Finding and Preventing Run-Time Error Handling Mistakes

Westley Weimer George C. Necula
University of California, Berkeley
{weimer,necula}@cs.berkeley.edu [*]

ABSTRACT

It is difficult to write programs that behave correctly in the presence of run-time errors. Existing programming language features often provide poor support for executing clean-up code and for restoring invariants in such exceptional situations. We present a dataflow analysis for finding a certain class of error-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 safety policies because of incorrect error 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 error handling mistakes 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. We present a characterization of the most common causes of those errors and discuss the limitations of exception handling, finalizers and destructors in addressing them. Based on those errors, we propose a programming language feature 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 34kLOC program with it 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.

[*]This research was supported in part by the National Science Foundation Grants CCR-9875171, CCR-0085949, CCR-0081588, CCR-0234689, CCR-0326577, CCR-00225610, and gifts from Microsoft Research. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging; D.3.3 [**Programming Languages**]: Constructs and Features

General Terms

Experimentation, Languages, Measurement, Reliability, Verification

Keywords

Dataflow, Exceptions, Finalizers, Destructors, Try-Finally

1. INTRODUCTION

An IBM survey [12] reports that up to two-thirds of a program may be devoted to error handling. Our experiments with a suite of open-source Java programs ranging in size from 4,000 to 1,600,000 lines of code suggest that error handling is a lesser fraction of all source code but that it is still significant. Between 1% and 5% of program text in our experiments was comprised of error-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. Aside from programs specifically designed from the ground up for reliability (e.g., [7]), these proportions grow with program size and age. These broad numbers suggest that error handling is an important part of such programs and that much effort is devoted to it.

Despite the importance of addressing run-time errors, poor handling abounds. Here we are not concerned with the frequency of run-time errors but instead with how the program reacts to such exceptional situations. Error handling mistakes in which "applications don't properly handle error conditions that occur during normal operation" are one of the top ten causes of Java web application security risks [1]. Exception handling itself is insufficient: along with others [2], we observe that the two most common exception handling strategies are (1) do nothing, and (2) abort the program. In addition, existing exception handling mechanisms in programming languages are too low-level: higher-level machinery for error-handling is needed. In particular, attention is rarely paid to restoring invariants and adhering to interface requirements (e.g., releasing previously-acquired resources). Based on a static analysis, we will present experimental evidence that many Java programs make mistakes of this nature. The biggest problem is that programs fail to account for all possible execution paths in the presence of run-time errors.

It is hard to restore invariants correctly on all paths. We propose a mechanism where program actions can be associated with *compensations*, code that semantically undoes effects and restores invariants. This mechanism is similar to destructors and finalizers. This system ensures that if an action is taken, the program cannot halt without first executing the compensation. Thus a program that acquires a resource protected by this mechanism will release that resource along all execution paths, including those on which run-time errors occur. The compensating actions themselves are recorded and executed at run time. Compensations are stored in special run-time stacks. Our system uses stacks because dependencies between important resources make it desirable to execute the most recent compensation first.

The contributions of this paper include:

- A static analysis for locating mistakes in resource management run-time error handling. We report on such mistakes in a large number of programs.

- A language-level mechanism for associating compensations with actions and guaranteeing that the compensations are executed along all paths.

Section 2 provides a motivating example and shows common practices in error handling. We discuss our analysis for finding error-handling mistakes in Section 3. In Section 4, we apply the analysis and show that many programs make mistakes in their error handling. We attempt to characterize common mistakes in Section 5. In Section 6 we examine problems with error-handling via destructors and finalizers. Section 7 describes formally our proposed language features for run-time error handling. Case studies arguing that our proposed features are efficient and natural are given in Section 8. We place this work in context in Section 9 and compare it to existing techniques. In Section 10 we mention future research directions. Section 11 concludes.

2. MOTIVATING EXAMPLE

Consider this code, taken from Ohioedge CRM, the largest open-source customer relations management project [41].

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

This program uses language features to facilitate run-time error handling (i.e., Java's `finally`), but many problems remain. `Connections`, `PreparedStatements` and `ResultSets` represent global resources associated with an external database, so the program should close each one as quickly as possible. If a run-time error occurs on line 4, the runtime system will raise an exception, and the program will close the open `Connection` on line 10. However, if a run-time error occurs on line 7 (or 6 or 8), the resources associated with `ps` and `rs` will not be freed.

Moving the `close` calls from line 8 into the `finally` block is insufficient for at least two reasons. First, the `close` method itself can raise exceptions (as indicated by lines 10 and 11), so a failure while closing `rs` might leave `ps` dangling. Second, if an error occurs on line 5, an attempt will be made to `close` the never-opened and still-`null` `rs`, causing a `null`-pointer exception.

This sort of code is quite common and highlights a number of important observations. First, the programmer is aware of the safety policies: `try` and `close` abound. Second, there are many paths where error handling is poor. Third, there are a few control-flow paths where the error handling works correctly, so the programmer is aware of the correct policy. 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 error-handling must be included. In addition, if another procedure wishes to make use of `Connection`s, it must duplicate all of this error handling code. This duplication is frequent in practice; the 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 error handling is sometimes ignored (e.g., [7]). 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, and we claim it could be made easier with more support from the programming language.

In the next section we will discuss an analysis for automatically discovering such error-handling mistakes. For example, this analysis will report three paths in the example above. If an exception occurs on line 6, a `PreparedStatement` is leaked. If an error occurs on line 7, both the `PreparedStatement` and the `ResultSet` are forgotten. Finally, if the first call to `close` on line 8 raises an exception, the `PreparedStatement` is again leaked.

3. ERROR-HANDLING ANALYSIS

We present a static analysis for locating mistakes in resource management run-time error handling. This analysis yields paths through methods on which mistakes may occur and can be used to direct changes to the source code to improve error handling. The analysis may report false positives and may miss 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.

We consider each method body in turn, symbolically executing all code paths, abstracting away data values but paying special attention to control flow and exceptions.

The first step is to create the control-flow graph. Constructing a control-flow graph that explicitly accounts for exceptional control flow is non-trivial in Java. While `try-catch-finally` is conceptually simple, it has the most complicated execution description in the language specification [24] and requires four levels of nested "if"s in its English description. In short, it contains a large number of corner cases that programmers often overlook.

3.1 Fault Model

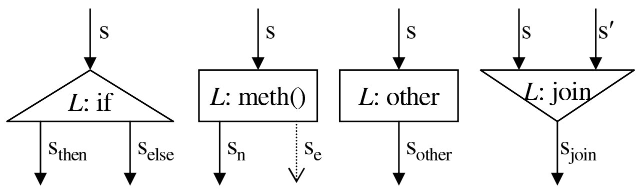
Modeling exceptional control-flow requires determining where exceptions can be raised. We treat `throw` statements directly, but all other expressions fall under a specific fault model we have adopted. In Java, method declarations explicitly list all `checked` exceptions that they can (transitively) raise. We assume that all method and constructor invocations can either return normally or raise any of their declared exceptions. This choice is motivated by experiments demonstrating that actual failures (e.g., pulling the plug on a remote machine) do map to application-visible `checked` exceptions at call sites [8]. In addition to such `checked` exceptions, Java also contains `unchecked` exceptions for situations like `null`-pointer dereferences and division by zero. Thus we could treat any division expression, for example, as terminating normally or raising a `DivisionByZero` exception. In our fault model we do not consider such implicit `unchecked` exceptions because they do not necessarily correspond to the run-time errors that concern us.

Finally, there is the practical issue of unavailable code (in particular, public domain code written against commercial database libraries). When a method in an unavailable library is invoked, its signature is also unavailable so we cannot determine what exceptions it may raise. Our fault model is to assume that it can raise any exception mentioned in a lexically-enclosing `catch` clause or raised by the enclosing method. This fault model gives the programmer the benefit of the doubt and also concentrates the analysis on mistakes in existing error handling (which is common in our experience), not mistakes caused by completely forgetting error handling (rare in our experience). Once an exception has been raised, type-checking is required in order to determine control flow. Barring `finally` clauses, execution transfers to the nearest enclosing `catch` clause with a declared exception parameter that is a supertype of the raised exception's type. Note that our fault model is Java-specific but that our dataflow analysis is language-independent.

3.2 Dataflow Analysis

Given the control-flow graph, our flow-sensitive, intraprocedural dataflow analysis [29, 13, 18] is designed to find paths along which programs forget to discharge obligations in the presence of run-time errors. We abstract away data values, and retain as symbolic state a path through the program and a multiset of outstanding resource types 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 types. We begin the analysis of each method body with an empty path and no obligations. If a symbolic state at the end of method contains outstanding obligations, we term it a *violation* and report it.

The analysis is parametric with respect to a safety policy. The safety policy enumerates the abstract obligations (e.g., `Socket` and `ResultSet` are different resources that must be tracked separately) and lists method invocations (by receiver class and name) that create and discharge such obligations. We formalize the safety policy as a set of triples: method names, an add-or-remove annotation, and a unique resource identifier. For example, these triples are a subset of our



$$
\begin{aligned}
s_{then} &= \mathsf{visit}(s, L)\\
s_{else} &= \mathsf{visit}(s, L)\\
s_n &= \begin{cases} \mathsf{add}(s,o,L) & \text{if } (\text{meth}, \mathsf{add}, o) \in \mathsf{Policy}\\ \mathsf{del}(s,o,L) & \text{if } (\text{meth}, \mathsf{del}, o) \in \mathsf{Policy}\\ \mathsf{visit}(s,L) & \text{otherwise}\end{cases}\\
s_e &= \begin{cases} \mathsf{del}(s,o,L) & \text{if } (\text{meth}, \mathsf{del}, o) \in \mathsf{Policy}\\ \mathsf{visit}(s,L) & \text{otherwise}\end{cases}\\
s_{other} &= \mathsf{visit}(s, L)\\
s_{join} &= \begin{cases} \mathsf{visit}(\mathsf{shorter}(s,s'),L) & \text{if } s.\mathsf{obs} = s'.\mathsf{obs}\\ \mathsf{visit}(s,L) \cup \mathsf{visit}(s',L) & \text{otherwise}\end{cases}
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{visit}(\langle \mathsf{obs}, \mathsf{path}\rangle, L) &= \{\langle \mathsf{obs}, \mathsf{path} \bullet L\rangle\}\\
\mathsf{add}(\langle \mathsf{obs}, \mathsf{path}\rangle, o, L) &= \{\langle \{o\} \cup \mathsf{obs}, \mathsf{path} \bullet L\rangle\}\\
\mathsf{del}(\{o\} \cup \mathsf{obs}, \mathsf{path}\rangle, o, L) &= \{\langle \mathsf{obs}, \mathsf{path} \bullet L\rangle\}\\
\mathsf{shorter}(\langle o,p\rangle, \langle o,p'\rangle) &= \begin{cases} \langle o,p\rangle & \text{if } |p| \le |p'|\\ \langle o,p'\rangle & \text{otherwise}\end{cases}
\end{aligned}
$$

Figure 1: Analysis flow functions. obs **is a multiset of obligations,** path **is a sequence of locations. Each incoming state from** s **(an** $\langle \mathsf{obs}, \mathsf{path}\rangle$ **pair) is considered individually and produces a set of outgoing states.**

database safety policy and handle `ResultSet`s:

(`java.sql.Statement.executeQuery`, add, `ResultSet`)
(`java.sql.PreparedStatement.executeQuery`, add, `ResultSet`)
(`java.sql.ResultSet.close`, del, `ResultSet`)

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 [13] and Meta-compilation [18] 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.

The symbolic state $s = \langle \mathsf{obs}, \mathsf{path}\rangle$ propagated from node to node is a multiset of outstanding obligations, obs, and a sequence of program point labels, path. Thus at the beginning of line 6 in the example in the previous section, we would carry the information that a `Connection` and a `PreparedStatement` obligation are present (but not which variables contain them) and that lines 1, 2, 3, 4 and 5 have been executed.

3.3 Flow Functions

The flow functions are determined by the safety policy and are given in Figure 1. The four main types of control flow nodes are branches, method invocations, other statements and join points.

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 symbolic state propagates directly from the incoming edge to both outgoing edges. We write $\mathsf{visit}(s, L)$ to mean the symbolic state s with location L appended to its path.

A method invocation may terminate normally, represented by the s_n edge in Figure 1. Such an invocation of an obligation-creating method adds the appropriate obligation to the symbolic state (and is propagated to the next program point). We write $\mathsf{add}(s, o, L)$ to mean a new symbolic state like s that includes obligation o and has L appended to its path. A method invocation may also discharge an obligation depending on the safety policy, and we use $\mathsf{del}(s, o, L)$ to mean a new symbolic state like s with obligation o deleted and L appended to the path. An attempt to discharge an obligation that is not present is reported as a violation, but that never occurred in our experiments. All other successful method invocations do not change the outstanding obligations.

A method invocation may also raise a declared exception, represented by the s_e edge in Figure 1. Note that unlike the successful invocation case and as per our fault model, the symbolic state does not accrue obligations since the method did not complete normally. However, an attempt to discharge an obligation that raises an exception still removes that obligation. Thus we do not require that programs loop around `close` functions, invoking them until they succeed. Since `close` functions can generally not be retried and almost no programs we have observed do so, it would create unnecessary false positives. In any event, the resulting symbolic state is propagated to the nearest appropriate handler.

Continuing the above example, evaluating line 6 would propagate `Connection`, `PreparedStatement` and `ResultSet` to line 7 and `Connection` and `PreparedStatement` to line 10 (assuming that `executeQuery` is declared to raise an exception). As described above, any invocation of an obligation-discharging method discharges the obligation, but we still consider both normal and exceptional control flow. For example, `rs.close()` on line 8 always removes the `ResultSet` from the set of obligations, but may propagate to both the next statement on line 8 and the `finally` block on line 10.

The grouping (or join) function tracks separate paths through the same program point provided that they have different obligation lists. Our join function uses the *property simulation* approach [13] to grouping sets of symbolic states. We merge states with identical obligations by retaining only the shorter path for error reporting (modeled here with the function $\mathsf{shorter}(s_1, s_2)$).

To ensure termination we stop the analysis and flag an error when a program point occurs twice in a path with different obligation sets (e.g., if a program acquires obligations inside a loop). For the safety policies we considered, that never occurred. 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 57k LOC `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.

3.4 Error Report Filtering

Finally, we use heuristics as a post-processing step to filter reported violations and avoid false positives. Based on an exhaustive analysis of the false positives reported by this analysis, we designed three simple filtering rules.

When a violation is reported, we examine its `path`. Every time it passes through a conditional of the form `t = null` we remove an outstanding obligation that has the same type as `t`. 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 the `path` for assignments of the form `field = t`. For each such assignment we remove one outstanding obligation with the same type as `t`. 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 6, this cleanup method is almost never an actual finalizer.

Finally, if the `path` contains a `return t`, we remove one outstanding resource of type `t`. Such functions are effectively wrappers around the standard library constructors and the obligation for discharging 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.

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. We discuss the ramifications along with the results in the next section.

3.5 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 flow-sensitive because we want to consider control flow and because the abstract state of a resource (e.g., acquired or released) 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 types as the per-path symbolic state. 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 symbolic states separate if they have distinct sets of obligations.[1] We report a violation when a path leaves a method (normally or exceptionally) with an outstanding obligation.

[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	Lines of Code	Methods with Errors	paths with errors per safety policy		
			DB	File	Strm
`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: Error handling mistakes by program and policy. The "Methods" column indicates the total number of distinct *methods* that contain violations. The "DB", "File", and "Stream" columns give the total number of acyclic control-flow *paths* within those methods that violate the given policy.

4. POOR HANDLING ABOUNDS

In this section we apply the analysis from the previous section and show that many programs make mistakes in their run-time error handling.

Safety Policies. We surveyed `catch` blocks, `finally` blocks, and object finalizers in order to see what error handling existing programs found important. In order to make our analysis as applicable as possible, we looked for safety with respect to some standard Java library resources. These policies have an acquire-release flavor: if a resource has been acquired, a special function must later be called to release it. We arrived at a set of resources and policies by systematically inspecting Java programs and then checking the official definition in Java Platform API Specification. In our experiments, any program that used a resource of a certain type had at least some paths along which it adhered to that resource's safety policy. Correctly handling these resources is important but difficult to do perfectly in the presence of run-time errors.

Figure 2 shows results from this analysis. The "Methods" column shows the number of methods that violate at least one policy. The "DB" policy refers to an API for linking Java programs to SQL databases as mentioned in Section 2. Java programs consider this policy to be particularly important: the vast majority of `finally` blocks tried to deal with it. In includes ten method calls governing three resources: `Connections`, `Statements`, and `ResultSets`. The "Stream" policy deals with any class (even a user-defined one) that inherits from `java.io.InputStream`. The Java Platform API indicates that "system resources" may be associated with such streams. The "File" policy covers acquiring and releasing `java.io.FileInputStreams`. In addition, we applied a simple "Socket" policy (not detailed in Figure 2) that covers the `Socket` and `ServerSocket` constructors and `close` methods and found 14 paths with violations in 4 of the programs.

Programs. The programs were taken from open source repositories (e.g., [41]), ranging from business software (`compiere`) to music players (`jogg`), from databases (`hsqldb`) to games (`pcgen`) to heterogeneous concurrent modeling (`ptolemy2` [4]). 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 files mentioned resources covered by these safety policies, and in `aspectj` only 16% of the files did, making them behave like smaller programs.

False Positives. Figure 2 includes every violation reported by the analysis that was not automatically filtered out using the heuristic techniques presented in the previous section. All of the 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. The heuristics eliminate all false positives that the analysis would report on these programs.

The heuristics reduce 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.

Error Paths. All paths in Figure 2 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. Java's `IOException`, `SQLException` and `SecurityException` were the three most common exceptions that programs handled poorly in this manner.

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`. For simplicity, such cases are categorized in favor of the leftmost policy in Figure 2. 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. In the next section we will summarize trends in error-handling mistakes.

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 `ejbca`'s `RemoveVerifyServlet.loadUserDB`, a `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. [19], since the `ejbca` program takes care to handle `BufferedReader`s in some cases and is thus inconsistent with itself.

5. MISTAKE CHARACTERIZATION

In this section we attempt to characterize some of the errors found by our analysis, paying special attention to the qualities a handling mechanism should have in order to address these errors naturally.

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 mean that the programmer is aware of the possibility of run-time errors. Such examples show that it would be useful to have an automatic mechanism that does the right thing in common cases with no programmer intervention.

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.

A single project will often re-use an error-handling design pattern that contains flaws. In the rest of the discussion, we assume that (1) if method `a1()` is called then method `c1()` should be called, (2) `c1()` should not be called unless `a1()` succeeds (similarly for all `ai()` and `ci()`), and (3) that all methods can raise exceptions and suffer from run-time errors. Thus the previous example would be rendered:

```
try { a1(); wrk(); } catch { c1(); return; } wrk(); c1();
```

In `osage` multiple methods use this form:

```
try { a1(); a2(); } finally { c2(); c1(); }
```

Such handling can fail if `a2` or `c2` raises an exception. Some programs, like `compiere`, treat multiple resources sequentially but still fail to handle errors perfectly:

```
a1(); c1(); a2(); c2(); // no try-finally
```

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

```
try { a1(); a2(); } finally { c1(); }
```

Such partial handling covers some of the resources, but not all. The `ohioedge` program contains examples like:

```
for (...) { a1(); wrk(); c1(); }
```

Such handling can be difficult to reason about statically, especially if the important resources are not variables local to the loop body. Finally, various programs often use flags (and often use them correctly) to track resources and free them early:

```
try     { a1(); f = 0; if (...) { f = 1; c1(); } wrk(); }
finally { if (!f) { c1(); } }
```

In many cases, like the example in Section 2, 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 [7]) illustrate:

```
int f = 0;
try {
  a1(); f = 1;
  a2(); f = 2;
  a3(); f = 3;
} finally {
  switch (f) {
    case 3: try { c3(); } catch (Exception e) {}
    case 2: try { c2(); } catch (Exception e) {}
    case 1: try { c1(); } catch (Exception e) {}
  }
}
```

This 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:

```
int f = 0;
try {
  a1(); f = 1;
  if (p2) { a2(); f = 2; did_a2 = true; }
  a3(); f = 3;
} finally {
  switch (f) {
    case 3: try { c3(); } catch (Exception e) {}
    case 2: if (did_a2) try { c2(); } catch // ...
    case 1: try { c1(); } catch (Exception e) {}
  }
}
```

Thus attempting to deal with the issue introduces 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.

6. 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.

Destructors 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, 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., [22]). 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 7.1.

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.[2] In contrast, the database locks from the example in Section 2 should be released as quickly as possible, making finalizers an awkward fit for performance reasons. 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 [24]. 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. 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 4.

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 [5], 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 4, 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 4. 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 4 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 handle run-time errors is difficult. In the next section, we will describe language mechanisms that make it easy to do the right thing: all of the mistakes presented here could have been avoided using our proposed language extensions. In addition, the analysis presented in this section can easily verify that programs using our mechanisms are handling these resources correctly.

7. COMPENSATIONS

Based on our characterization of existing mistakes and coding practices in Section 5 and existing programming language techniques in Section 6, 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* [21]. 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

[2]Even this 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.

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.[3] 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 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 given 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 3 can be easily modified to show that programs that make use of compensation stacks do not forget obligations.

7.1 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 [37]), and adding convenient syntactic sugar for lexically-scoped compensation stacks.

In our system, the client code from Section 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(S);
... // do I/O-related work with rs
```

[3]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`).

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 6, 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., `methodScopeStack`) 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 [7]:

```
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 `CompensationStack`s:

```
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. 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 and Section 4).

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 4. 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.

8. 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 3, the programs were modified so that their existing error-handling makes 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.

The first case study, Aaron Brown's undoable email store [7], can be viewed as an SMTP and IMAP proxy that provides operators with system-level time travel. 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 microbenchmarks 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` [42], 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 [11, 8], 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.[4] The annotated

[4]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

version of the program raises no such exceptions: compensation stacks ensure that the database objects are handled correctly. The average response times for the original program (over multiple runs) is 52.06 milliseconds (ms), with a standard deviation of 100 ms. The average response time for the annotated program is 43.44 ms with a standard deviation of 77 ms. The annotated program is 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. 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.

9. RELATED WORK

Beyond destructors and finalizers, previous related work falls into five main categories: type systems, regions, exception schemes, ideas on error handling, and transactional models.

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. In addition, most such type systems work at the level of the resources themselves.

DeLine and Fähndrich [15] 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 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.

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 [20] extends the Vault type system with additional features that ease the burden of programming with linear types, but aliasing can still be difficult.

Regions. Gay and Aiken [22] propose a system for memory management using explicit first-class regions [43]. 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.

Exceptions. Most modern programming languages feature *exceptions* that behave according to the *replacement model* [23, 45] (see also [31, 14, 36, 26, 38]). Alonso et al. [2] believe that poor support for exception handling is a major obstacle for large-scale and mission-critical systems. Hagen et al. [27] 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.

Common Lisp's "`unwind-protect` *body cleanup*" 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 funcitons 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).

Dony [17] 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 [10] 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.

Error handling. Valetto and Kaiser [44] 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 [9] 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 [16] 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 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

already been completed.

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 8) 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 VINO operating system [39] 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.

Transactions. Database transactions provide a strong and well-founded approach to error handling [25]. However, many find the consistency and durability of transactions to be too heavyweight for most programming purposes (e.g., [2, 32, 14]).

Restructuring a program to make use of transactions can be a large, invasive change. Borg et al. [6] describe a checkpointing system that allows unmodified programs to survive hardware failures. Essentially, every system call is intercepted and logged. Others (e.g., [35, 40]) 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. [34] 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 (e.g., [33]). One such concept, the compensating transaction, semantically undoes the effects of another transaction *after* that transaction has been committed [30]. Designing a full compensating transaction that completely undoes the effects of a previous action is often difficult. Our system relaxes this requirement. Our system is also slightly more general than a pure linear saga [30] and more closely resembles a form of nested or interleaved linear sagas.

10. FUTURE WORK

Stack inference. In order to make this language feature as useful as possible to existing programs, we hope to reduce the annotation burden on the programmer. We can manually annotate the interfaces for several generally applicable safety policies, as in Section 4. Such interfaces would then be useful for any program that uses those libraries. A further step would be to devise an inference algorithm for compensation stack placement, similar to algorithms for region inference [43]. The goal would be to place compensation stack declarations so as to give them the smallest lifetime possible without freeing any resources while they are still in use (or otherwise invoking compensation code too early). Function calls that require compensation stacks would use the nearest enclosing stack. Given such an inference algorithm and annotated interfaces, this technique could be applied automatically to existing programs for certain safety policies.

Safety policies. We also plan to move to more interesting safety policies. Since such policies are usually program specific, we hope to mine specifications from the program source code. Our experiments indicate that when programs consider a resource, they handle it correctly at least some of the time. We hope to leverage existing techniques (e.g., 3, 28, 19]) to allow us to infer possible specifications. Such specifications can then be presented to the programmer and, if accepted, can be applied automatically. Specification mining based on the source code generally yields an unacceptable number of false positives. In this sort of application domain, however, where extra code will be executed at runtime rather than having the project rejected at compile-time, such false positives may be slightly more acceptable.

Theoretical work. Finally, much theoretical work remains to be done. We would like to have a formal proof of our safety guarantees for a more flexible system: if a program uses our features and type-checks according to our rules, then it can be guaranteed that its actions will be paired with compensations. We also hope to examine the interactions between this language feature and other features. For example, there are a number of known issues relating finalizers, concurrency and deadlocks [5]. We hope to prove, for example, that using this method (or a restriction to it) to keep track of compensations will not introduce any concurrency problems that were not present in the original code. We would also like to present a unifying framework for compensation stacks and regions.

11. CONCLUSIONS

We have presented an analysis that discovers when existing programs fail to restore invariants or invoke cleanup code in the presence of run-time errors. Using this analysis we have discovered over 800 methods with mistakes in their error-handling in almost 4 million lines of code. We then analyzed those mistakes qualitatively and we have discussed the strengths and weaknesses of exceptions, destructors and finalizers for run-time error handling. Based on that analysis, we have proposed a programming language feature based on advanced transaction models. 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, finding error-handling mistakes and suggesting features that would help to prevent them is an important step toward making more robust programs.

12. ACKNOWLEDGMENTS

We thank Aaron Brown for providing an explanation of and workload for his undo program, as well as fruitful discussion about Java error handling. Mark Brody, George Candea and Tom Martell generously provided their infrastructure and their workload generator for Pet Store. We thank Christopher Hylands Brooks for an insightful discussion into error handling in general and in ptolemy2. The anonymous reviewers provided useful suggestions, especially with respect to experimental results. Finally, one of the authors would like to thank Dr. Loren Rhodes for very early encouragement.

13. REFERENCES

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

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

[3] G. Ammons, R. Bodik, and J. R. Larus. Mining specifications. In *ACM Symposium on Principles of Programming Languages*, pages 4–16, 2002.

[4] P. Baldwin, S. Kohli, E. A. Lee, X. Liu, and Y. Zhao. Modeling of sensor nets in Ptolemy II. In *Proceedings of Information Processing in Sensor Networks*, Apr. 2004.

[5] H.-J. Boehm. Destructors, finalizers and synchronization. In *ACM Symposium on Principles of Programming Languages*. ACM, Jan. 2003.

[6] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle. Fault tolerance under UNIX. *ACM Transactions on Computer Systems*, 7(1), Feb. 1989.

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

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

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

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

[11] M. Chen, E. Kiciman, E. Fratkin, E. Brewer, and A. Fox. Pinpoint: Problem determination in large, dynamic, internet services. In *International Conference on Dependable Systems and Networks*, Washington D.C., 2002.

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

[13] M. Das, S. Lerner, and M. Seigle. Esp: path-sensitive program verification in polynomial time. *SIGPLAN Not.*, 37(5):57–68, 2002.

[14] 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.

[15] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *ACM Conference on Programming Language Design and Implementation*, pages 59–69, 2001.

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

[17] C. 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.

[18] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Symposium on Operating Systems Design and Implementation*, 2000.

[19] D. R. Engler, D. Y. Chen, and A. Chou. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Symposium on Operating Systems Principles*, pages 57–72, 2001.

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

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

[22] D. Gay and A. Aiken. Memory management with explicit regions. In *ACM Conference on Programming Language Design and Implementation*, pages 313–323, 1998.

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

[24] J. Gosling, B. Joy, and G. L. Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, 1996.

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

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

[27] C. Hagen and G. Alonso. Exception handling in workflow management systems. *IEEE Transactions on Software Engineering*, 26(9):943–959, Sept. 2000.

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

[29] G. 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.

[30] H. F. Korth, E. Levy, and A. Silberschatz. A formal approach to recovery by compensating transactions. In *The VLDB Journal*, pages 95–106, 1990.

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

[32] B. Liskov and R. Scheifler. Guardians and actions:

Linguistic support for robust, distributed programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, July 1983.

[33] C. Liu, M. E. Orlowska, X. Lin, and X. Zhou. Improving backward recovery in workflow systems. In *Conference on Database Systems for Advanced Applications*, Apr. 2001.

[34] D. E. Lowell, S. Chandra, and P. M. Chen. Exploring failure transparency and the limits of generic recovery. In *USENIX Symposium on Operating Systems Design and Implementation*, Oct. 2000.

[35] D. E. Lowell and P. M. Chen. Discount checking: transparent, low-overhead recovery for general applications. Technical Report CSE-TR-410-99, University of Michigan, Nov. 1998.

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

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

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

[39] 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.

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

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

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

[43] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 1997.

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

[45] S. Yemini and D. Berry. A modular verifiable exception handling mechanism. *ACM Transactions on Programming Languages and Systems*, 7(2), Apr. 1985.