResultSet`s represent important global resources associated with an external database. Our specification of correct behavior, which we will formalize later, is that the program should `close` each one as quickly as possible.

In some situations the exception handling in Figure 2.2 works correctly. If a run-time error occurs on line 6, the runtime system will signal an exception, and the program

```
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: }
```

Figure 2.3: Revised Ohioedge CRM Exception Handling Code

will close the open `Connection` on line 14. However, if a run-time error occurs on line 8 (or 9 or 10), the resources associated with `ps` and `rs` may not be freed.

One common solution is to move the `close` calls from line lines 10 and 11 into the `finally` block, as shown in Figure 2.3. 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 a failure while closing `rs` on line 12 might leave `ps` dangling.

Failures while closing files most commonly occur in the presence of network filesystems. In order to reduce write latency and network congestion, network file system clients often locally buffer writes to remote files. Buffered data is sent out either when there is enough of it to making sending a packet worthwhile or when the file is closed or flushed. If the network connectivity between the client and the file server degrades while the data is buffered, closing the file will cause the client to try and fail to send the buffered data. This will be reported to the user as an exception raised by `close`. Ideal but rarely seen responses to such a situation including trying to close the file again later, warning the user, and saving the data to a local file instead.

In this particular example the important resources are database objects (e.g., `ResultSet`s) and not operating system file handles. The conceptual model is similar, however. A `ResultSet` is obtained by executing a database transaction on a remote database, an action that is similar to doing a filesystem read or write. Problems with either the database itself (e.g., the database may be using opportunistic concurrency control and may thus notice transaction problems "late") or the connectivity with the database may show up when the `ResultSet` is closed. Such problems are rare but do occur in practice.

The code in Figure 2.3 may also attempt to `close` an object that has never been created. If an error occurs on line 6 after `cn` has been created but before `rs` 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 with the situation. The first, shown in Figure 2.4, involves using nested `try-finally` blocks. One block is required for each important resource that is dealt with simultaneously. After each resource is acquired a `try-finally` block is immediately started, ensuring that if something goes wrong with the code's normally processing the resource will still be dealt with. Since each `finally` block is limited to a handling a single resource an error in one `close` will not

```
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: }
```

Figure 2.4: Nested Try-Finally Ohioedge CRM Exception Handling Code

cause other `close`s to be skipped. Programmers tend not to like this approach because it has a number of software engineering disadvantages. For example, programs commonly use three to five important simultaneous resources but programmers are rarely willing to use three to five nested `try-finally` blocks.

The second standard approach is to use special sentinel values or run-time checks to ensure that the resources are handled properly. In Figure 2.5 the database objects are explicitly initialized to the special sentinel value `null`. If an object is successfully created or allocated it will no longer have the value `null`. In the single `finally` block each object is checked against `null`. If the object is not `null` then it must have been successfully acquired so an attempt is made to `close` it. In this example any exceptions signaled by `close` are ignored on lines 11-13. This approach has the advantage that one `try-finally` statement

```
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) { }
}
```

Figure 2.5: Run-Time Check Ohioedge CRM Exception Handling Code

can handle any number of simultaneous resources. Unfortunately it is often difficult for humans to write such bookkeeping code correctly.

The code extracted from the Ohioedge CRM is quite 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 the general possibility of exceptional situations: language-level exception handling (e.g. `try` and `finally`) are 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 make the same mistakes. Developers have cited this required repetition to explain why exception handling is sometimes ignored [BP03]. 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 keeping track of progress in a counter variable). Handling such problems is complicated and error-prone in practice.

In the next sections we will discuss an analysis for automatically discovering such exception-handling mistakes. For example, the analysis will report three paths in Figure 2.2. If an exception occurs on line 8, a `PreparedStatement` is leaked. If an exception occurs on line 9, both the `PreparedStatement` and the `ResultSet` are forgotten. Finally, if the first call to `close` on line 10 raises an exception, the `PreparedStatement` is again leaked. First, however, we formalize our notions of what the program should be doing and what may legitimately go wrong.

2.5 Specifications

We will use finite state machines to specify how programs should manipulate certain important resources and interfaces. The edge labels in such a finite state machine represent important events that take place during the program's execution relating to those resources. For example, one event may represent the creation of a resource or the closing of a resource. We associate one finite state machine specification with every dynamic instance of such a resource. Each resource is tracked separately. A program may thus have two file handle resources, for example, only one of which is currently open. Each finite state machine starts in its start state. At the end of the program each finite state machine must be

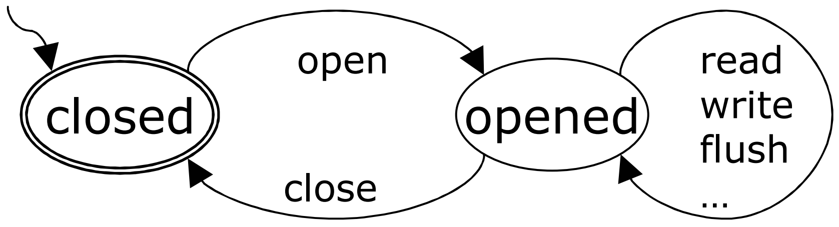


Figure 2.6: A simple finite state machine specification for File resources.

in an accepting state or the program is said to violate the specification with respect to that resource instance. In addition, if the finite state machine ever makes an illegal transition the program is said to violate the specification.

Figure 2.6 shows a simple finite state machine safety specification for handling File resources. An uninitialized File starts out in the closed state. From there the only valid operation is to open the File which transitions the specification to the opened state. The File can then be used (e.g. via read and write). In order to comply with the specification the File must be returned to the closed state via the close event. It is also illegal to attempt to read or write from a File when it is not in the opened state.

In order to demonstrate that existing Java programs violate safety policies while handling exceptional situations we need to start with policies that those programs are trying to enforce. In our survey of open-source Java programs we manually examined all `catch`, `finally` and `finalize` blocks in order to find what policies were important in common programs. Many of those policies were program-specific but three dealt with generic "system library" resources that were shared across most programs. For simplicity we did not consider events like write in Figure 2.6 that do not change the state of the resource.

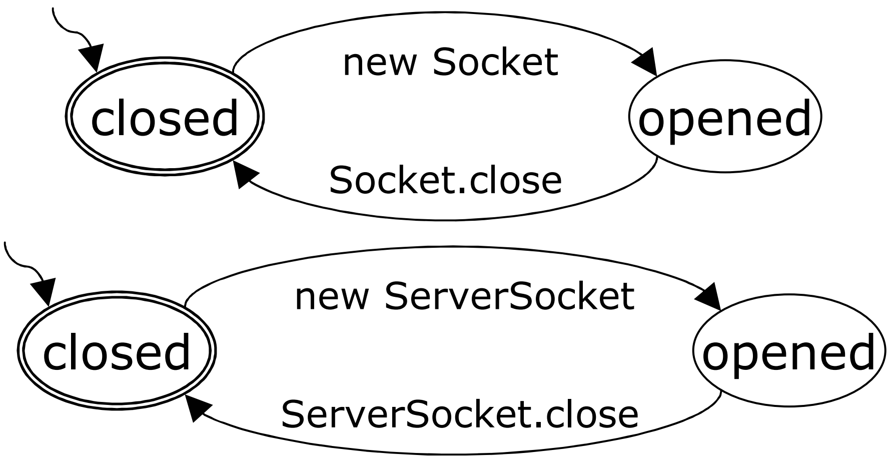


Figure 2.7: A manually-derived specification for Java `Socket` resources.

Figure 2.7 shows a safety specification for Java `Socket` objects. `Socket`s are used in UDP and TCP network communication. Both `Socket`s and `ServerSocket`s are based on file handles and should be freed. In addition, all `Socket`s on the same machine (or network interface) share the same limited port number address space. While all incoming connections to an HTTP server may attempt to connect to a `ServerSocket` listening on port 80, the operating system will assign every individual connection a port between 1024–5000 or 49152–65535 to distinguish between connections from different clients. The Java Tutorial explains that, "upon acceptance, the server gets a new socket bound to a different port. It needs a new socket (and consequently a different port number) so that it can continue to listen to the original socket for connection requests while tending to the needs of the connected client." [CWH00] For simplicity we choose not to model the `accept` method which can create `Socket`s from a `ServerSocket`. An analysis using this specification to find software errors may thus miss potential errors.

We formally represent a specification finite state machine using the standard five-

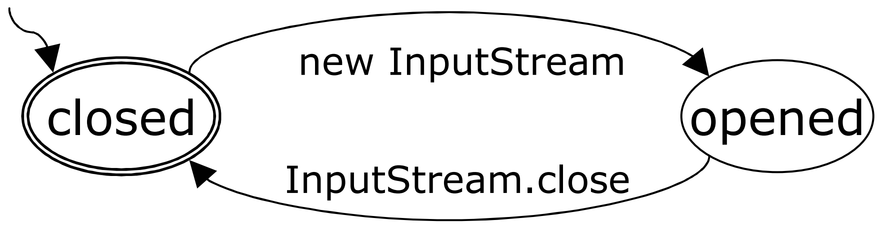


Figure 2.8: A manually-derived specification schema for `InputStream` resources.

tuple $\langle \Sigma, S, s_0, \delta, F \rangle$. [HMU00] The first specification in Figure 2.7 would be given as:

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

The set of events (or the input alphabet) is Σ, the set of states is S, the initial state is s_0, the set of final (or accepting) states is F. The transition function δ maps states and events to new states and represents the "edges" of the state machine. The specifications we consider are often equivalent up to state and event renaming. While it is possible to consider more complicated specifications we initially chose to concentrate on important, simple specifications for which violations are easy to understand, easy to fix and important to fix.

Figure 2.8 shows a safety specification schema for objects that derive from and extend the class `java.io.InputStream`. Java library classes such as `PushbackInputStream` and `ObjectInputStream` as well as user-defined classes build upon the standard `InputStream` interface. We instantiate an instance of this safety specification schema for every class that derives from `java.io.InputStream` in the program under consideration. `Stream`s are typically based on file handles and "streams represent resources which you must always clean up explicitly, by calling the close method." [O'H05] We could construct a similar specification schema for `OutputStream`s but do not do so for simplicity.

Figure 2.9 shows a safety specification for resources associated with the Java Database Connection (`JDBC`) interface. Typically (as in Figure 2.2) a `Connection` is established with an external SQL database. A SQL query is represented as a `Statement` or `PreparedStatement` object. The result of executing the query on the remote database is represented as the `ResultSet`. It is typically considered good practice to close all of these objects: "Always close Statements, PreparedStatements, and connections: This practice involves always closing JDBC objects in a `finally` block to avoid resource limitations found in many databases." [Ash04] The Oracle9*i* JDBC Developer's Guide and Reference makes the results of failing to do so explicit: "You must explicitly close the `ResultSet` and `Statement` objects after you finish using them. This applies to all `ResultSet` and `Statement` objects you create when using the Oracle JDBC drivers. The drivers do not have finalizer methods; cleanup routines are performed by the `close()` method of the `ResultSet` and `Statement` classes. 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. Closing a result set or statement releases the corresponding cursor in the database." [PSWP02] 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. The specification in Figure 2.9 covers multiple

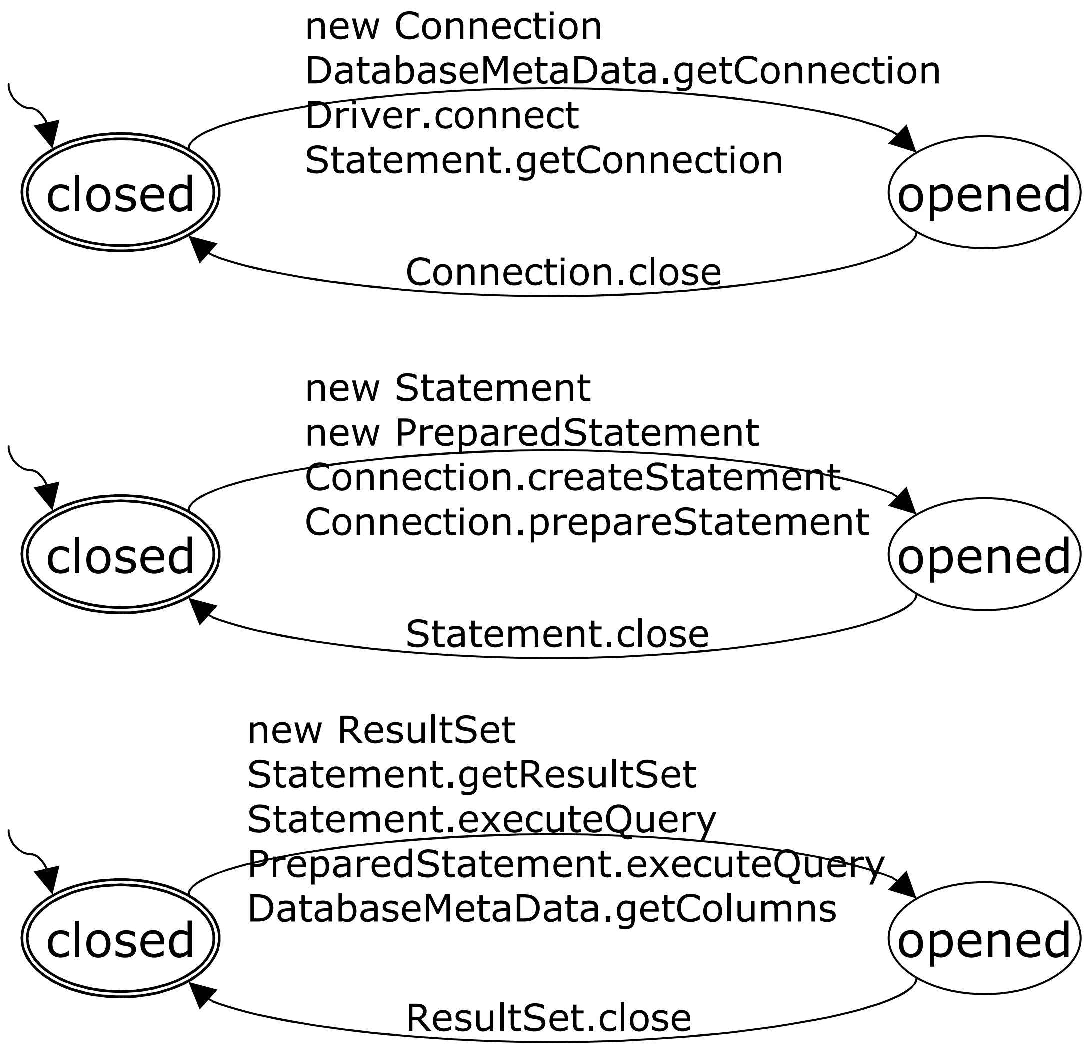


Figure 2.9: A manually-derived specification for resources associated with `JDBC` database connections.

types of objects, each of which can be obtained or allocated multiple ways.

The specifications in Figure 2.7, Figure 2.8 and Figure 2.9 were chosen in part because programmers respect them. Our goal is to improve software quality by finding and fixing bugs. If we find “bugs” with respect to a specification that is widely ignored or that causes no harm if it is violated, it will not be worth the programmer’s time or effort to fix those “bugs.” Network sockets, file handles and databases are widely regarded as important by the Java programmers that make use of them. In particular, every program we found that violated one of these specifications also contained at least one path (and in general many paths) that adhered to the specification. These specifications represent interface usage patterns and contracts that programmers are truly trying to handle correctly. We will use the safety policies as well as the fault model discussed in the next section in order to find bugs in programs.

2.6 Exceptional Situation Fault Model

From the perspective of software reliability and bug-finding, a *fault model* limits the situations under which the program is expected to behave correctly. For example, if the fault model allows an adversary to replace the original program with malicious code before it executes it is very difficult to ensure that running the program will produce the desired results. Fault models are typically somewhat 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 [WFBA00]. 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. In the realm of security, worst-case fault models are the norm under the assumption that any exploitable security hole will eventually be targeted. We are interested in studying long-running program behavior in exceptional situations. 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 errors. Typically, however, we have access only to the program source code and cannot reliably 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. has found exactly that connection: "We did not find literature that documents the extent to which different JVMs actually translate such low-level faults into Java-visible exceptions. We performed a number of ad hoc experiments [...] to determine whether Java exceptions were indeed a reasonable way to simulate such faults. The results were satisfactory: using the Sun HotSpot JVM, 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 one of the [checked] exceptions [being signaled at the language-level.]" [CDCF03]

The Java programming language features two types of exceptions: `checked` (or *declared*) and `unchecked` [GJS96]. `Unchecked` exceptions typically describe program bugs and are signaled when an array is accessed beyond its limits, when a null pointer is dereferenced or when the program attempts to divide an integer by zero. We do not include `unchcked` exceptions in our fault model, and thus do not report bugs related to them, for two reasons. The primary reasons is that experience indicates that if we allow `unchecked` exceptions in our fault model and report a bug that can only arise if the presence of an `unchecked` exception, the programmer is less likely to believe the error report. Programs typically have complicated and difficult-to-capture invariants to ensure that such exceptions do not occur. For example, is undecidable to determine statically whether a particular division statement will divide by zero. Even though many `unchecked` exceptions do occur in practice (e.g., the `NullPointerException`) they have a second-class status in the eyes of many Java programmers. If we want the output of our tool to be trusted we must avoid reporting bugs that programmers do not believe. The second reason we avoid `unchecked` exceptions is that they typically have no associated handling behavior. Programs are not required to (and thus rarely do) handle `unchecked` exceptions. A program that tries to read beyond the end of an array is not typically expected to somehow recover (although there are cases where such things happen). Instead, the `unchecked` exception typically kills the running program and provides debugging information (e.g., a stack trace) so that the programmer can fix the bug once and for all. If a program is going to terminate immediately anyway there is no reason to explicitly close certain resources (e.g., operating system file handles will be reclaimed by the operating system automatically when the process terminates).

Programmers should attempt to restore invariants even in the presence of `unchecked` exceptions, but this fault model will not issue warnings when they fail to do so.

From the perspective of our bug-finding analysis, if the programmer is willing to consider additional error reports it is simple to extend the fault model with classes of `unchecked` exceptions (e.g., by treating integer division as a “method” that either returns normally or raises a divide-by-zero exception). Finally, the compensation stacks we will propose in Chapter 4 help to guard resources and restore invariants in the presence of both `checked` and `unchecked` exceptions.

`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 network connectivity problems, security violations, disk errors and database transaction failures. The Java Type System [GJS96] requires that programmers either catch and handle all `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. For example, the declaration for the `createNewFile` method of the `java.io.File` class looks like this (with some of the comment and all of the body elided):

```
/**
 * This method creates a new file [...]
 *
 * @exception IOException  If an I/O error occurs
 *
 * @exception SecurityException  If the SecurityManager will not allow
 * this operation to be performed.
 */
public boolean createNewFile() throws IOException, SecurityException {
  [...]
}
```

`Checked` exceptions are thus part of the contract that a programmer programs against when using a Java interface. The `createNewFile` method above can *either* return a `boolean` *or* it can signal an `IOException` *or* it can signal a `SecurityException`.

Our fault model is that any invoked method can either terminate normally or signal any of its declared `checked` exceptions. For example, we expect the program to adhere to the specifications for its important resources (e.g., `Sockets`, `ResultSets`) even if the `createNewFile` method signals a `SecurityException`. We do not, however, expect the program to adhere to its specifications in the presence of `unchecked` exceptions (i.e., programmer bugs) like `DivisionByZero`, since they typically do not involve exception handling, can be dealt with by normal testing, and are not necessarily a visible part of an interface.

As a corner case our fault model forgives all errors when the programmer explicitly aborts the program. A call to `java.lang.System.exit` terminates the program and does not flag any errors even if some of resources have not been properly handled (e.g., even if a `Stream` object has not been returned to its accepting "closed" state). Explicit calls to `exit` typically mean either that the program's work is done and that the outside system will clean up all resources or that the programmer is aware of the error. We are primarily interested in finding errors 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.

Finally, when we do not have the declared list of `checked` exceptions for a method we adopt a conservative fault model that 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 xzar() throws IOException {
  int a = MysteryOne();          // unknown method
  int b ;
  try {
    b = MysteryTwo();            // unknown method
  } catch (MysteryException e) {
    b = 0;
  }
  return a + b;
}
```

If we do not have the declarations for `MysteryOne` and `MysteryTwo` in the above code we will assume that `MysteryOne` always terminates normally and that `MysteryTwo` can either terminate normally or signal a `MysteryException`. We conservatively do not assume that either method can signal an `IOException` even though the enclosing method `xzar` declares that it may propagate such exceptions. We conducted our experiments on open-source programs for which the source code was publicly available, but many of those programs relied on proprietary third-party libraries for which the source was not freely available. This fault model may fail to expose real bugs (e.g., if `MysteryOne` really can signal an `IOException`) but will avoid making hasty conclusions about the presence of errors in the program. Our fault model is thus well-suited for bug-finding but ill-suited to proving correctness or verifying the absence of bugs.

2.7 A Static Analysis For Exception-Handling Code

We now present a static dataflow analysis for finding bugs in a program with respect to a given specification and a given fault model. The analysis is path-sensitive, intraprocedural and context insensitive. It abstracts away data values and only keeps track of the resources mentioned in the specification.

2.7.1 Building the Control Flow Graph

The analysis begins by selecting a method from the target program and constructing its control flow graph (CFG). Our CFG construction is standard [ASU86] except for our handling of method invocations (including constructor calls, etc.) and for our handling of `finally`. Method invocations are treated according to our fault model. A method invocation node has an edge leading to the statement that comes after it as well as zero or more edges representing possible exceptional situations. To determine these edges representing exceptional control flow 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 declared raised exception is a subtype of the exception the `catch`-clause handles. If it is we add an exceptional control flow edge from the method to the beginning of 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 an exceptional control flow edge to the `end` node of the CFG. Following our fault model, if we do not have the source for the invoked method we assume that its list of `checked` exceptions is exactly equal to the list of lexically-enclosing `catch` clauses.

`Finally` clauses are the second complication in our CFG construction. In essence, we must remember how control reaches a `finally` block in order to determine where control flows after that block. In a `try-finally` statement the `finally` clause is executed if the `try` clause terminates normally or if the `try` clause signals an exception. If the `try` clause does not singal 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

```
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 Trace
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	`AB   F`
8	`IO`	`Network`	`Security`	`ABC E`

Figure 2.10: Example code involving exceptions and `finally`.

is executed. However, if the body of the `finally` clause itself signals a new exception or executes a `continue`, `break` or `return` statement, that new control flow overrides the "pending" exception.

Consider the example code in Figure 2.10. There are a number of ways that control can pass through that code fragment. We enumerate each possible path through the code in Figure 2.10 in order 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 2.10 the middle columns 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 exception) and `none` indicates that the method terminated normally (i.e., without raising an exception).

Of the eight control flow paths through the code in Figure 2.10, only the first is possible if there are no exceptions. Since neither `A` nor `C` signaled an exception, control passes from the 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 handled at `B` and is thus not re-signaled 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 re-signaled after `C`. Since `NetworkException` is not a subtype

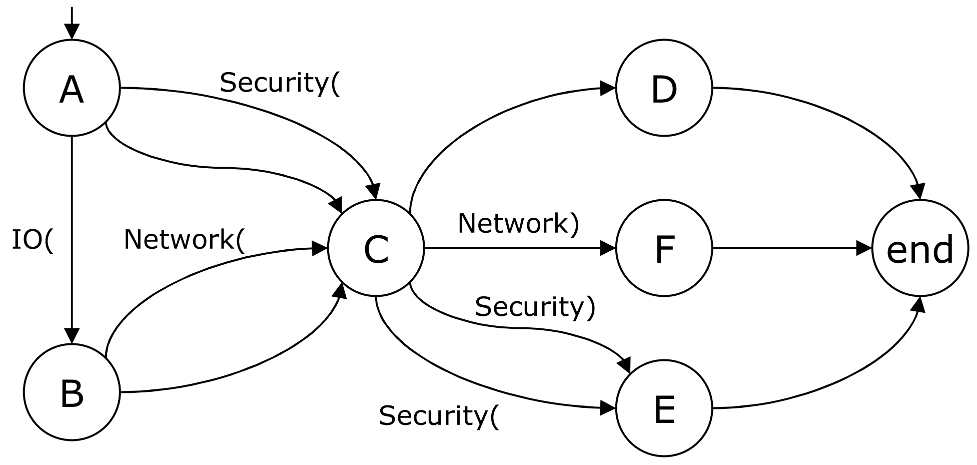


Figure 2.11: A control flow graph for the example code involving exceptions and finally.

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 notoriously-difficult-to-verify `JSR` (jump to subroutine) Java Bytecode instruction was designed specifically in order to implement this behavior [LY97]. While `try-catch-finally` is conceptually simple, it has the most complicated execution description in the language specification [GJS96] 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.

Our goal in constructing the control flow graph is to admit exactly those paths and behaviors. Unfortunately, naively adding exceptional control flow edges to account for all of the behaviors in Figure 2.10 introduces too many behaviors by failing to account for context and execution history. Applying the CFG construction algorithm given above to the code in Figure 2.10 yields a CFG similar to the one in Figure 2.11. Blank edges represent normal control flow. Labeled edges represent exceptional control flow and are labeled with the associated exception (and possibly some context-free reachability information). 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 explore that infeasible path because any error reported on it would necessarily be spurious.

One solution is to duplicate every `finally` block (and, typically, all of the code that comes after it) 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 scalable analysis and the required duplication is non-trivial.

A second solution is to use a variant of context-free language reachability [RHS95], as shown in Figure 2.11. 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 in order to obtain a precise context-sensitive dataflow analysis by correctly matching up method invocations (left parentheses) and returns (right parentheses) [RHS95]. Here we use left parentheses to indicate "normal" or "originally-signaled" exceptions and right parentheses to indicate exceptions that are re-signaled after a `finally` block. The language is more complicated than the standard 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. In our implementation we compute our path-sensitive dataflow 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 and only re-signaling an exception along a path when appropriate.

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 achieve the desired behavior by implementing `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. For example, we might model the following code:

```
while (predicate) {
  X();
  try {
    if (this) break; // execute finally block ...
    if (that) continue;  // ... before leaving
  } finally {
    Y();
  }
}
```

as equivalent to:

```
try {
  while (predicate) {
    try {
      X();
      try {
        if (this) throw BreakPseudoException;
        if (that) throw ContinuePseudoException;
      } finally {
        Y();
      }
    } catch (ContinuePseudoException) { }
  }
} catch (BreakPseudoException) { }
```

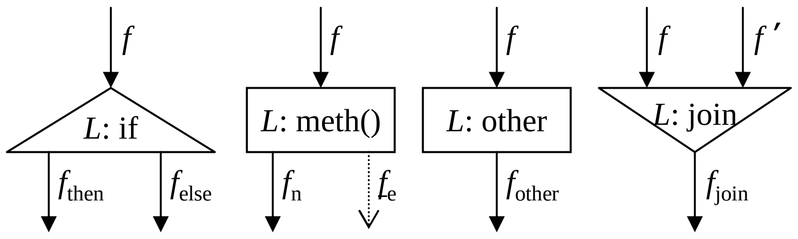
Thus we will correctly model Java's behavior of executing the `finally` clause (and statement `Y`) if the `break` or `continue` is executed. We use these techniques to build a CFG for each method in the program.

2.7.2 Dataflow Analysis

Using the control flow graph constructed according to the fault model we now present our dataflow analysis to find violations of the given specification. This analysis yields paths through methods on which mistakes may occur and can be used to direct changes to the source code to improve exception handling. The analysis may mistakenly report correct code as buggy and may fail to report real errors. We have chosen to take a fully static approach to avoid the problems of test case generation and the unavailability of third-party libraries. Path coverage and test case generation are particularly thorny problems in the context of run-time errors and exceptions, which are typically rare and difficult to trigger.

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

Given the control-flow graph, our flow-sensitive, intraprocedural dataflow analysis [Kil73, DLS02, ECCH00] is designed to find paths along which programs violate the specification (typically by forgetting to discharge obligations) in the presence of run-time



$$
\begin{aligned}
f_{\text{then}} &= \mathsf{visit}(f, L) \\
f_{\text{else}} &= \mathsf{visit}(f, L) \\
f_{\text{n}} &= \begin{cases} \mathsf{visit}(f, L) & \text{if meth} \notin \Sigma \\ \mathsf{visit}(\langle \{s'\} \cup \mathcal{S}, L' \rangle, L) & \text{else if } f = \langle \mathcal{S} \cup \{s\}, L' \rangle \text{ and } \delta(\langle s, \text{meth} \rangle) = s' \\ \mathsf{visit}(\langle \{s\} \cup \mathcal{S}, L' \rangle, L) & \text{else if } f = \langle \mathcal{S}, L' \rangle \text{ and } \delta(\langle s_0, \text{meth} \rangle) = s \\ \emptyset & \text{otherwise (indicates a policy violation)} \end{cases} \\
f_{\text{e}} &= \begin{cases} \mathsf{visit}(\langle \{s'\} \cup \mathcal{S}, L' \rangle, L) & \text{if } f = \langle \mathcal{S} \cup \{s\}, L' \rangle \text{ and } \delta(\langle s, \text{meth} \rangle) = s' \\ & \text{and } s' \in F \\ \mathsf{visit}(f, L) & \text{otherwise} \end{cases} \\
f_{\text{other}} &= \mathsf{visit}(f, L) \\
f_{\text{join}} &= \begin{cases} \mathsf{visit}(\mathsf{shorter}(f, f'), L) & \text{if } f = \langle \mathcal{S}, L \rangle \text{ and } f' = \langle \mathcal{S}, L' \rangle \\ \mathsf{visit}(f, L) \cup \mathsf{visit}(f', L) & \text{otherwise} \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{visit}(\langle \mathcal{S}, L' \rangle, L) &= \{\langle \mathcal{S}, L' \bullet L \rangle\} \\
\mathsf{shorter}(\langle \mathcal{S}, L \rangle, \langle \mathcal{S}, L' \rangle) &= \begin{cases} \langle \mathcal{S}, L \rangle & \text{if } |L| \leq |L'| \\ \langle \mathcal{S}, L' \rangle & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 2.12: Analysis flow functions.

errors. We abstract away data values, and retain as symbolic dataflow 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. If a dataflow fact at the end of method contains outstanding obligations (i.e., a resource governed by the specification that is not in an accepting state), 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 2.5). If a specification contains multiple state machines we assume that the program is checked against each one independently. In practice is it trivial to extend the algorithm presented here to handle multiple specifications simultaneously. 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 the ESP [DLS02] and Metacompilation [ECCH00] projects, we combine a degree of symbolic execution with dataflow and often keep state associated with multiple distinct paths that pass through the same program point.

Each path-sensitive dataflow fact f is a pair $\langle \mathcal{S}, L \rangle$. The first component $\mathcal{S}$ is a multiset of specification states. So for each $s \in \mathcal{S}$ 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. For example, a program could have two open `Socket`s. The second component L is a *path* or list of program points between the start of the method and the current CFG edge. The path L is important for reporting potential violations. Consider the following program using the `Socket` policy from Figure 2.7:

```
L1: new Socket
```

```
L2: if (predicate) {
L3:   Socket.close
    }
```

The dataflow fact f just before program point `L3` is the pair $\langle\{\mathsf{opened}\}, \mathtt{L2} \bullet \mathtt{L1}\rangle$. The interpretation is that if the program runs such that statements `L1` and `L2` are executed in that order there will be one outstanding object governed by the `Socket` policy and it will be in the opened state of that policy. We store the path in reverse order, similar to a backtrace in a debugger, but the choice is arbitrary.

2.7.3 Flow Functions

The flow functions are determined by the safety policy and are given in Figure 2.12. 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 dataflow facts at join points each flow function technically takes a single incoming dataflow fact and computes a set of outgoing dataflow 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 dataflow 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, etc.) and our dataflow fact propagates directly from the incoming edge to both outgoing edges. We write $\mathsf{visit}(f, L)$ 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 2.12. 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 dataflow fact f contains a state s that could transition on that method we apply that transition and then append the label L. As a concrete example:

```
L1: new Socket
L2: Socket.close
```

The fact f coming in to `L2` is $\langle\{\mathsf{opened}\}, \texttt{L1}\rangle$. The state s that could transition on `Socket.close` is opened. Since $\delta(\text{opened}, \texttt{Socket.close}) = \text{closed}$, we obtain $s' = \text{closed}$ and end up with the expected outgoing state $\langle\{\mathsf{closed}\}, \texttt{L2} \bullet \texttt{L1}\rangle$. This is similar to the way tracked resources are handled in the Vault type system [DF01].

The third possibility for a method involves creating a new important resource. 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 from the start state occur only at the start state (e.g., $\forall (s_0, e) \in \text{Dom}(\delta). \forall (s', e') \in \text{Dom}(\delta).\ e = e' \implies s' = s_0$).

The final case for a method invocation indicates a potential error in the program. In this case we have an event that is important to the specification but for which there is no appropriate object. For example, a method that begins with `Socket.close` does not have a legal `Socket` to `close`. 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 (i.e., the outgoing fact is the empty set) in order to avoid cascading error reports.

A method invocation may also raise a declared exception, represented by the f_e edge in Figure 2.12. Note that unlike the successful invocation case and as per our fault model, we do not typically update the specification state in the outgoing dataflow fact. This is because the method did not actually terminate successfully and thus presumably did not actually 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, invoking them until they succeed. Since no programs we have observed do so, it would create unnecessary spurious error reports. The check $s' \in F$ requires that the result of applying this method would put the object in an accepting state.

The grouping (or 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 [DLS02] to grouping sets of symbolic states. We merge facts with identical obligations by retaining only the shorter path for error reporting purposes (modeled here with the function $\mathsf{shorter}(s_1, s_2)$). In general, however, we may end up considering the same program point multiple times. For example:

```
if (predicate) {
      new Socket
  L1:
} else {
      new Connection
      L2:
    }
L3:
```

The join point at `L3` has incoming edges from `L1` and `L2`. Since an opened `Socket` and an opened `Connection` are not the same, `L3` (and all succeeding statements) will be considered twice: once with the history from `L1` and once with the history from `L2`.

To ensure termination we stop the analysis and flag an error 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) error handling was always such that an exception would escape the enclosing loop. The analysis is exponential in the worst case (e.g., sequential `if` statements with every path containing a different obligation list) but quite efficient in practice. For example, performing this analysis on the 57,000-line `hibernate` program, including parsing, typechecking and printing out the resulting error traces, took 104 seconds and 46 MB of memory on a 1.6 GHz machine.

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

2.7.4 Error Report Filtering

Finally, we use heuristics as a post-processing step to filter candidate violations. The analysis as presented is conservative in that it will find all violations of the policy with respect to the fault model but it may also point out spurious warnings. A spurious error report that refers to code that does not contain a mistake is called a *false positive.* Based on a random sample of two of our benchmarks, 30% of the error reports produced by our analysis are false positives. We believe that number to be unacceptably high because we want the cost of using this analysis, including the cost of wading through screens of false reports, to be low. Based on an exhaustive analysis of the false positives reported by this analysis, we designed three simple filtering rules.

When a violation $\langle \mathcal{S}, L \rangle$ is reported, we examine its path L. Every time the path passes through a conditional of the form `t = null` we look for a state $s \in \mathcal{S}$ where $s \notin F$ and s represents an object of type `t`. If we find such a state we remove it from $\mathcal{S}$. This addresses the very common case of checking for `null` resources:

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

Since we abstract away data values, we would report a false positive in such cases. Intuitively, the resource is not leaked along this path because the program has checked and ensured that it was not allocated.

Second, we examine L for assignments of the form `field = t`. For each such assignment we remove one non-accepting state of type `t` from $\mathcal{S}$. When important resources are assigned to object fields, the object almost invariably contains a separate “cleanup” method that is charged with releasing those resources. As we shall discuss in Section 4.2, this cleanup method is almost never an actual finalizer.

Finally, if L contains a `return t`, we remove one non-accepting state of type `t` from $\mathcal{S}$. Methods with such `return` statements are effectively wrappers around the standard library constructors and the obligation for handling the resource falls to the caller. We did not observe wrappers for standard library `close` functions, 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 $\mathcal{S}$ has been depleted so as to contain only states $s \in F$ the candidate violation is not reported. Our first heuristic helps to reduce false positives introduced by data abstraction. The second and third heuristics help to address false positives caused by the intraprocedural nature of our analysis. These three simple filters eliminate *all* false positives we encountered but could cause this analysis to miss real errors. Based on a random sample of two of our benchmarks, applying these three filters causes our analysis to miss 10 real bugs for every 100 real bugs it reports. We discuss the analysis results in the next section.

2.7.5 Analysis Summary

Our fault model is specific to Java, and we use it to construct a control-flow graph where method invocations can raise declared exceptions. We chose Java because experiments show that exceptions and run-time errors are correlated and because method signatures include exception information. Our dataflow analysis is language-independent.

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. The analysis is intraprocedural for efficiency since we track separate execution paths. This leads to false positives, which we can eliminate easily in practice, but our heuristics for doing so may also mask real errors. The analysis abstracts away data values, keeping instead a set of outstanding resource states with respect to the specification as per-path dataflow 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 dataflow facts separate if they have distinct sets of resources.[1] We report a violation when a path leaves a method (normally or exceptionally) with a resource that is not in an accepting state.

2.8 Poor Handling Abounds

In this section we apply the analysis from Section 2.7.2 and the specifications from Section 2.5 to show that many programs make mistakes in their handling of exceptional situations. We consider a diverse body of twenty-seven Java programs totaling four million lines of code. Each program is described briefly in Figure 2.13. Most of the programs were taken from the Sourceforge open source program repository [Sou03]. The programs include databases, business software, networking applications and software development tools. Most of the programs are well-known real-world applications in their areas. For example, `compiere` claims to be "the most popular open source business application with

[1]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.

Program	Description
`javad`	Java class file **disassembler**
`javacc`	**parser generator** for Java
`jtar`	GNU **tape archive** utility ported to Java
`jatlite`	infrastructure for building robustly **communicating agents**
`toba`	**translates Java class files into C** source code
`osage`	Java object **relational persistence framework**
`jcc`	direct **Java source to C translator**
`quartz`	**job scheduling system** that can be integrated with J2EE
`infinity`	**resource browser and editor** for the Infinity game engine
`ejbca`	J2EE-based **certificate authority**
`ohioedge`	multi-functional **customer relationship management** software
`jogg`	graphical **mp3 player** for ogg vorbis files
`staf`	software **testing automation framework**
`hibernate`	object / **relational persistence and query** service
`jaxme`	**compiles Java/XML binding schema** to Java classes
`axion`	**relational database** management system
`hsqldb`	high-performance SQL **relational database** engine
`cayenne`	**object relational** mapping framework and GUI **modeling tools**
`sablecc`	framework for **generating compilers** and interpreters
`jboss`	enterprise **middleware system** and application server
`mckoi-sql`	**SQL database** system
`portal`	**web portal**: personalization, web email, blogs, document libraries, message boards, etc.
`pcgen`	**character generator** for role-playing games
`compiere`	**enterprise resource planning, customer relationship** management, supply chain management and accounting
`aspectj`	**aspect-oriented extension** to Java
`ptolemy2`	**heterogeneous concurrent modeling** and design
`eclipse`	integrated **development environment**

Figure 2.13: Description of Java programs analyzed.

800,000+ downloads", `ptolemy2` is a popular modeling program [BKL$^+$04], `jboss` claims to be the "#1 most widely used J2EE application server", `hibernate` [Hib04] claims to be the "#1 most widely used object/relation mapping solution for Java", and Eclipse has won dozens of awards for best development environment.

Figure 2.14 shows results from this analysis. The "Methods" column shows the number of methods that violate at least one policy. The "Database" policy refers to the API for linking Java programs to SQL databases given in Figure 2.9. Java programs consider this policy to be particularly important: the vast majority of `finally` blocks tried to deal with it. The `Stream` policy deals with any class (even a user-defined one) that inherits from `java.io.InputStream` but not `java.io.FileInputStream` and is given in Figure 2.8. The `File` policy covers acquiring and releasing `java.io.FileInputStream`s and is also given in Figure 2.8. Although both "normal" `Stream`s and `FileStream`s are important, many developers consider `FileStream`s to be more important so we have separated out the numbers that refer to them. We also applied the `Socket` policy from Figure 2.7. 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 safety policies. For example, in `eclipse` and `ptolemy2` only 10% of the source files mentioned resources covered by these safety policies, and in `aspectj` only 16% of the files did, making them behave like smaller programs.

Figure 2.14 includes every violation reported by the analysis that was not automatically filtered out using the heuristic techniques presented in Section 3.4.1. All of the

Program	Lines of Code	Methods with Errors	paths with errors per safety policy		
			Database	`File`	`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**	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**	34	6	19
`jaxme 1.54`	58k	**6**	1	12	0
`axion 1.0m2`	65k	**15**	1	61	5
`hsqldb 1.7.1`	71k	**18**	22	8	13
`cayenne 1.0b4`	86k	**7**	2	27	6
`sablecc 2.17.4`	99k	**3**	0	0	6
`jboss 3.0.6`	107k	**40**	134	5	53
`mckoi-sql 1.0.2`	118k	**37**	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**	0	504	46
`eclipse 5.25.03`	1.6M	**126**	0	181	252
total	3.9M	**818**	1129	1359	825

Figure 2.14: Error handling mistakes by program and policy.

The "Methods" column indicates the total number of distinct *methods* that contain violations. The "Database", "`File`", and "`Stream`" columns give the total number of acyclic control-flow *paths* within those methods that violate the given policy.

methods with errors were then manually inspected to verify that they contained at least one error. This manual inspection assumed that a method could raise any of its declared exceptions (i.e., it used the same fault model discussed in Section 2.6). The heuristics eliminate all false positives that the analysis would report on these programs. Thus from the perspective of our fault model there are no false positives in Figure 2.14.

The heuristic filters reduced the number of reported methods by 20% (from 1034 to 818) and the number of reported paths by 15% (from 3922 to 3320). The applicability of a heuristic depends on the coding practices of the program. For example, in `ejbca`, which favors populating `catch` blocks with statements like `if (c != null) c.close()`, there are 10 methods that are not reported because of the `if` filter and 4 that are not reported because of a combination of the `if` and `return` filters. In `mckoi-sql`, which makes use of wrappers and accessors like `getInputStream()`, 25 methods are elided by the `return` filter, 2 are not reported because of the assignment filter, and 1 is suppressed because of a combination of filters.

From our perspective, such false positives are worth mentioning because they represent places where code quality could be improved by other language-level mechanisms; if an analysis cannot reason about the code, the programmer may not be able to either.

All paths in Figure 2.14 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 result demonstrates that existing exception handlers contain mistakes.

The most common problematic exception was the Java `IOException`: it occurred somewhere in 597 of the error paths and was the final, uncaught exception in 474 of them. The `SQLException` was a close second, occurring in 877 traces and going uncaught in 114 of them. The `SecurityException` was third with 86 mentions and 68 uncaught instances. The disparity between these two numbers is quite telling: it shows that programs have some sort of error handling (`SQLException`s are caught) but that the handling code itself is not always correct (resources are still leaked). Other common exceptions with poor error handling included `FileNotFound`, `ClassNotFound` and `UnsupportedEncoding`.

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`, thus violating both the `Socket` and the "Database" specification. For simplicity, such cases are categorized in favor of the leftmost policy in Figure 2.14. To give one example, of the 59 possible error paths reported in `hibernate`, 34 involved violating multiple policies along a single path with up to 4 forgotten resources at once. Errors that cross safety policies argue strongly for the need to have an error-handling mechanism that supports multiple resources in sequence.

Finally, some programs contain some methods that never close these resources at all and others that close them carefully. For example, in `ejbca`'s `HttpGetCert.sendHttpReq` method, a `BufferedReader` is created but not closed (although two other resources are closed in that method). However, in the `loadUserDB` method of `ejbca`'s `RemoveVerifyServlet` class, `BufferedReader` is given its own `try-finally` statement and its `close` call is given its own exception handler within that `finally` block. We report `sendHttpReq` as a method with an error-handling mistake, following Engler et al. [ECC01], since the `ejbca` program takes care to handle `BufferedReader`s in some cases and is thus inconsistent with itself.

2.9 Analysis Conclusions

The analysis results in Section 2.8 show that common Java programs make a large number of mistakes with respect to important resources in the presence of exceptional situations. We found over 800 such mistakes in almost 4 million lines of code. Finding that many methods with errors helps to justify our design decisions.

Our fault model was actually fairly conservative with respect to injecting exceptional situations: neither `unchecked` exceptions nor third-party code were considered. Adding in other sources of exceptional situations might lead to discovering more bugs but might also lead to less believable bug reports. We will return to the issue of the relative importance of the bugs we find in Section 3.8.

Our analysis was intraprocedural, both because we were interested in scalability and because a complete call graph is difficult to construct for dynamically-bound component-based Java programs. Our analysis also abstracted away data values. Both of these choices helped to introduce false positives. We were able to filter out all false positives in practice, but our simple filtering rules led to a 10% false negative rate. We reported around 800 mistakes and could presumably have reported 80 more if we had been willing to pay the price of wading through false positives or constructing a more precise analysis. However, we feel it is more important to concentrate on fixing the 800 bugs we have already located or to find new classes of bugs by using better specifications than to try to find a few more similar mistakes.

Our analysis was motivated by an examination of the difficulties in using language-level exception handling (Section 2.1). We then looked at one restricted notion of what programs should be doing in the presence of exceptions (Section 2.5). We also had to put forth a fault model describing the interaction between legitimate real-world exceptional situations and software (Section 2.6). Given all of those components we presented a static dataflow analysis (Section 2.7).

We considered only a small number of simple specifications under the assumption that they would be sufficient to find a large number of mistakes. That assumption was borne out in practice. In the next chapter we will return to the issue of more complex specifications.

I don't divide the world into the weak and the strong, or the successes and the failures, those who make it or those who don't. I divide the world into learners and non-learners.

Benjamin Barber

Chapter 3

Mining Specifications To Find Defects

In this chapter we present an algorithm for automatically inferring specifications like those in Section 2.5. The algorithm is based on our previous observations about how programs deal with exceptional situations from Chapter 2. We will compare our algorithm to others and perform a qualitative and quantitative evaluation. Our goal is to present an algorithm that is mostly automatic, works on large programs, and finds specifications that can be used to find bugs and thus to improve software quality. The specification miner proposed here was first discussed in earlier work [WN05].

3.1 Introduction

Analyses that attempt to find software bugs or verify programs need a formal notion of what the program should be doing. Such a notion is often called a partial correctness specification or a safety policy. The qualifier "partial correctness" means that only some aspects of the programs behavior will be regulated. A partial correctness specification might cover the use of sockets or the handshaking in a network protocol but would not cover everything it means to be a webserver. The qualifier "safety" refers to a policy where violations can be detected in a fixed amount of time by a monitor. Safety policies typically have a "do not" flavor: do not attempt to acquire a lock you already have and do not attempt to send data over a closed socket. In contrast, "liveness" policies often deal with things that happen "eventually" in the future: the scheduler eventually services every request, the program will perform this action infinitely often or every lock is eventually released. In the case of specifications governing resources and APIs the line between safety and liveness often blurs for the special case of releasing a resource. A policy requiring a resource to be released eventually falls under the category of liveness, but can often be shoehorned into the realm of safety by requiring the the resource be released within a finite time or by the end of the method.

We are interested in finding bugs in programs before the programs are deployed. Verification tools that find such bugs require specifications. Most commonly available tools require or accept safety policies expressed as finite state machines (as in Section 2.5). For example, the SLAM [BR01], MOPS [CDW04], ESP [DLS02], Vault [DF01], Metacompilation [ECC01] and ESC [LN98] projects all make use of such specifications, as does the analysis we presented in Chapter 2.

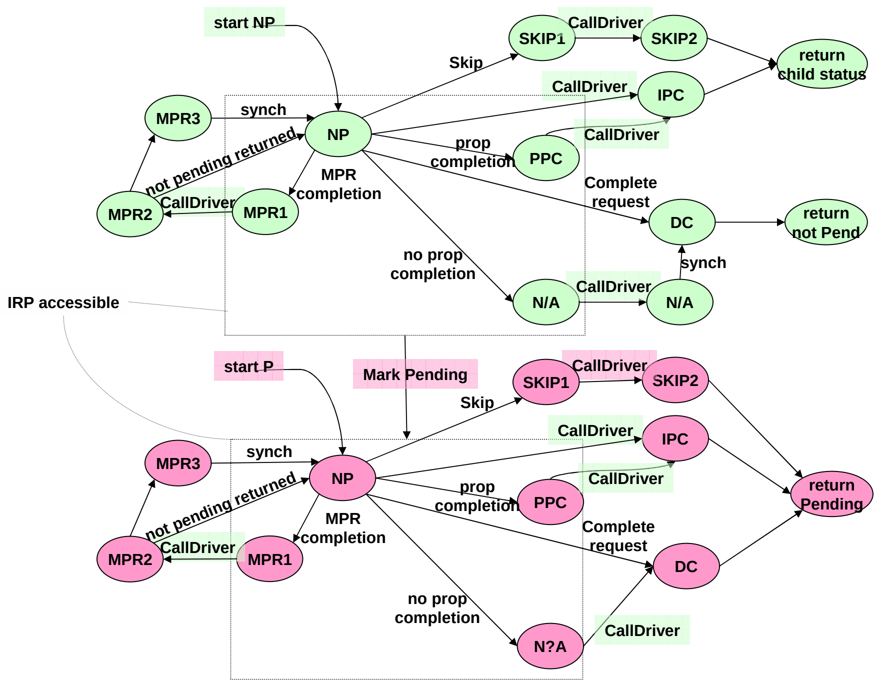


Figure 3.1: Windows Device Driver IO Request Packet Specification

3.2 Specification Complexity

Creating correct specifications is difficult, time-consuming and error-prone. Verification tools can only point out disagreements between the program and the specification. Even assuming a sound and complete tool, an imperfect specification can still yield false positives, by pointing out non-bugs as bugs, and false negatives, by failing to point out desired bugs. Crafting specifications typically requires program-specific knowledge.

Figure 3.1 shows an example of a complicated safety policy, a variant of which is used in practice [BR01]. The policy governs asynchronous pending and completion by device drivers in Microsoft Windows and was painstakingly formalized by Manuel Fähndrich from driver documentation. One problem with such complicated specifications is that it is difficult to tell if the specification itself is correct. When a specification is used to find bugs in a program, a potential bug is really just a disagreement between the specification and the program. In many cases it is the specification that needs to be amended. Some research projects explicitly address the task of debugging a faulty specification [AMBL03] but it is typically an expensive manual process. Since we are interested in low-overhead techniques that can be applied immediately we will consider simpler specifications (like those in Section 2.5) whenever possible. Smaller specifications can typically be inspected and verified rapidly (e.g., in under thirty seconds).

One way to reduce the cost of writing specifications is to use implicit language-based specifications (e.g., null pointers should not be dereferenced) or to reuse standard library specifications. More recently, however, a variety of attempts have been made to infer program-specific temporal specifications and API usage rules [ACMN05, ABL02, ECC01, WML02] automatically. These *specification mining* techniques take programs (and possibly dynamic traces, or other hints) as input and produce candidate specifications as output. In general, specifications could also be used for documenting, refactoring, testing, debugging, maintaining, and optimizing a program.

We focus here on finding and evaluating specifications in a particular context: given a program and a generic verification tool, what specification mining technique should be used to find bugs in the program and thereby improve software quality? Thus we are concerned both with the number of "real" and "false positive" specifications produced by the miner and with the number of "real" and "false positive" bugs found using those "real" specifications.

enter	class NormalizedEntityPersister's lock() method
invoke	hibernate.LockMode.greaterThan()
invoke	hibernate.engine.SessionImplementor.getBatcher()
invoke	java.util.Map.get()
invoke	hibernate.engine.Batcher.prepareStatement()
invoke	hibernate.persister.ClassPersister.getIdentifierType()
invoke	hibernate.type.Type.nullSafeSet()
invoke	hibernate.persister.ClassPersister.isVersioned()
invoke	hibernate.persister.ClassPersister.getVersionType()
invoke	hibernate.type.Type.nullSafeSet()
exception	hibernate.Hibernate2Exception
invoke	hibernate.engine.SessionImplementor.getBatcher()
invoke	hibernate.engine.Batcher.closeStatement()

Figure 3.2: Static trace fragment from `hibernate`

3.3 General Specification Mining

A *specification miner* takes a program as input and produces one or more candidate specifications with respect to a set of interesting program events. The program is typically presented to the miner in the form of a set of static or dynamic *traces*, each of which is a sequence of events and annotations (e.g., data values, records of raised exceptions). Static traces are generated from the program source code. Dynamic traces are produced by running an instrumented version of the program against a workload. In practice, *events* are usually taken to be context-free function calls (i.e., just the name of the called function rather than the entire call stack).

Figure 3.2 shows an example static trace fragment from the `hibernate` program. The trace begins inside the `lock` method of a class. A number of method invocations occur in sequence, an exception is raised, and then some additional methods are invoked (presumably inside a `catch` or `finally` block). The full trace would include additional information like

```
Session sess = sfac.openSession();
Transaction tx;
try {
    tx = sess.beginTransaction();
    // do some work
    tx.commit();
} catch (Exception e) {
    if (tx != null) tx.rollback();
    throw e;
} finally {
    sess.close();
}
```

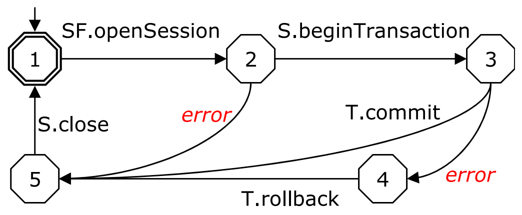


Figure 3.3: Documented `Session` temporal safety policy for `hibernate`

line numbers, argument types and values, and control flow. A specification miner would be given a large set of such traces from which to extract a candidate specifications.

Mined specifications (or policies) are typically finite state machines with events as edges. Miners typically produce state-machine specifications in which the edge labels Σ are a subset of the events from the traces. A run of the program adheres to the policy if it generates a sequence of events accepted by the state machine. As in Section 3.3, such policies commonly limit how an interface may be invoked (e.g., `close` cannot be called before `open` and must be called after it). Many program verifiers can check such properties, either per-object (as a form of typestate) or globally. Ammons et al. [ABL02] present a more formal treatment of the mining problem.

As a concrete example, we consider a policy for the interfaces of the `SessionFactory`, `Session` and `Transaction` classes in the `hibernate` program, a 57,000-line framework that provides persistent Java objects [Hib04]. The `Session` class is the central interface between `hibernate` and a client. The `Session` documentation includes explicit pseudocode and an injunction that clients should adhere to it. The code and five-state state machine specification are shown in Figure 3.3. We denote `SessionFactory` by `SF`, `Session` by `S`, and `Transaction` by `T`. A typical use of this interface would visit states 1 through 3, "do some work" there (involving events like `S.flush` and `S.save` that are not part of the input alphabet of the FSM and thus do not affect it), and then visit 5 and return to 1. In the next section we discuss our mining algorithm using this specification as a concrete example.

3.4 Specification Mining Algorithm

Our work on specification mining was motivated by observations of run-time error handling mistakes. Based on Chapter 2 we believe that client code frequently violates simple API specifications in exceptional situations (i.e., in the presence of run-time errors). We found such bugs using generic "library" specifications (i.e., the `Socket`, `Stream` and Database rules from Section 2.5) but we believe that we will be able to have a greater impact on software quality by looking for program-specific mistakes. Our mining algorithm produces policies dealing with resource leaks or forgotten obligations. We have found that programs repeatedly violate such policies, especially when run-time errors are involved.

Our technique is in the same family as that of Engler et al. [ECC01] but is based on assumptions about run-time errors, chooses candidate event pairs differently, presents significantly fewer candidate specifications and ranks presented candidates differently.

We attempt to learn pairs of events $\langle a, b \rangle$ corresponding to the two-state state machine policy given by the regular expression $(ab)*$. For example, from traces generated by the state machine in Figure 3.3 we might learn ⟨`SF.openSession`, `S.close`⟩, because every accepting sequence that transitions from state 1 to state 2 via `SF.openSession` must also transition from state 5 to state 1 via `S.close`. We learn multiple candidate specifications per program and present a ranked list to the user. For example, we might also learn the candidate specification ⟨`SF.openSession`, `T.rollback`⟩. Unlike some mining algorithms that produce detailed policies that must be manually debugged or modified, we produce simple policies that are designed to be accepted or rejected. With this approach we will not be able to learn the "complete" policies in Figure 3.3 (let alone Figure 3.1). However, the full policy in Figure 3.3 is closely approximated by ⟨`SF.openSession`, `S.close`⟩ and ⟨`S.beginTransaction`, `T.commit`⟩.

In a normal execution, events a and b may be separated by other events and difficult to discern as a pair. After an error has occurred, however, the cleanup code is usually much less cluttered and contains only operations required for correctness. Intuitively, a programmer who is aware of the specification will have included b in an exception handler, `finally` block, or other piece of cleanup code, making it easier to pick up than in a normal execution path. The pseudocode in Figure 3.3 demonstrates this sort of cleanup for the `T.rollback` and `S.close` events. If `S.close` is the only legal way to discharge a `Session` obligation, we expect to see `S.close` in well-written cleanup code.

We classify intraprocedural static traces as "error" traces if they involve exceptional control flow. These are the traces containing at least one method call that terminates with the raising of an exception. As described in Section 2.6, a Java method invocation can either complete successfully or signal a run-time error by raising one of its declared exceptions. For example, `SF.openSession` can *either* return a `Session` object *or* raise a program-specific `HibernateException`. Traces in which an exception is raised by a method invocation (i.e., traces in which an uncaught exception raised by a callee falls through and is handled by a caller) are "error" traces. The trace fragment in Figure 3.2 is an error trace.

Such exceptions are assumed to signal run-time errors or unusual situations. Traces in which no such exceptions are raised are "normal" traces. In Figure 3.3, a normal trace of events would involve the state sequence 1–2–3–5–1. An error trace would visit 1–2–5–1 or 1–2–3–4–5–1.

3.4.1 Filtering Candidate Specifications

In a trace consisting of k distinct events there are $k(k-1)/2$ possible candidate two-state specifications supported by the trace (i.e., $\langle$event i, event $j\rangle$ for all $1 \leq i < j < k$). Our set of static traces for `hibernate` alone includes 2028 unique events. Not all of them occur in the same trace but the number of potential specifications is still quite large. For example, the trace fragment in Figure 3.2 admits around fifty possible two-state specifications. A specification miner must therefore filter out the vast majority of those potential specifications in order to obtain a smaller list of true candidate specifications.

Let N_{ab} be the number of normal traces that have a followed by b, and let N_a be the number of normal traces that have a but not b. We define E_{ab} and E_a similarly for error traces. Given a set of traces, we consider all event pairs $\langle a, b\rangle$ from those traces such that all of the following occur:

Exceptional control flow (ex). Our novel filtering criterion is that event b must occur at least once in some cleanup code (e.g., a `catch` or `finally` block): we require

$E_{ab} > 0$. We assume that if the policy is important to the programmer, language-level error handling will be used at least once to enforce it. In `hibernate`, the `SF.openSession` and `S.beginTransaction` events never occur in cleanup code, thus ruling them out as the second event in a pair. The `T.commit`, `T.rollback` and `S.close` events all do occur in cleanup code, however. Other miners limit events to those on a user-specified list. For example, in Engler et al. [ECC01] the list includes functions whose names contain substring like "lock", "unlock", or "acquire". We prefer to automate the creation of this list because of the cost of acquiring specific knowledge about each target program. However, if such domain knowledge is available, it can be used instead of, or in addition to, the default from cleanup code. The occurrence of the event in normal execution traces will be used in Section 3.4.2 to rank candidate specifications.

One error (`oe`). There must at least one *error* trace with a but without b: we require $E_a > 0$. We are here only interested in learning specifications that will lead to finding program errors, and we assume that the programmer will make mistakes in the handling of exceptional situations. In contract, *normal* traces with a but not b would suggest that $\langle a, b\rangle$ is not a specification since we assume that the programmer adheres to the specification on easier-to-understand non-exceptional paths. Note that a specification miner interested in mining specifications for some task aside from bug-finding (e.g., program understanding or refactoring) would not use this filtering rule.

Same package (`sp`). Events a and b must be declared in the same package. For example, we assume that no temporal specification will be concerned with the relative order of an invocation of an `org.apache.xpath.Arg` method and a `net.sf.Hibernate.Session` method from separate libraries. The user can specify wider or narrower related groups if such information is available.

Dataflow (`df`). Every value and receiver object expression in b must also be in a. When dealing with static traces we require that every non-primitive type in b also occur in a. We thus assume that `Session SessionFactory.openSession()` may be followed by `void Session.close()` but forbid the opposite ordering. Intuitively, this also corresponds to finding edges that share the same node in policies like Figure 3.3. This notion is in contrast to other miners where a more precise dataflow analysis rules out some unwanted specifications. In our experiments this lightweight dataflow requirement has been sufficient to capture our intuitive notion of correlated events.

Given a set of candidate specifications that meet those requirements, we then rank them before presenting them to the user.

3.4.2 Ranking Candidate Specifications

In order to improve the usability of this technique, we present to the user a ranked list of the candidate specifications that satisfy the criteria described above. Our heuristics will assign higher ranks to candidates that are more likely to be real policies. We do not rank policies based on the number of bugs the policy would find in the program. However, as we will see in Section 2.8, ranking plays a much smaller role than eliminating extraneous candidates.

We assume $\langle a, b\rangle$ is more likely to be a policy if the programmer intends to adhere to it many times. We assume that normal traces represent the intent of the programmer and that some error traces represent unforeseen circumstances likely to contain bugs; thus

Event a	Event b	Real	N_a	N_{ab}	E_a	E_{ab}	Fail	rank	z-rnk	z-rnk$\|_N$
`SF.openSessi`	`S.close`	Yes	3	100	1348	1040		0.971	-73.5	2.40
`S.beginTrans`	`S.close`	?	2	56	1037	501		0.966	-73.3	1.66
`S.beginTrans`	`T.commit`	Yes	2	56	565	973		0.966	-33.9	1.66
`S.flush`	`S.close`	no	9	39	200	473		0.812	-17.0	-2.02
`T.commit`	`S.close`	?	1	57	474	504	df	0.983	-38.5	2.10
`S.beginTrans`	`S.save`	no	4	54	37	1501	ex	0.931	9.90	0.788
`SF.openSessi`	`T.commit`	?	47	56	1415	973	df	0.544	-81.0	-12.1
`SF.openSessi`	`println`	no	82	21	2121	267	sp	0.204	-130	-23.4

Figure 3.4: Static trace observations for `Session` events in `hibernate`.

we rank pairs according to the fraction of *normal traces* in which a is followed by b.

Our ranking for a candidate $\langle a, b\rangle$ is $N_{ab}/(N_{ab} + N_a)$. The best ranking is 1, and a reported specification with rank 1 has a followed by b in all normal paths. This ranking follows our earlier intuition that a pair for which N_a is high is unlikely to be considered a viable candidate.

Figure 3.4 shows observations for `Session`-related events on a set of static traces. The "real" column indicates whether $\langle a, b\rangle$ is definitely (Yes), possibly (?) or definitely not (no) a valid policy based on Figure 3.3. N_a is the number traces with a but not b, N_{ab} is the number of normal traces with a followed by b. E_a and E_{ab} measure the same figures for error traces. The counts are based on our dataset of `hibernate` static traces. The "Fail" column indicates which of our filtering requirements the pair fails to meet. Only the first four pairs meet the requirements and would be reported as candidates by our algorithm. The "rank" column reports $N_{ab}/(N_a + N_{ab})$ and high values indicate more likely specifications. The "z-rnk" column shows the z-statistic applied to all traces as in Engler et al. [ECC01], while the "z-rnk$|_N$" column shows the z-statistic restricted to normal traces.

All eight pairs could potentially be policies, but our requirements in Section 3.4.1

filter out the last four. Since `SF.openSession` does not occur in any error-handling code, we do not consider pairs like $\langle$`S.close`, `SF.openSession`$\rangle$. As desired, we rule out pairs like $\langle$`SF.openSession`, `T.commit`$\rangle$ with our dataflow requirement (there is no `Transaction` object available in event a). Our package requirement correctly rules out policies involving `printf`-like logging methods. Although logging functions do occur in cleanup code and can be connected by dataflow to other important events (e.g., if the object under consideration is cast to a string before being passed to a logging function), they are rarely part of the same package as important objects. Finally, while we cannot rule out pairs like $\langle$`S.flush`, `S.close`$\rangle$ (where `S.flush` is one of the "do some work" options that would occur at state 3 of Figure 3.3), we rank it lower because a smaller fraction of normal paths have that pairing (e.g., in Figure 3.4 that pair ranks 0.812 while the best pair involving `S.close` ranks 0.971).

The z-rank and z-rank$|_N$ columns of Figure 3.4 show the result of using the z-statistic for proportions [FPP98], an alternative ranking scheme, to rank candidate specifications, with the z-rank$|_N$ column being computed over normal traces only. The z-rank was used by Engler et al. [ECC01]. The z-statistic increases with the total number of observations involving a and decreases with the number of observations involving a but not b. Ignoring some constant factors, z-rank$|_N$ is equal to our ranking multiplied by $\sqrt{N_a + N_{ab}}$. We provide an empirical comparison of these three rankings in Section 2.8.

3.5 Other Specification Mining Techniques

We now describe the main characteristics of several existing specification mining approaches.

Strauss. Ammons et al. [ABL02] present a miner in which events from dynamic traces that are related by traditional dataflow dependencies form a *scenario*. The user provides a *seed* event and a maximum scenario size N. A scenario contains at most N ancestors and at most N descendants of the seed event. The seed can be any interesting event that is assumed to play a role in the specification. Such scenarios are fed to a probabilistic finite state machine learner. The output of the learner, a single policy, is minimized and may further be "cored" by removing infrequently traversed edges or "debugged" and simplified with the user's help [AMBL03].

WML-static. Whaley et al. [WML02] propose two methods for deriving interface specifications for classes based on an explicit representation of typestate in member fields.

In the first, the user specifies a class in the program. Traces are generated statically by considering all pairs $\langle a, b\rangle$ of invocations for methods a and b of that class. If b conditionally raises an exception when a field has a certain constant value and a always sets that field to that value, $\langle a, b\rangle$ is considered a violation of the interface policy. For example, the `close` method might set the field `opened` to false, and the `read` method might raise an exception if `opened` is false. The single final specification consists of all other pairs $\langle a, b\rangle$, represented as an NFA with one state per method. This miner explicitly looks for "a must not be followed by b" requirements, and by considering all possible method pair interactions it discovers what can follow a as well. In our experiments, we used an extended version of the miner that considers multiple fields and inlines boolean methods.

JIST. The JIST tool of Alur et al. [ACMN05] refines the WML-static miner by using predicate abstraction for a more precise dataflow analysis. The user specifies a class and an undesired exception, as well as providing a set of predicates and a specification of size k. A boolean model of the class is constructed based on the predicate set, and a model checker determines if invoking a sequence of methods raises the given exception. If it can, that sequence is removed from the specification. The process finds the most permissive policy of that size that is safe with respect to the predicates and the exception. As with Strauss, the output of the analysis is minimized using an off-the-shelf FSM library. In a WML-static policy, states represent the last invoked method. In JIST, states represent predicate valuations, which in turn represent object state. For example, JIST could produce a policy in which the sequence $\langle a, b \rangle$ is allowed but $\langle a, a, b \rangle$ is not. Thus, in JIST's more general policies, states do not correspond directly to the last method invocation.

WML-dynamic. Whaley et al. [WML02] also present a dynamic trace analysis that learns a *permissive* policy for a given class. Such a permissive specification is the most restrictive policy that accepts all of the training traces. Each field of the class is considered separately. Only events representing client calls to methods of that class that read or write that field are examined. If a is immediately followed by b in the trace, an edge from a to b is added to the policy. The single output policy for the class is formed from the per-field policies.

ECC. Engler et al. [ECC01] describe a technique for mining rules of the form "b must follow a" as part of a larger work on may-must beliefs, bugs, and deviant behavior. If b follows a in any trace, the event pair $\langle a,\ b \rangle$ is considered as a candidate specification.

A pair $\langle a,\ b \rangle$ is a candidate policy if the events a and b are related by dataflow and if there are both traces in which a is followed by b and traces in which a is not followed by b. A series of dependency checks is employed: two events are related if they have either the same first argument, or have no arguments, or if the return value from the first passed as the sole argument to the second. The user may also restrict attention to a certain set of methods.

ECC produces a large number of candidate policies. Engler et al. use the z-statistic for proportions to hierarchically rank candidates:

$$z(n, e, p_0) = (e/n - p_0)/\sqrt{p_0(1 - p_0)/n}$$

where n is the number of traces that contain an a followed by a related b, and e is the number of traces that contain an a without such a b. The z-statistic measures the difference between the observed ratio and an expected ratio p_0. Engler et al. use the ranking because it grows with the frequency with which the pair is observed together and decreases with the number of counter-examples observed. They take $p_0 = 0.9$ based on the assumption that perfect fits are uninteresting in bug-finding and that error cases are found near counter-examples. In our experiments we have found that ECC's assumptions tend to hold true for normal traces but not for error traces (where the frequency counts are quite high if the traces are static and often quite low if the traces are dynamic).

3.6 Qualitative Comparison of Mining Techniques

In this section we present experiments comparing these mining techniques. We evaluate a miner in terms of the policy it produces and later in terms of the number of bugs found by the that policy. When comparing miners we abbreviate our miner (defined in Section 3.4) by WN. Specifications can have many other uses (e.g., refactoring, optimizing, proofs of correctness, test suite evaluation, program understanding, etc.), but such uses tend to be even more difficult to compare and evaluate directly. We leave a comparative study with different evaluations for future work.

The first experiment compares miner performance on policies governing `hibernate`'s `SessionFactory`, `Session` and `Transaction` classes, as described in Section 3.3. This example was chosen because one policy for it is clearly described in the documentation, and also because that policy is complex enough that none of the miners can expect to learn it perfectly (e.g., our technique is unable to find all of the pieces of the full specification because of its assumptions about run-time errors). ECC and our technique both find policies about these classes (and others) automatically. For the purposes of comparison, however, we restrict all miners to policies about these three classes. For Strauss, WML and JIST we also provide all of the appropriate parameters (e.g., class names, predicates). We present the mined specifications, describe them qualitatively, and then report the number of bugs each specification finds.

For the purposes of the comparison we present the same raw trace data to each algorithm that looks at client code. Different techniques ignore different trace aspects (e.g., ECC ignores package declarations, our technique ignores precise data values, Strauss ignores exception annotations). In addition, some amount of human help was given to every miner. For ECC and WN, two of the top seven candidate policies were manually selected. For Strauss and WML-dynamic, a slice or core of the learned policy was selected. For JIST and WML-static, all relevant predicates and fields were given.

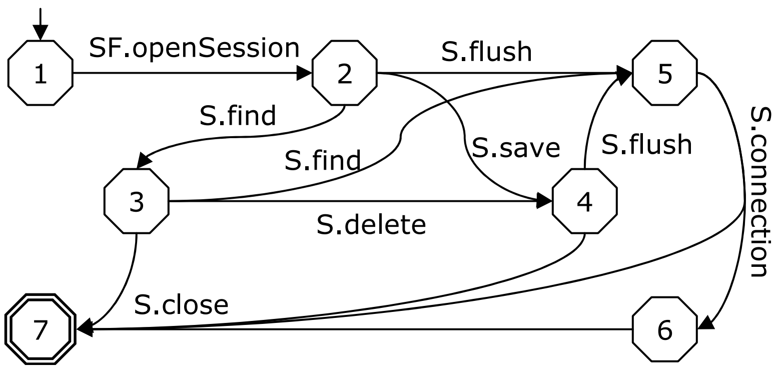


Figure 3.5: A slice of the `Session` policy learned by Strauss.

3.6.1 Mined `hibernate` `Session` specifications

Strauss, WML-dynamic, ECC, and our technique all learned policies similar to the documentation-based policy shown in Figure 3.3.

The Strauss policy (Figure 3.5) captures the beginning and the end of the Figure 3.3 (e.g., start with `openSession`, end with `close`) closely but is less precise than Figure 3.3 in the middle. Strauss's use of frequency information means that common sequences of events like `find` and `delete` are included as part of the policy. Paths through states 2–6 are all particular instantiations of the "do some work" state 3 in Figure 3.3. Compared to Figure 3.3, a sequence of two `flush` events after an `openSession` is incorrectly rejected by the Strauss policy while a sequence that has `beginTransaction` but no `rollback` or `commit` is incorrectly accepted. Figure 3.5 gives the "hot core" of the policy; the full learned specification overfits the data and has 10 states and 45 transitions.

The WML-dynamic policy permissively accepts all of the input traces. A slice is shown in Figure 3.6, the full policy has 27 states and 117 transitions. We abbreviate

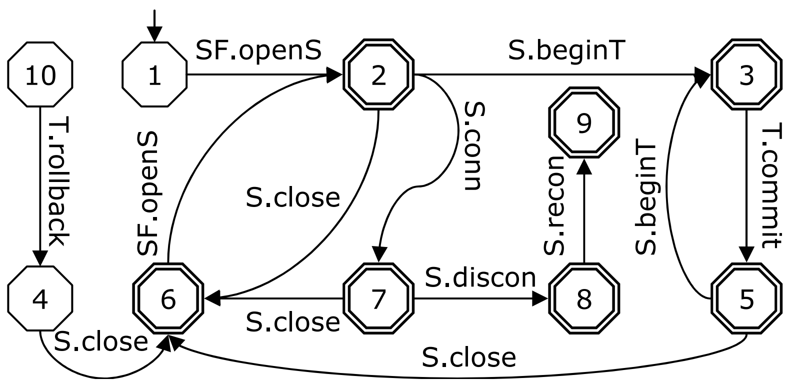


Figure 3.6: A slice of the `Session` policy learned by WML-dynamic.

#	z-rank	Event a	Event b	Real
1	9.896	S.beginTransaction	S.save	Yes
2	1.686	S.reconnect	S.load	no
3	1.634	S.getLockMode	S.close	no
4	0.609	SF.openConnection	SF.closeConnection	Yes
5	0.430	S.disconnect	S.reconnect	no
6	0.309	S.getLockMode	S.load	no
7	-0.430	S.disconnect	S.load	no

Figure 3.7: The top seven `Session` policies learned by ECC.

`openSession` by `openS` and so on for readability. The slice captures the highlights of Figure 3.3 (e.g., the `openSession-beginTransaction-commit-close` cycle in states 1–2–3–5–6) but fails to reject observed illegal behavior (e.g., forgetting `close`) and rejects unobserved legal behavior (e.g., `reconnect` followed by `close`). WML-dynamic makes a strong frequency assumption: a transition is valid if and only if it is observed. By contrast, our algorithm's `ex` and `oe` filters rule out some observed illegal behavior. An ideal bug-finding context for WML-dynamic would involve training data of known high quality and high coverage and some form of cross-validation for error detection.

#	Rank	Event a	Event b	Real	ECC Rank
1	1.000	S.iterate	S.close	no	286
2	1.000	S.getIdentifier	S.close	no	28
3	0.971	SF.openSession	S.close	Yes	256
4	0.971	S.createQuery	S.close	no	269
5	0.969	S.find	S.close	no	290
6	0.966	S.beginTransaction	T.commit	Yes	175
7	0.966	S.beginTransaction	S.close	no	254

Figure 3.8: The top seven `Session` policies learned by our miner (WN).

Figure 3.7 shows the top seven policies for these classes learned by ECC. Each policy requires an instance of "Event a" to be followed an instance of the corresponding "Event b". The "Real" The "z-rank" column gives the hierarchical z-rank for the policy. The "Real" column column indicates whether the specification is decidedly a false positive (no) or possibly valid (Yes). ECC learned 350 such candidate policies. The z-statistic favors frequent pairs: the pair ⟨`beginTransaction`, `save`⟩ occurs on more than 1,500 traces, and is thus a common practice, but is not strictly required.

The results from our algorithm are given in Figure 3.8. The "Rank" column gives our rank for that candidate specification as defined in Section 3.4.2. The "ECC Rank" column shows the ranked number (out of 350, low represents a likely specification) given to that policy by the ECC algorithm. In general policies favored by our algorithm were ranked unfavorably by ECC. Our approach learned 15 candidate policies, of which 2 are real. Two of the three main aspects of the documented specification, ⟨`openSession`, `close`⟩ and ⟨`beginTransaction`, `commit`⟩, appear as #3 and #6 on the list. Since we explicitly look only for pairs $\langle a, b\rangle$ that occur in almost all normal traces we will not find the `rollback` policy (no normal traces include `rollback` events).

3.6.2 `Hibernate Session` typestate specifications

The `hibernate` documentation mentions one notion of `Session` typestate. It says that `Session` "instances of mapped entity classes ... may exist in one of two states: transient ... [or] persistent" and gives typestate transitions (e.g., "transient instances may be made persistent by calling `save`"). The code does contain defensive programming checks using this typestate that raise exceptions, (e.g., if the object passed to `save` was already persistent) just as WML-static and JIST assume. Unfortunately, neither WML-static nor JIST are able to learn this typestate because it is checked by verifying that an instance object is in a Java `Map`, a dynamic data structure kept at run-time, as shown in this `hibernate` code fragment:

```
if (object==null) throw new // WML-static and JIST handle this
    NullPointerException("attempted to lock null");
object = ProxyHelper.unproxy(object, this);
EntityEntry e = getEntry(object); // but not this dynamic Map lookup
if (e==null) throw new TransientObjectException
    ("attempted to lock a transient instance");
```

In this domain typestate checks *are* used for defensive programming, but the typestate depends on input values or complicated logic. In addition, no check raises an exception if `close`, `commit` or `rollback` are forgotten, and in general inspecting library code will miss policies about such methods, so WML-static and JIST cannot learn the full specification in Figure 3.3.

WML-static (Figure 3.9) discovers five illegal sequences of `Session` methods. It finds a useful undocumented `Session` typestate: the `connection` and `connect` variables track the state of a `Session` as it connects to, disconnects from and reconnects to a database. The `S.write` method checks these underlying typestate variables but does not set them.

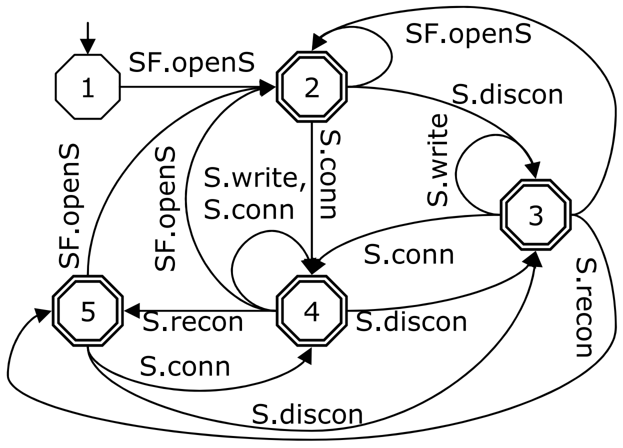


Figure 3.9: `Session` policy learned by WML-static.

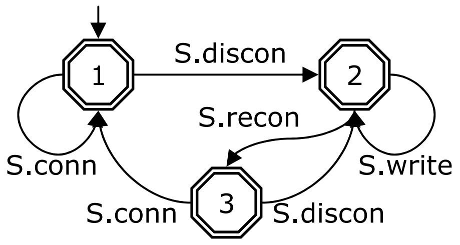


Figure 3.10: `Session` policy learned by JIST.

For WML-static and JIST, all unlisted method invocations (e.g., `S.close`) are orthogonal to the learned policy.

JIST (Figure 3.10) produces a more precise policy (e.g., it discovers that `connection` cannot be followed by `writeObject`) because it does not require methods to have a uniform impact on the object's typestate. In `Session` there are two typestate predicates that explicitly guard exceptions: `connection` and `connect`. Each state in the JIST policy represents a distinct valuation of two variables. The `writeObject` method may only be called when both are false. The `reconnect` method always sets the `connect` to true, so both techniques discover that it cannot be followed by `writeObject`. The `connection` method, however, has a different effect on the state variables depending on their current values, so WML-static cannot reason precisely about it. Any unlisted transition involving one of those four events violates the specification. All other method invocations (e.g., `S.close`) are orthogonal to the learned policy.

In our experiments the important difference between JIST and WML-static was not JIST's greater dataflow precision but JIST's more accurate characterization of interesting traces. All of the data manipulation was either too complicated for both methods to model (e.g., in heap data structures) or simple enough to meet WML-static's assumptions (e.g., comparing fields and constant values). These observations support our algorithmic design choice to use simple a dataflow requirement but to pay careful attention to characterizing exceptional traces.

In the next section we provide a quantitative comparison of specification miner bug-finding performance.

3.7 Specification Mining Experiments

We present experiments to compare the performance of our specification miner and the four others mentioned. We compare all algorithms on the `Session` policy and we compare our algorithm with ECC on one million lines of Java. We make quantitative evaluations of miner performance in terms of the number of bugs found by the specifications learned by the miner.

Mining Technique	False Positives	Real Errors
Strauss-Full	27	0
Strauss-Cored	20	46
ECC #1	30	20
ECC #4	1	0
WN #3	4	46
WN #6	3	20
WML-static	9	0
JIST	1	0

Figure 3.11: Miner bug-finding power for `hibernate` `Session` policies.

3.7.1 Comparison with Other Specification Miners

Given a candidate policy we use the algorithm described in Section 2.7 to find potential bugs by checking the policy against the source code. We could also use a number of similar or more powerful static checkers to evaluate the specifications [BR01, DLS02, ECC01, HJMS02]. Each potential bug is classified as a false positive or a real error by manual inspection. For example, if an application fails to close a file but immediately shuts down as a result of the error, the "leaked" file is classed as a false positive (see Section 2.6). However, a leaked database lock between the JVM (held on behalf of the program) and an external database is a bug if no finalizers close the connection when the program (but not the JVM) shuts down (as in Section 2.5).

In Figure 3.11 we present the results of using the mined specifications to find bugs in the `hibernate` program. Each "false positive" or "real error" represents a method where a trace fails to adhere to the given policy. The WML-dynamic approach is not shown because its specification accepts all of the traces by construction (thus it finds no bugs but yields no false positives).

Strauss-Full, the entire 10-state policy learned by Strauss, yields too many false positives to be effective for bug-finding. Twenty-five of the false positives are from traces along which `S.close` occurs after a sequence of "work" that the specification fails to accept. However, since the specification also has many accepting states (in particular, the state after `SF.openSession` accepts), errors involving forgetting `S.close` are not reported.

Strauss-Cored, the sliced policy shown in Figure 3.5, gives a reduced number of false positives compared to Strauss-Full, but still suffers from the same problems. However, Strauss-Cored is able to find 46 methods in which `openSession` is called but `close` is not (and 4 false positives involving `openSession`).

ECC, using specification #1 (the policy with the highest z-rank, see Figure 3.7), finds 20 methods that deal with `beginTransaction` improperly, 3 false positives involving `beginTransaction` and 27 false positives involving `save`. ECC specification #4 turns out not to be useful for bug finding. Its z-rank is high (28 of 30 traces that mention a also mention b), but it only occurs at one point in the source code. Either the z-rank$|_N$ or our ranking would rank it much lower ($N_a = 1, N_{ab} = 1$).

Our method using specification #3 finds all 46 of the `Session` leaks found by Strauss-Cored (and the same four false positives). In fact, the Strauss-Cored report is a superset of the WN #3 report. Using specification #6 we are able to find the 20 methods with `commit` and `rollback` mistakes that are also found by ECC. Along 20 of the 23 error paths we report in which `beginTransaction` occurs but `commit` does not, `rollback` does not either. The ECC #1 report is a superset of the WN #6 report (but with additional false positives).

Neither the WML-static nor the JIST specification lead to the discovery of any bugs in this example. No traces contain `S.discon` followed by `S.discon`, for example (or indeed any other erroneous violations of this typestate specification). The JIST specification yields fewer false positives because it more accurately represents the underlying `Session` typestate.

We conclude from these experiments that (1) the various techniques produce different kinds of specifications, in accordance with their assumptions about how programmers make mistakes and (2) not all of the assumptions underlying these miners were born out by this example (such as the assumption that typestate would be explicitly and simply represented or assumptions about event frequency). WML-static and JIST were both able to find an undocumented typestate specification. Their low false positive count shows that they were able to form specifications that were permissive enough to accept most client behaviors. Strauss, ECC and our technique were all good at yielding specifications that found bugs. Our technique found all bugs reported by other techniques and did so with the fewest false positives.

3.7.2 Bug Finding and Candidate Specification Ranking

We conducted experiments to compare our technique and the ECC technique on a subset of the benchmarks from Figure 2.13. The benchmarks were chosen for ease of comparison with the hand-written specifications from Section 2.5, and may favor the "*a* must be followed by *b*" specifications that both WN and ECC are designed to mine. We explicitly compare the bugs found via specification mining to the bugs found via the generic "library" specifications (i.e., `Stream`, `File` and "Database") reported in Figure 2.14. We

Program	Lines of Code	WN (our miner) Real Specs	WN (our miner) Bugs via Specs	ECC Real Specs	ECC Total Specs	ECC Bugs via Specs	Library Policy Bugs
`infinity`	28k	1 / 10	4	0 / 227	6468	0	14
`hibernate`	57k	9 / 51	93	3 / 424	9591	21	13
`axion`	65k	8 / 25	45	0 / 96	4159	0	15
`hsqldb`	71k	7 / 62	35	0 / 224	5032	0	18
`cayenne`	86k	5 / 35	18	3 / 311	8432	8	17
`sablecc`	99k	0 / 4	0	0 / 80	2506	0	3
`jboss`	107k	11 / 114	94	2 / 444	12852	4	40
`mckoi-sql`	118k	19 / 156	69	2 / 346	10860	5	37
`ptolemy2`	362k	9 / 192	72	3 / 656	23522	12	27
total	993k	69 / 649	430	13 / 2808	83422	50	172

Figure 3.12: Bugs found with specifications mined by ECC and our technique.

are unable to directly compare the other mining techniques because of the cost involved in manually specifying classes, predicates, and other parameters they require in advance.

Figure 3.12 presents our experimental results. The "Real Specs" column counts valid specifications (determined by manual inspection) against candidate specifications. For WN, all candidate policies were inspected. For ECC, only candidates with non-negative z-rank were inspected. Thus "3/424" means that 3 real specifications were found by manually inspecting the 424 candidate specifications with a non-negative ranking. The "Total Specs" column counts all policies reported by ECC. The "Bugs via Specs" column counts methods that violate the "Real Specs". Finally, the last column counts methods violating the generic "library"-based policies from Section 2.5 (it is the same as number reported in Figure 2.14).

ECC is able to find 4 specifications missed by our algorithm. In one of these examples, the b event never occurs in any error handling code (and thus does not meet our `ex` requirement). The ⟨`IndexStore.create`, `IndexStore.init`⟩ pair from `mckoi-sql` is one such policy. The `create` documentation states that the user "must call the `init` method after this is called." However, the `init` method never occurs in any exception handling code. Removing the `ex` requirement causes our algorithm to produce 1,114 candidate specifications for `hibernate` alone. Given the paucity of real specifications that are mistakenly filtered out by the requirement and the plethora of false positives that it avoids, we believe that basing our algorithm on exceptional control flow paths was a good decision.

Of the 69 real specifications we found, 24 involved methods from separate classes, arguing against class-based module requirements. Only one valid specification, ⟨`JDBCCommand.getConnection`, `java.sql.Connection.close`⟩ from `jboss`, involved methods from different libraries. On the other hand, for example, 30 of the first 100 false positive specifications reported by ECC for `axion` could have been avoided with our `sp` package-level module requirement. We believe these results argue strongly in favor of package-level requirements.

A common false positive repoted by the ECC technique paired the family of methods `ListIterator.hasNext` and `ListIterator.next`. The vast majority of paths that contain the former also contain the latter, and iterators occur frequently, causing the z-rank (whether restricted to normal traces or not) for such pairs to be high (`iterator` specifications occur as one of the top five candidates for ECC on six of our nine programs).

A common false positive for our technique paired `read` or `write` with `close` (instead of pairing `open` with `close`). As the ⟨`flush`, `close`⟩ data in Figure 3.4 demonstrate, "intermediate" work functions like `read` are almost invariably eventually followed by `close` if they are present, but the more desirable `open`-based specification usually ranks higher.

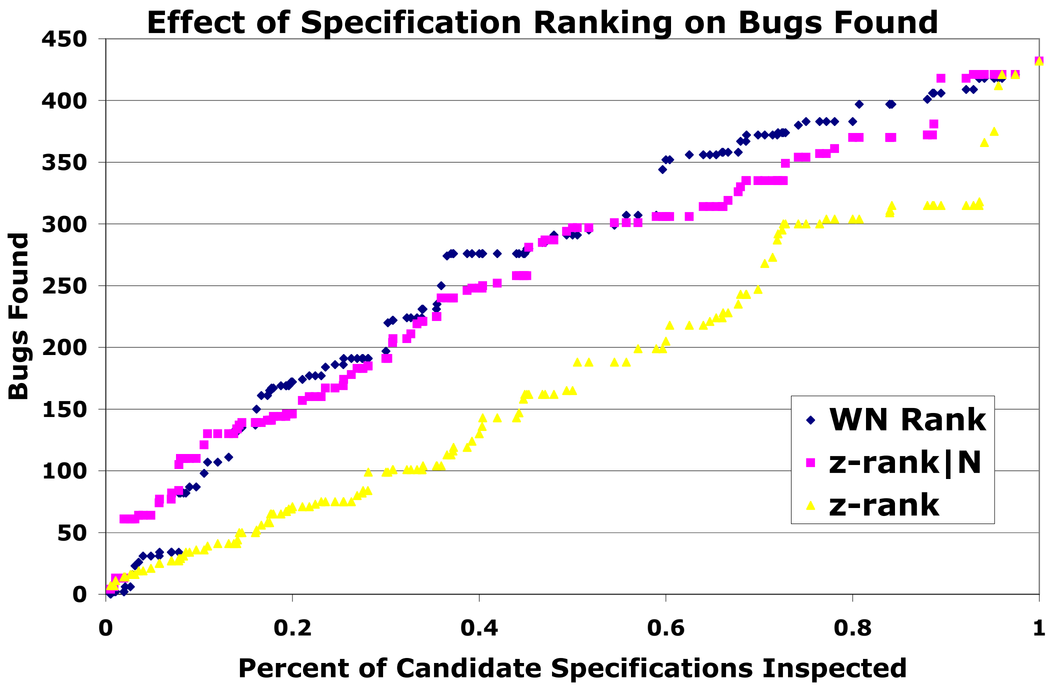


Figure 3.13: Effect of rank order on bug finding.

Almost every valid specification our technique found was listed somewhere in ECC's voluminous output of candidate specifications. For example, our 59th candidate `jboss` policy, which pairs `BeanLockSupport`'s `sync` and `releaseSync` methods, finds four real errors and is #9522 on the ECC list (z-rank$= -54$, z-rank$|_N= -29$).

Figure 3.13 shows the number of bugs found as a function of the ranking used to sort candidate specifications produced by our algorithm. Compared to the z-rank, our ranking ("WN Rank" is $N_{ab}/(N_a + N_{ab})$) only required 42% of the specifications to be inspected (instead of 72%) in order to find two-thirds of the bugs. The z-rank restricted to normal traces does better than the z-rank but worse than the WN rank. However, we conclude that since various rankings work only moderately better than a random shuffle, it is very important to produce a small number of extraneous candidates.

Our results for ECC are consistent with, but slightly better than, previously published figures in which 23 errors were found via specification mining on the Linux 2.4.1 kernel (about 840,000 lines of code) [ECC01]. ECC was designed to target C operating systems code. It actually performs better (in errors found per line of code) in this domain of Java programs than in their reported experiments, although there is no reason to believe that the bug density should be the same.

3.8 The Importance of Detected Bugs

One additional consideration is the utility of the found bugs. Even if a bug 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 bug. Commercial software ships with known bugs [LAZJ03]. Bugs 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" bugs to fix to keep programmers occupied.

The issue of bug quality is particularly important in this work. Our analysis finds bugs that show up in the presence of exceptional situations. We report resource leaks along paths that contain one or more exceptions or run-time errors. We must thus demonstrate that these bugs are a serious problem "in the real world" and are not just a theoretical possibility. Unfortunately, a thorough evaluation the importance of a bug is beyond the scope of this work and is typically situation-specific. Aspects such as the performance or security impact of a bug or the cost of fixing it can be difficult to measure quantitatively. We present some evidence to suggest that the bugs we report are important.

Our mining technique favors resource leaks and forgotten obligations. One of the authors of `ptolemy2` was willing to rank bugs we found on his own five point scale. For that program, 11% of the bugs we reported were in tutorials or third-party 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 bugs that rated a 4 or 5 were fixed immediately. 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 in order to determine whether the bugs found by our analysis were important enough to fix. The direct experiment of finding bugs, reporting them to developers and then counting how many are fixed is difficult to perform, especially in the open-source community. For example, it is generally agreed upon that social aspects like “having a champion for your tool inside the development organization” [DLS02] and “not reporting too many bugs at once” [ECC01] play a large role in determining whether reported bugs are fixed. Instead of comparing the present against the future, we compare the past against the present.

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 noted the first 100 bugs reported. Without reporting any of the bugs to `eclipse` programmers we then looked for each of those bugs 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 bugs found by our tool in `eclipse 2.0.0` had been fixed by `eclipse 3.0.1`. Between those version `eclipse` underwent many refactorings so manual inspection was necessary because the buggy code had often moved from one class to another. Given our stated goal of improving software quality by finding and fixing bugs 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 bugs it reports would have to be fixed later (and presumably at least ten times more expensively, see Section 1.1) anyway.

Figure 3.14 shows the `eclipse 2.0.0` code for the `parseInstalledPlugin` method of the `update.internal.core.SiteFileFactory` class. The bug is that the `FileInputStream` created on line 17 may not be freed. The bug is not trivial to fix because the `FileInputStream` is anonymous. A new variable must be introduced to hold it and a care must be taken to `close` that variable later. In Figure 3.15 a new local variable called "`in`" has been introduced on line 11 to hold the `FileInputStream`. The call on line 17 of Figure 3.14 corresponds to lines 22–23 of Figure 3.15. A `finally` clause has been added on line 37 and "`in`" is closed (if it is not `null`, see Figure 2.5 for other examples of this idiom) on line 38. The bug was fixed on March 2, 2004 when version 1.57 of that file was checked in by user `kkolosow` with the comment "close InputStream" [Ecl03].

We speculate that the remainder of the reported bugs were not fixed because they were difficult to track down. Trishul Chilimbi of Microsoft Research reports that only 10%

```
01: private void parseInstalledPlugin(File dir) throws CoreException {
02:   PluginIdentifier plugin = null;
03:   File pluginFile = null;
04:   try {
05:     if (dir.exists() && dir.isDirectory()) {
06:       File[] files = dir.listFiles();
07:       DefaultPluginParser parser = new DefaultPluginParser();
08:       for (int i = 0; i < files.length; i++) {
09:         if (files[i].isDirectory()) {
10:           if (!(pluginFile = new
11:               File(files[i], "plugin.xml")).exists()) {
12:             pluginFile = new File(files[i], "fragment.xml");
13:           }
14:           if (pluginFile != null && pluginFile.exists() &&
15:               !pluginFile.isDirectory()) {
16:             IPluginEntry entry = parser.parse(
17:               new FileInputStream(pluginFile) );
18:             VersionedIdentifier identifier =
19:               entry.getVersionedIdentifier();
20:             plugin = new PluginIdentifier(identifier, files[i],
21:               entry.isFragment());
22:             addParsedPlugin(plugin);
23:           }
24:         } // files[i] is a directory
25:       }
26:     } // path is a directory
27:   } catch (IOException e) {
28:     String pluginFileString = (pluginFile==null) ? null :
29:       pluginFile.getAbsolutePath();
30:     throw Utilities.newCoreException(
31:         Policy.bind("SiteFileFactory.ErrorAccessing",
32:           pluginFileString), e);
33:   } catch (SAXException e) {
34:     String pluginFileString = (pluginFile==null) ? null :
35:       pluginFile.getAbsolutePath();
36:     throw Utilities.newCoreException(
37:         Policy.bind("SiteFileFactory.ErrorParsingFile",
38:           pluginFileString), e);
39:   }
40: }
```

Figure 3.14: `eclipse 2.0.0` with `SiteFileFactory` bug (line 17)

```
01: private void parseInstalledPlugins(File pluginsDir)
02:     throws CoreException {
03:   if (!pluginsDir.exists() || !pluginsDir.isDirectory())
04:     return;
05:   File[] dirs = pluginsDir.listFiles(new FileFilter() {
06:     public boolean accept(File f) { return f.isDirectory(); }
07:   });
08:   DefaultPluginParser parser = new DefaultPluginParser();
09:   for (int i = 0; i < dirs.length; i++) {
10:     File pluginFile = new File(dirs[i], "META-INF/MANIFEST.MF");
11:     InputStream in = null;
12:     try {
13:       BundleManifest bundleManifest = new BundleManifest(pluginFile);
14:       if (bundleManifest.exists()) {
15:         PluginEntry entry = bundleManifest.getPluginEntry();
16:         addParsedPlugin(entry, dirs[i]);
17:       } else {
18:         if (!(pluginFile = new File(dirs[i], "plugin.xml")).exists())
19:             pluginFile = new File(dirs[i], "fragment.xml");
20:         if (pluginFile != null && pluginFile.exists()
21:             && !pluginFile.isDirectory()) {
22:           in = new FileInputStream(pluginFile);
23:           PluginEntry entry = parser.parse(in);
24:           addParsedPlugin(entry, dirs[i]);
25:         }
26:       }
27:     } catch (IOException e) {
28:       String pluginFileString = (pluginFile == null)
29:         ? null : pluginFile.getAbsolutePath();
30:       throw Utilities.newCoreException(Policy.bind(
31:         "SiteFileFactory.ErrorAccessing", pluginFileString), e);
32:     } catch (SAXException e) {
33:       String pluginFileString = (pluginFile == null)
34:         ? null : pluginFile.getAbsolutePath();
35:       throw Utilities.newCoreException(Policy.bind(
36:         "SiteFileFactory.ErrorParsingFile", pluginFileString), e);
37:     } finally {
38:       if (in != null) try { in.close(); } catch (IOException e) { }
39:     }
40:   }
41: }
```

Figure 3.15: `eclipse 3.0.1` with `SiteFileFactory` bug fixed (line 38)

of resource leaks (i.e., the sorts of bugs reported by this analysis) are always resolved and that 90% of the leaks that are resolved take more than three days to resolve [HC04]. That is, these bugs of this form are by nature difficult to track down and fix. This argues strongly for static tools that can pinpoint such bugs automatically.

It is difficult to obtain numbers indicating what fraction of the bugs reported were later fixed for various bug-finding research projects. Our two experiments suggest that 43% of the bugs we report are considered real by developers. As one external datapoint, the FindBugs project [HP04] 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 bugs.

3.9 Specification Mining Conclusions

Using our mining algorithm to find bugs was decidedly better than using generic library policies. We found 430 bugs using mined policies compared to 172 using generic ones. We found 380 more bugs and 56 more policies than ECC using 2000 fewer candidate specifications. Our experiments highlighted the practical importance of our algorithmic assumptions, in particular our use of exceptional control flow.

As automatic program verification tools become more prevalent, specifications become the limiting factor in verification efforts, and specification mining for the purposes of finding bugs becomes more important. Given a program, a specification miner emits candidate policies that describe real or common program behavior. We proposed a novel miner that uses information about exceptional paths. We compare the bug-finding power of various miners. In 1 million lines of Java code, we found 430 bugs using mined specifications compared to 172 using generic “library”-based ones, and we found more bugs than comparable mining algorithms. Our experiments highlighted the relative unimportance of candidate ranking and the practical importance of our algorithmic assumptions, in particular our use of exceptional control flow for specification mining.

Now that we have found a large number of bugs in programs the next chapter deals with fixing those bugs.

PL people try to show that things that work in practice also work in theory.

Jim Ezick

Chapter 4

Language Features To Address Defects

In this chapter we characterize the mistakes found by the analysis from Chapter 2. Based on those mistakes we claim that existing language features for exception handling (e.g., `try-finally` blocks) are ill-suited for handling certain classes of resources in the presence of run-time errors. We propose a new feature, the compensation stack, to address the sorts of mistakes uncovered by our analysis. We provide two case studies to demonstrate the benefits of our compensation stacks. Finally, we discuss other related attempts to address similar problems. Some of the work presented in this chapter was previously described in an earlier publication [WN04].

4.1 Defect Characterization

In this section we characterize some of the mistakes found by our analysis, paying special attention to the qualities an exception handling mechanism should have in order to address these bugs more naturally. We use these observations to guide the design of our new language feature.

In some cases, `try-finally` handling is skipped entirely, 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 annotation on line 1 and extant handling in other methods from the same program show that the programmer is aware of the possibility of exceptional situations. 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. That is, it would be nice to provide minimal guarantees even in the case where the programmer is not thinking about exception handling.

It is also common for `try-finally` statements to protect some, but not all, operations, as in this fragment from `staf`'s `STAXMonitor` class:

```
01: ObjectInputStream ois = null;
02: try {
03:   ois = new ObjectInputStream(/* ... */);
04:   // ...
05: } catch (StreamCorruptedException ex) {
06:   if (ois != null) { ois.close(); }
07:   showErrorDialog(/* ...  */);
08:   return false;
09: }
10: Object obj = ois.readObject();  // no try
11: ois.close();                    // no finally
```

Care is taken to deal with run-time errors that occur on lines 3–4 when `ois` is created and used, but reading from `ois` on line 10 is done without an enclosing `try-finally`. These examples show that it would be useful to have a mechanism that allows fine-grained control for some error handling but automatic behavior for others.

In the rest of the discussion we will abstract away the resource names and the unrelated application code for clarity. We assume that important resource `x` is acquired with a call to `open_x` and released with a call to `close_x`. As with the safety policies in Section 2.5 we assume that if `open_x` is called then `close_x` must be called and that `close_x` cannot be called unless `open_x` has been called. We also assume that `open_x`, `close_x` and `work` (representing application logic) can signal exceptions. Thus the previous example would be rendered:

```
01: try {
02:   open_1();
03:   work();
04: } catch {
05:   close_1();
06:   return;
07: }
08: work();
09: close_1();
```

In `osage` multiple methods use this form:

```
01: try {
02:   open_1();
03:   open_2();
04: } finally {
05:   close_2();
06:   close_1();
07: }
```

If an exception is raised on line 2, control will jump to line 5 and execute `close_2` before `open_2`. Typically this causes a `null`-pointer exception but it can have additional effects depending on the resource in question. For example, multiple releases on poorly-written (or efficiently-written) locks can lead to unprotected resources later. If an exception is raised on line 3, control will jump to line 5 and execute `close_2` before `open_2` and in addition `close_1` will never be executed. Finally, if an exception is raised on line 5 then `close_1` will never be executed. A single project will often re-use an error-handling design pattern that contains flaws. A cut-and-paste approach is particularly common in JDBC applications (i.e., those that should adhere to the “Database” policy from Section 2.5).

Some programs, like `compiere`, treat multiple resources sequentially but still fail to handle errors perfectly:

```
01: open_1();
02: work();
03: close_1();
04: open_2();
05: work();
06: close_2();
```

Here an exception raised on line 2 will cause `close_1` to be skipped and an exception raised on line 5 will cause `close_2` to be missed. As with the `axion` example above, it would be convenient to have a language feature that worked even if `try` and `finally` were not present.

The `quartz` program contains a number of instances of:

```
01: try {
02:   open_1();
03:   open_2();
04:   work();
05: } finally {
06:   close_1();
07: }
```

Such partial handling covers some of the resources, but not all. There is no corresponding `close_2` for the `open_2`.

The `ohioedge` program contains examples like:

```
01: for (iterator) {
02:   open_1();
03:   work();
04:   close_1();
05: }
```

Such handling can be difficult to reason about statically, especially if the important resources are not variables local to the loop body. Such code typically iterates over all of the elements in a collection, for example by acquiring a lock on each element of a linked list before mutating its contents. If an exception occurs on line 3 in the middle of the loop then the lock is leaked for an element in the middle of the list, yielding a difficult-to-debug deadlock.

Various programs often use flags (and often use them correctly) to track resources and free them early:

```
01: try {
02:   open_1(); flag = 0;
03:   work();
04:   if (...) {
05:     flag = 1; close_1();
06:   }
07:   work();
08: } finally {
09:   if (!flag) { close_1(); }
10: }
```

Here the variable `flag` keeps track of whether or not the resource needs to be freed. When `flag` is 1 the resource has already been released. This coding practice is very common for contentious resources like database locks. If the `work` on line 7 takes time to execute it is worth releasing the resource early if possible.

In many cases, like the example in Section 2.1, error handling with multiple resources contains an insufficient number of `try` statements to handle all paths. One common approach to handling this problem is to introduce a flag variable (or check individual objects against `null`), as the following examples (adapted from [BP03]) illustrate:

```
01: int flag = 0;
02: try {
03:   open_1(); flag = 1;
04:   work();
05:   open_2(); flag = 2;
06:   work();
07:   open_3(); flag = 3;
08:   work();
09: } finally {
10:   switch (flag) {
11:     case 3: try { close_3(); } catch (Exception e) {}
12:     case 2: try { close_2(); } catch (Exception e) {}
13:     case 1: try { close_1(); } catch (Exception e) {}
14:   }
15: }
```

Here the variable `flag` tracks the program's progress through the method. When `flag` is n, resources 1–n have been acquired and must be released. Note that in the `switch` statement on lines 10–14 control will "fall though" from case 3 to case 2 and from case 2 to case 1.

This "counting `flag`" approach has a number of software engineering disadvantages. One is that the cleanup code is distant from the action code. Another is that control-flow that determines the actions must be duplicated in reverse for the cleanup. Every distinct path of normal control flow must have a corresponding path in the exceptional error-handling control flow. The following code fragment demonstrates this complexity:

```
01: int flag = 0, did_two = 0;
02: try {
03:   open_1(); flag = 1;
04:   work();
05:   if (...) { open_2(); flag = 2; did_two = 1; }
06:   work();
07:   open_3(); flag = 3;
08:   work();
09: } finally {
10:   switch (flag) {
11:     case 3: try { close_3(); } catch (Exception e) {}
12:     case 2: if (did_two) { try { close_2(); } catch (Exception e) {} }
13:     case 1: try { close_1(); } catch (Exception e) {}
14:   }
15: }
```

Here the second resource is only acquired in some cases and additional variables (`did_two`) and run-time checks (line 12) must be added. Adding a `for` loop instead of an `if` statement on line 5 would require bookkeeping to determine exactly how far through the loop the code had advanced. We would prefer to automate such bookkeeping whenever possible.

Standard attempts to deal with resources in the presence of exceptional situations introduce additional logic into the program that must be maintained (and reproduced at every resource use). If the control-flow is non-trivial (e.g., a `while` loop or a visitor that performs actions on btree elements) it might not even be desirable to reproduce the control flow (e.g., in the btree case it would involve jumping to the middle of the tree and then traversing it in reverse). In such general cases it makes more sense to record which actions were taken at run-time and then clean up exactly what is required. A mechanism that does not require the programmer to reproduce control flow or introduce extra bookkeeping is desired here. In the next section we will examine destructors and finalizers, which are modern programming language features that could be used to address such concerns, and argue that they are not sufficient.

4.2 Destructors and Finalizers

Destructors and finalizers are 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++ [Str91] but are also present in other languages like C# [HWG03]. 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 static 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 generalize the notion of 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 e.g., Gay and Aiken [GA98]). 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 error 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 4.4.

A *finalizer* is another special method associated with a class. Finalizers are typically used with Java [GJS96] but are also present in other languages like C# [HWG03]. 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'H05]. In contrast, the elements of the "Database" policy from Section 2.5 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. 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 [GJS96]. 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. 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 in order 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 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 2.8.

Finally, it is worth noting that Java programmers do not make even a sparing use of finalizers to address these problems. Some Java implementations do not implement finalizers correctly [Boe03], 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 2.8, there were only 13 user-defined finalizers (`hibernate` had 4; `osage` had 3; `jboss` and `eclipse` had 2; `javad` and `aspectj` had 1). In our experience, Java programmers basically do not use finalizers. One might also hope that standard libraries would make use of finalizers, but this is not always the case. The GNU Classpath 0.05 implementation of the Java Standard Library does not use finalizers for any of the resources governed by the safety policies in Section 2.8. 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 Standard Libraries may well use finalizers for all such important resources, one cannot currently portably count on the Library to do so. 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.

The results in Section 2.8 argue that language support is necessary: merely making a better `Socket` library will not help if `Socket`s, 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 mistakes presented here could have been avoided using our proposed language extension. In addition, the analysis presented in Section 2.7 could easily verify that programs using our mechanism are handling these resources correctly.

4.3 Compensation Stacks

Based on our characterization of existing mistakes and coding practices in Section 4.1 and existing programming language techniques in Section 4.2, we propose a language extension where program actions and interfaces are 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.

Our compensation stacks are an adaptation of the database notions of *compensating transactions* and *linear sagas* [GMS87]. 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 $a_1 ... a_n$ with compensating transactions $c_1 ... c_n$. This system guarantees that either $a_1 ... a_n$ executes or $a_1 ... a_k c_k ... c_1$ executes. Note that the compensations are applied in reverse order. We have found this model to be a good fit for this sort of run-time error handling. Many conceptually simple program actions actually require that multiple resources be handled in sequence.

Our system allows programmers to link 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. If a compensating action raises an exception while executing, the exception is logged but compensation execution continues.[1] When a compensation terminates (either normally or exceptionally), it is removed from the compensation stack.

Compensation stacks normally behave like generalized 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

[1]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, error-prone 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`).

executed. The ability to execute 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 its code (presumably based on outside knowledge not directly encoded in the safety policy). This flexibility allows compensations that truly undo effects to be avoided on successful executions, and it requires that the programmer annotate a small number of success paths rather than every possible error path. Additional compensation stacks may be declared to create a "nested transaction" effect. Finally, the analysis in Section 2.7 can be easily modified to show that programs that make use of compensation stacks do not forget obligations.

4.4 Compensation Stack Implementation

We implemented compensation stacks using a source-level transformation for Java programs. This entails defining a `CompensationStack` class, adding support for closures (as in Odersky and Wadler [OW97]), and adding convenient syntactic sugar for lexically-scoped compensation stacks.

In our system, the client code from Figure 2.2 looks like this:

```
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. We will now elaborate those details and develop our system to the point where such code behaves correctly along all paths.

The first step in such an 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 `Connection`s 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);
}
```

As we mentioned in Section 4.2, 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 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 the six-line version at the beginning of this section for the common case.

Compensations can contain arbitrary code, not just method calls. For example, consider this code fragment adapted from [BP03]:

```
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 */ }
  ... // do work 2
} finally {
  S.run();
}
```

Resource finalization and state changes are thus 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 simply method invocations.

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 2.4 and Section 2.8).

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. With this feature, we found it easy to avoid the mistakes that were reported hundreds of times in Section 2.8. In the common case of a lexically-scoped linear saga of resources, the 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.

4.5 Case Studies

We hand-annotated two programs to show that it is easy to modify existing programs to use compensation stacks (and by implication that it would not be difficult to write a new program from scratch using them) and to demonstrate that the run-time overhead is low. Guided by the dataflow analysis in Section 2.7, the programs were modified so that their existing error-handling made use of compensation stacks; no truly new 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 `Socket`s 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 first case study, Aaron Brown's undo-able email store [BP03], 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., `socket`s and databases). The resulting program was 225 lines shorter (about 1%) because redundant 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. Using fifty micro-benchmarks and one example workload (all provided by the original author), the annotated program's performance was almost identical to the original. Performance was measured to be within one standard deviation of the original, and was generally within one half of a standard deviation; the run-time overhead associated with keeping track of obligations at run-time was dwarfed by I/O and other processing times. Compensations were used to handle every request answered by the program. Finally, by changing a method invocation in some insufficiently-guarded cleanup code to always raise one of its declared run-time errors 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 failure correctly and proceeded normally. While this sort of targeted fault injection is hardly representative, it does show that the errors we are addressing with compensations can have an impact on reliability.

The second case study, Sun's `Pet Store 1.3.2` [Sun01], 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 `Connection`s. Using an independent workload [CKF+02, CDCF03], 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 `RelationSet`s 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.[2] 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

[2]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.

the annotated program is 43.44 ms with a standard deviation of 77 ms. The annotated program is both 17% faster and also more consistent because less middleware intervention was necessary.

Together, these case studies suggest that stacks of compensations are a natural and efficient model for this sort of run-time error handling. The decrease in code size argues that common idioms are captured nicely by this formalism and that there is a software engineering benefit to associating error handling with interfaces. The unchanging or improved performance indicates that leaving some checks to run time is quite reasonable. Finally, the checks ensure that cleanup code is invoked correctly along all paths through the program.

4.6 Related Work

Related work falls into six broad categories: approaches to cleaning up resources, type systems, regions, exception schemes, ideas on error handling, and transactional models.

4.6.1 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 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 error-handling (like that of `undo` in Section 4.5) 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 [vdW02]. 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 `cleanup`s as "a structured way to deal with things that need to be done later." [SPS02] `Cleanup`s are executed when `gdb` commands are finished, when an error occurs, or on explicit request. A `cleanup` is a chain of function pointers and arguments. In this example:

```
struct cleanup *old = make_cleanup (null_cleanup, 0);
data = xmalloc (sizeof blah);   // acquire resource
make_cleanup (xfree, data);     // promise to free resource later
/* do dangerous work */
do_cleanups (old);              // free resource now
```

A `cleanup` chain is used to ensure that the allocated memory returned by `xmalloc` is eventually freed by passing it to `xfree`. `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.

4.6.2 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 [DF01] 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, and 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 [FD02] 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 since we want the explicit aliasing of storing a reference to the resource in the compensation stack and allowing the program to continue to manipulate it.

4.6.3 Regions

Region-based memory management aims for both the predictability and efficiency of stack-based allocation and the safety of automatic memory management. Care must be taken to avoid freeing a region too early lest the program follow a dangling reference.

Gay and Aiken [GA98] propose a system for memory management using explicit first-class regions [TT97]. Their regions are conceptually similar to our compensation stacks. In their system, reference counts keep programs from deleting regions too early. In our system, stacks keep programs from forgetting to perform compensations. Regions allow one to express data locality, whereas Putting compensations in the same stack allows the programmer to express the conceptual locality of a compound transaction. Later, Gay and Aiken [GA01] discuss a system with support for subregions. In it a region may only be freed if it has no remaining subregions.

Tofte and Talpin [TT97] describe the general theory of region-based memory management and present a region inference algorithm for garbage-collected programs. Region inference provides automatic memory management, broadly decides object lifetimes at compile-time and specializes memory management to the given program. In their system, the store is organized as a stack of regions with nested lifetimes.

Hallenberg et al. [HET02] give a system for combining garbage collection and such region-based memory management. They report that region inference works "most of the time" and that garbage collection (i.e., run-time checking) holds the promise of eliminating the need to hand-tune programs to work well with regions. A type-and-effect system is used to statically ensure that objects are allocated in regions such that no dangling references are ever followed.

4.6.4 Exceptions

Most modern programming languages feature *exceptions* that behave according to the *replacement model* [Goo75, YB85] (see also [Lev77, DHL90, MT97, HA98, RM99]). Alonso et al. [AHA$^+$00] believe that poor support for exception handling is a major obstacle for large-scale and mission-critical systems. Hagen et al. [HA00] 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 [Don01] 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.

Cargill [Car94] 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 errors are handled correctly. Our technique is particularly well-suited to handling the matched acquire-free behavior in his presentation.

4.6.5 Error handling

Quite a bit of attention from a number of research communities has been devoted to issues of 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.

Valetto and Kaiser [VK02] note that adaptation to 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 [CD99] 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 [DR03] 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 errors in compensation code.

The VINO operating system [SESS96] 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 error 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.

4.6.6 Transactions

Database transactions provide a strong and well-founded approach to error handling [Gra81]. However, many find the consistency and durability of transactions to be too heavyweight for most programming purposes (e.g., [AHA$^+$00, LS83, DHL90]). For example, Java programs that want transactional support for certain pieces of data (e.g., e-commerce applications looking updating inventory tables) often make explicit calls an external database (as in the "Database" policy of Section 2.5). 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. [BBG$^+$89] describe a checkpointing system that allows unmodified programs to survive hardware failures. Essentially, every system call is intercepted and logged. Others [SW91, LC98, SSF99] provide similar services. Our compensation annotations are a much less drastic change to the program semantics than the incorporation of full-fledged transactions.

In addition, these transaction techniques address an orthogonal error handling issue. In Borg et al.'s system, a buggy process that acquires a lock twice and deadlocks on initialization will continue to deadlock no matter how many times it is recovered. Lowell et al. [LCC00] formalize this point by noting that the desire to log all events actually conflicts with the ability to recover from all 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 bugs cause crashes that would not be prevented by a failure-transparent operating system. Our technique hopes to address those sorts of bugs, although it is less automatic.

Many researchers have found that advanced transactional concepts fit closely with language-level error handling. [DDN$^+$98, LOLZ01] One such concept, the compensating transaction, semantically undoes the effects of another transaction *after* that transaction has been committed [KLS90]. Designing a full compensating transaction that completely undoes the effects of a previous action is often difficult. Our system relaxes this requirement. Alonso et al. [AKA$^+$94] consider the notion of *linear sagas* [GMS87] in a similar context. Our system is slightly more general than a pure linear saga [KLS90] and more closely resembles a form of nested or interleaved linear sagas.

4.7 Language Feature Conclusions

We examined the mistakes found by our bug-finding analysis qualitatively and discussed the strengths and weaknesses of exceptions, destructors and finalizers for run-time error handling. Based on our observations we have proposed a programming language feature based on the notions of compensating transactions and linear sagas. Stacks of compensations are first-class objects that can be manipulated by the program and used to store compensating actions. They behave much like destructors but provide a more general stack structure, guarantees on heap objects, and easy early execution and bookkeeping. Compensations themselves are recorded and executed at run time. Case studies show that such a feature can be used to achieve improved reliability with minimal overhead. Since error handling is a large and important part of programs, suggesting features that would help to prevent them is an important step toward making more robust programs.

I can only conclude that I'm paying off
karma at a vastly accelerated rate.

Susan Ivanova, "Points of Departure"

Chapter 5

Conclusion

Software reliability remains an important and expensive issue. This work presents a three-point approach for addressing a certain class of software reliability problems. We focus on exceptional situations, a previously-underinvestigated aspect of software reliability.

First, we presented a static dataflow analysis for finding bugs in how programs deal with important resources in the presence of exceptional situations. In order to find bugs in programs we formalized some initial specifications of what the programming should be doing. In order to find bugs 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 in in order to make the analysis easier to use by eliminating false positives. The analysis found over 800 methods with mistakes in almost 4 million lines of Java code.

Second, we returned to the problem of specifying what the program should be doing. We presented an algorithm for automatically mining program-specific partial correctness specifications. The specifications took the form of event pairs that should be called in sequence. Our specification miner was based on our observations of how programs behave in exceptional situations. In a series of qualitative and quantitative experiments our miner was able to find more real specifications that could be used to find more real bugs than other comparable existing miners. Our mined specifications found 430 bugs in one million lines of code compared to 172 bugs from our hand-written policies and 50 bugs from another miner. Our miner also produced an order of magnitude fewer false positives than a comparable approach. In addition, we presented evidence to suggest that 43% of the bugs found by our analysis would be fixed by developers.

Third, given over 1,200 resource-handling bugs in exceptional situation we designed a language feature to make it easier to fix such mistakes. We characterized the errors found by our analysis and determined that existing language features were insufficient. We proposed that programmers keep track of important obligations at run-time in special compensation stacks. In two case studies we showed that it is easy to apply compensation stacks to existing Java programs and that they can be used to make programs slightly simpler and, in some cases, slightly more reliable.

We believe this work was a successful step toward making software more reliable in the presence of exceptional situations. Given a Java program we can automatically infer specifications local to that program. Using those specifications and some hand-crafted ones we can analyze the program to find mistakes. Once mistakes have been located we can provide the programmer with an easy-to-use tool for addressing them. All of this can be done cheaply, before the program is deployed. We hope that this approach, or techniques like it, will be more frequently adopted in the future.

Bibliography

[ABL02] Glenn Ammons, Rastislav Bodik, and James R. Larus. Mining specifications. In *Principles of Programming Languages*, pages 4–16, 2002.

[ACMN05] Rajeev Alur, Pavol Cerny, P. Madhusudan, and Wonhong Nam. Synthesis of interface specifications for Java classes. In *Principles of Programming Languages*, 2005.

[Adv03] Advisor. Beware: 10 common web application security risks. Technical Report Doc 11756, Security Advisor Portal, January 2003.

[AHA$^+$00] G. Alonso, C. Hagen, D. Agrawal, A. El Abbadi, and C. Mohan. Enhancing the fault tolerance of workflow management systems. *IEEE Concurrency*, 8(3):74–81, July 2000.

[AKA$^+$94] G. Alonso, M. Kamath, D. Agrawal, A. El Abbadi, R. Gunthor, and C. Mohan. Failure handling in large-scale workflow management systems. Technical Report RJ9913, IBM Almaden Research Center, San Jose, CA, November 1994.

[AMBL03] Glenn Ammons, David Mandein, Rastislav Bodik, and James Larus. Debugging temporal specifications with concept analysis. In *Programming Language Design and Implementation*, San Diego, California, June 2003.

[Ash04] Derek Ashmore. Best practices for JDBC programming. *Java Development Journal*, 9(10), October 2004.

[ASU86] Alfred Aho, Ravi Sethi, and Jeffrey Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1986.

[BAr81] John F. BArtlett. A NonStop kernel. Technical Report PN87603, Tandem Computers, June 1981.

[BBG$^+$89] Anita Borg, Wolfgang Blau, Wolfgang Graetsch, Ferdinand Herrmann, and Wolfgang Oberle. Fault tolerance under UNIX. *ACM Transactions on Computer Systems*, 7(1), February 1989.

[BKL$^+$04] Philip Baldwin, Sanjeev Kohli, Edward A. Lee, Xiaojun Liu, and Yang Zhao. Modeling of sensor nets in Ptolemy II. In *Proceedings of Information Processing in Sensor Networks*, April 2004.

[Bla02] Rex Black. Investing in software testing: The cost of software quality. Technical report, 2002.

[Boe03] Hans-J. Boehm. Destructors, finalizers and synchronization. In *Principles of Programming Languages*. ACM, January 2003.

[BP03] A. Brown and D. Patterson. Undo for operators: Building an undoable e-mail store. In *USENIX Annual Technical Conference*, 2003.

[BR01] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 2001, Workshop on Model Checking of Software*, volume 2057 of *Lecture Notes in Computer Science*, pages 103–122, May 2001.

[Car94] Tom Cargill. Exception handling: a false sense of security. *C++ Report*, 6(9), 1994.

[CD99] Luca Cardelli and Rowan Davies. Service combinators for web computing. *Software Engineering*, 25(3):309–316, 1999.

[CDCF03] George Candea, Mauricio Delgado, Michael Chen, and Armando Fox. Automatic failure-path inference: A generic introspection technique for internet applications. In *IEEE Workshop on Internet Applications*. San Jose, California, June 2003.

[CDW04] Hao Chen, Drew Dean, and David Wagner. Model checking one million lines of C code. In *Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2004.

[CKF$^+$02] Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: Problem determination in large, dynamic Internet services. In *International Conference on Dependable Systems and Networks*, pages 595–604. IEEE Computer Society, 2002.

[Cri87] F. Cristian. Exception handling. Technical Report RJ5724, IBM Research, 1987.

[CWH00] Mary Campione, Kathy Walrath, and Alison Huml. *The Java Tutorial.* Addison-Wesley Professional, 2000.

[DDN$^+$98] Asit Dan, Daniel M. Dias, Thao Nguyen, Marty Sachs, Hidayatullah Shaikh, Richard King, and Sastry Duri. The Coyote project: Framework for multi-party e-commerce. In *ECDL*, volume 1513 of *Lecture Notes in Computer Science*, pages 873–889. Springer, 1998.

[DF01] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Programming Language Design and Implementation*, pages 59–69, 2001.

[DGK96] Srinivas Devadas, Abhijit Ghosh, and Kurt Keutzer. An observability-based code coverage metric for functional simulation. In *ICCAD*, pages 418–425, 1996.

[DHL90] U. Dayal, M. Hsu, and R. Ladin. Organizing long-running activities with triggers and transactions. In *Proceedings of ACM SIGMOD*, pages 204–214. Atlantic City, May 1990.

[dLF02] Rogério de Lemos and José Luiz Fiadeiro. An architectural support for self-adaptive software for treating faults. In *Proceedings of the First ACM SIGSOFT Workshop on Self-Healing Systems (WOSS '02)*, pages 39–42. Charleston, South Carolina, November 2002.

[DLS02] Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: path-sensitive program verification in polynomial time. *SIGPLAN Notices*, 37(5):57–68, 2002.

[Don01] Christophe Dony. A fully object-oriented exception handling system. In *Advances in Exception Handling Techniques*, volume 2022 of *Lecture Notes in Computer Science*, pages 18–38, 2001.

[DR02] Brian Demsky and Martin Rinard. Automatic detection and repair of errors in data structures. Technical Report MIT-LCS-TR-875, MIT, December 2002.

[DR03] Brian Demsky and Martin C. Rinard. Automatic data structure repair for self-healing systems. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 2003.

[ECC01] Dawson R. Engler, David Yu Chen, and Andy Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *Symposium on Operating Systems Principles*, pages 57–72, 2001.

[ECCH00] Dawson Engler, Ben Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Symposium on Operating Systems Design and Implementation*, 2000.

[Ecl03] Eclipse.org. Eclipse platform technical overview. http://eclipse.org. Technical report, 2003.

[ELLR90] A. K. Elmagarmid, Y. Leu, W. Litwin, and M. E. Rusinkiewicz. A multi-database transaction model for Interbase. In *The 16th Annual International Conference on Very Large Data Bases*, August 1990.

[Els03] Martin Elsman. Garbage collection safety for region-based memory management. In *Proceedings of ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI'03)*. ACM Press, January 2003.

[FD02] Manuel Fähndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In *ACM Conference on Programming Language Design and Implementation*, June 2002.

[FPP98] David Freedman, Robert Pisani, and Roger Purves. *Statistics*. W. W. Norton, 1998.

[FRMW04] C. Fu, B. Ryder, A. Milanova, and D. Wannacott. Testing of java web services for robustness. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2004.

[GA98] David Gay and Alexander Aiken. Memory management with explicit regions. In *Programming Language Design and Implementation*, pages 313–323, 1998.

[GA01] David Gay and Alexander Aiken. Language support for regions. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 70–80, 2001.

[GJS96] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, 1996.

[GMS87] Hector Garcia-Molina and K. Salem. Sagas. In *ACM Conference on Management of Data*, pages 249–259, 1987.

[Goo75] John B. Goodenough. Exception handling: issues and a proposed notation. *Communications of the ACM*, 18(12):683–696, 1975.

[Gra81] Jim Gray. The transaction concept: virtues and limitations. In *International Conference on Very Large Data Bases*, pages 144–154. Cannes, France, September 1981.

[GRRX01] Allessandro Garcia, Cecília Rubira, Alexander Romanovsky, and Jie Xu. A comparative study of exception handling mechanisms for building dependable object-oriented software. *Journal of Systems and Software, Elsevier*, 59(2):197–222, November 2001.

[GS02] David Garlan and Bradley Schmerl. Model-based adaptation for self-healing systems. In *Proceedings of the First ACM SIGSOFT Workshop on Self-Healing Systems (WOSS '02)*, pages 27–32. Charleston, South Carolina, November 2002.

[HA98] Claus Hagen and Gustavo Alonso. Flexible exception handling in the OPERA process support system. In *International Conference on Distributed Computing Systems*, pages 526–533, 1998.

[HA00] Claus Hagen and Gustavo Alonso. Exception handling in workflow management systems. *IEEE Transactions on Software Engineering*, 26(9):943–959, September 2000.

[HC04] Matthias Hauswirth and Trishul Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *Proceedings of the Symposium on*

Architectural Support for Programming Languages and Operating Systems (ASPLOS), October 2004.

[HET02] Niels Hallenberg, Martin Elsman, and Mads Tofte. Combining region inference and garbage collection. In *SIGPLAN Conference on Programming Language Design and Implementation*, June 2002.

[Hib04] Hibernate. Object/relational mapping and transparent object persistence for Java and SQL databases. In *http://www.hibernate.org/*, July 2004.

[HJMS02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *Principles of Programming Languages*, pages 58–70, 2002.

[HL02a] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *International Conference on Software Engineering*, May 2002.

[HL02b] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *International Conference on Software Engineering*, pages 291–301. ACM Press, 2002.

[HME03] M. Harder, J. Mellen, and M. Ernst. Improving test suites via operational abstraction. In *Proceedings of the International Conference on Software Engineering*, May 2003.

[HMU00] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (2nd Edition)*. Addison-Wesley, November 2000.

[HP04] David Hovemeyer and William Pugh. Finding bugs is easy. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 132–136, New York, NY, USA, 2004. ACM Press.

[HWG03] Anders Hejlsberg, Scott Wilamuth, and Peter Golde. *The C# Programming Language*. AddisonWesley Professional, October 2003.

[IPW99] Atshushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In Loren Meissner, editor, *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, volume 34(10), pages 132–146, N. Y., 1999.

[Kil73] Gary A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 194–206. ACM Press, 1973.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[KLS90] Henry F. Korth, Eliezer Levy, and Abraham Silberschatz. A formal approach to recovery by compensating transactions. In *The VLDB Journal*, pages 95–106, 1990.

[Lac91] Serge Lacourte. Exceptions in Guide, an object-oriented language for distributed applications. In *Object-Oriented Programming, 5th European Conference (ECOOP)*, pages 268–287, 1991.

[LAZJ03] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. In *Programming Language Design and Implementation*, San Diego, California, June 9–11 2003.

[LC98] David E. Lowell and Peter M. Chen. Discount checking: transparent, low-overhead recovery for general applications. Technical Report CSE-TR-410-99, University of Michigan, November 1998.

[LCC00] David E. Lowell, Subhachandra Chandra, and Peter M. Chen. Exploring failure transparency and the limits of generic recovery. In *USENIX Symposium on Operating Systems Design and Implementation*, October 2000.

[Lev77] R. Levin. *Program structures for exceptional condition handling.* PhD thesis, Carnegie Mellon University, June 1977.

[LN98] K. Rustan M. Leino and Greg Nelson. An extended static checker for Modula-3. In Kai Koskimies, editor, *Compiler Construction*, volume 1383 of *Lecture Notes in Computer Science*, pages 302–305, April 1998.

[LOLZ01] Chengfei Liu, Maria E. Orlowska, Xuemin Lin, and Xiaofang Zhou. Improving backward recovery in workflow systems. In *Conference on Database Systems for Advanced Applications*, April 2001.

[LS83] Barbara Liskov and Robert Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, July 1983.

[LY97] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, January 1997.

[MT97] Robert Miller and Anand Tripathi. Issues with exception handling in object-oriented systems. In *Object-Oriented Programming, 11th European Conference (ECOOP)*, pages 85–103, 1997.

[NE02] Jeremy W. Nimmer and Michael D. Ernst. Automatic generation of program specifications. In *International Symposium on Software Testing and Analysis*, pages 232–242, Rome, Italy, July 22–24, 2002.

[NIS02] NIST. The economic impacts of inadequate infrastructure for software testing. Technical Report NIST Planning Report 02-3, NIST, May 2002.

[NMW02] George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy code. In *Principles of Programming Languages*, pages 128–139. ACM, January 2002.

[OBT$^+$02] David Oppenheimer, Aaron B. Brown, Jonathan Traupman, Pete Broadwell, and David A. Patterson. Practical issues in dependability benchmarking. In

Second Workshop on Evaluating and Architecting System dependabilitY (EASY 02). San Jose, CA, October 2002.

[O'H05] John O'Hanley. Always close streams. In *http://www.javapractices.com/*, 2005.

[OW97] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *ACM Symposium on Principles of Programming Languages*, pages 146–159, 1997.

[PSWP02] Elizabeth Hanes Perry, Mike Sanko, Brian Wright, and Thomas Pfaeffle. Oracle9i JDBC developer's guide and reference. Technical Report A96654-01 (Release 2 (9.2)), http://www.oracle.com, March 2002.

[RHS95] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 49–61, San Francisco, California, 1995.

[RM99] Martin P. Robillard and Gail C. Murphy. Regaining control of exception handling. Technical Report TR-99-14, Dept. of Computer Science, University of British Columbia, 1, 1999.

[SBN$^+$97] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[SESS96] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster:

Surviving misbehaved kernel extensions. In *Symposium on Operating Systems Design and Implementation*, pages 213–227, Seattle, Washington, 1996.

[Sha02] Mary Shaw. Self-healing: Softening precision to avoid brittleness. In *Proceedings of the First ACM SIGSOFT Workshop on Self-Healing Systems (WOSS '02)*, pages 111–113. Charleston, South Carolina, November 2002.

[Sou03] SourceForge.net. About SourceForge.net (document A1). http://sourceforge.net. Technical report, 2003.

[SPS02] Richard Stallman, Roland Pesch, and Stan Shebs. *Debugging with GDB*. Free Software Foundation, 2002.

[SSF99] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: a fast capability system. In *Symposium on Operating Systems Principles*, pages 170–185, 1999.

[Str91] Bjarne Stroustrup. *The C++ Programming Language (second edition)*. AddisonWesley Publishing Company, 1991.

[Sun01] Sun Microsystems. Java pet store 1.1.2 blueprint application. http://java.sun.com/blueprints/code/. Technical report, 2001.

[SW91] Frank Schmuck and Jim Wyllie. Experience with transactions in QuickSilver. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 239–53. Association for Computing Machinery SIGOPS, 1991.

[TT97] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 1997.

[vdW02] Sander van der Wal. Creating the C++ auto_ptr<> utility for Symbian OS. Technical report, http://www.symbian.com/developer/techlib/, August 2002.

[VK02] Giuseppe Valetto and Gail Kaiser. A case study in software adaptation. In *ACM Workshop on Self-Healing Systems (WOSS '02)*, pages 73–78, November 2002.

[WFBA00] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In *Networking and Distributed System Security Symposium 2000*, San Diego, California, February 2000.

[WM01] David Walker and Greg Morrisett. Alias types for recursive data structures. *Lecture Notes in Computer Science*, 2071:177+, 2001.

[WML02] John Whaley, Michael C. Martin, and Monica S. Lam. Automatic extraction of object-oriented component interfaces. In *International Symposium of Software Testing and Analysis*, 2002.

[WN04] Westley Weimer and George C. Necula. 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*, pages 419–431, New York, NY, USA, 2004. ACM Press.

[WN05] Westley Weimer and George C. Necula. Mining temporal specifications for error detection. volume 3440 of *Lecture Notes in Computer Science*, pages 461–476, January 2005.

[YB85] Shaula Yemini and Daniel Berry. A modular verifiable exception handling mechanism. *ACM Transactions on Programming Languages and Systems*, 7(2), April 1985.