Exception-Handling Bugs in Java and a Language Extension to Avoid Them

Westley Weimer

University of Virginia, Charlottesville, Virginia, USA, 22904
`weimer@virginia.edu`

Abstract. It is difficult to write programs that behave correctly in the presence of exceptions. We describe a dataflow analysis for finding a certain class of mistakes made while programs handle exceptions. These mistakes involve resource leaks and failures to restore program-specific invariants. Using this analysis we have found over 1,200 bugs in 4 million lines of Java. We give some evidence of the importance of the bugs we found and use them to highlight some limitations of destructors and finalizers. We propose and evaluate a new language feature, the compensation stack, to make it easier to write solid code in the presence of exceptions. These compensation stacks track obligations and invariants at run-time. Two case studies demonstrate that they can yield more natural source code and more consistent behavior in long-running programs.

1 Introduction

It is easier to fix software defects if they are found before the software is deployed. It is difficult to use testing to evaluate program behavior in exceptional situations, and thus difficult to use it to find exception-handling bugs. This chapter presents an analysis for finding a class of program mistakes related to such exceptional situations. It also describes a new language feature, the compensation stack, to make it easier to fix such mistakes.

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 fail because the disk is full or the underlying operating system is out of file handles. Similarly, a request to send a packet reliably may fail because of a network breakdown. Such examples represent actions that typically succeed but may fail through no fault of the main program.

Testing a program's behavior in exceptional situations is difficult because such situations, often called *faults* or *run-time errors*, must be artificially introduced while the program is executing. Since a program cannot perform correctly if all of its actions fail, a special *fault model* governs which faults may occur and when they may occur. Once the fault model has been established the faults must still be *injected* during testing. Both physical techniques [1] and special program analyses and compiler instrumentation approaches [2] have been used to inject faults. These approaches are still based on testing, however, and require indicative workloads and test cases.

Languages like Java, C++ and C# use language-level *exceptions* to signal and handle exceptional situations. The most common semantic framework for exceptions is the *replacement model* [3]. Complicated exception handling is difficult to reason about and to code correctly. It can become a source of software defects related to reliability. In particular, our experiments show that programs make mistakes when attempting to handle multiple cascading exceptions or multiple resources in the presence of a single exception.

In this chapter we use a fault model linking some run-time errors with certain language-level exceptions [1,4]. The model requires programs to behave correctly in the presence of common environmental conditions, like network congestion or database contention. It does not require that we consider memory-safety faults (e.g., array bounds-check failures). The model allows multiple "back-to-back" faults and is good at finding error-handling bugs. Our notion of correct behavior in the presence of exceptions is restricted to a few simple *specifications* for proper resource and API usage. We believe that programs should adhere to these specifications even in the presence of run-time errors.

We present a static dataflow analysis to find bugs in a program with respect to a given specification and a given fault model. An error report from the analysis includes a program path, one or more run-time errors and one or more resources governed by the specification. Such an error report claims that if the run-time errors occur at the given points along the program path, the program will misuse the given resources. The analysis is path-sensitive, intraprocedural and context-insensitive. It precisely models control flow, especially that related to exceptions. It abstracts away data values and only tracks the resources mentioned in the specification. Given a few simple filtering rules, the analysis reported no false positives in our experiments, but it can miss real errors. The analysis is quite successful at finding a certain class of mistakes: it found over 1,200 bugs in four million lines of code. In the few cases where we were able to make such measurements, 44-45% of the reported bugs were fixed by developers.

Based on that work finding bugs we propose the *compensation stack*, a language feature for ensuring that simple resources and API rules are handled correctly even in the presence of run-time errors. We draw on the concepts of compensating transactions, linear sagas, and linear types to create a model in which important obligations are recorded at run-time and are guaranteed to be executed along all paths. By enforcing a certain ordering and moving some bookkeeping from compile-time to run-time we provide more flexibility and ease-of-use than standard language approaches to adding linear types or transactions. We provide a static semantics for compensation stacks to highlight their differences from pure linear type systems and we present case studies using our implementation to show that they can be used to improve software reliability.

The rest of this chapter is organized as follows. We describe of the state of the art in handling exceptional situations at the language level in Section 2. We present a static data-flow analysis that finds exception-handling mistakes in Section 3. In Section 4 we present the results of our analysis, including experiments in Section 4.1 to measure the importance of the bugs found. We discuss finalizers and destructors in Section 5 and highlight some of their weaknesses in this context. In Section 6 we propose the compensation stack as a language feature. We describe our implementation in Section 7 and our type system in Section 8. In Section 9 we report on experiments in which we apply compensation stacks to error-handling in real programs in order to improve reliability.

2 Handling Exceptional Situations in Practice

The goal of an exception handler is program- and situation-specific. 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. We will not consider such high-level policy notions, instead focusing on generic, low-level notions of correctness related to resource handling and correct API usage.

```
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();
09:   ps.close();
10: } finally {
11:   try { cn.close(); } catch (Exception e1) { }
12: }
```

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

We begin with an example showing how error-handling mistakes can occur in practice. The code in Figure 1 was taken from Ohioedge CRM, the largest open-source customer relations management project. It uses language features to facilitate exception handling (i.e., nested `try` blocks and `finally` clauses), but many problems remain. `Connections`, `PreparedStatements` and `ResultSets` are important global resources associated with an external database. Our specification of correct behavior requires that the program eventually `close` each one.

In some situations the exception handling in Figure 1 works correctly. If a run-time error occurs on line 4, the runtime system will signal an exception, and the program will close the open `Connection` on line 11. However, if a run-time error occurs on line 6 (or 7 or 8), the resources of `ps` and `rs` may not be freed.

One common solution is to move the `close` calls from lines 8 and 9 into the `finally` block (e.g., before `cn.close` on line 11). 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 a `try-catch` and by its type signature), so a failure while closing the first resource might leave the last one dangling. Second, such code may also attempt to `close` an object that has never been created. If an error occurs on line 4 after `cn` has been created but before `rs` has been created, control will jump to line 10 and then invoke `rs.close` on line 11. Since `rs` has not yet been allocated, this will signal an error and control will jump to line 13 without invoking `close` on `cn`.

Using standard language features there are two common ways to address the situation. The first involves using nested `try-finally` blocks. One block is required for each resource handled simultaneously. This approach is rarely used correctly in practice, as methods commonly use three to five resources simultaneously. The second approach is to use special sentinel values or run-time checks to ensure proper resource handling. This approach has the advantage that one `try-finally` statement can handle any number of simultaneous resources. Unfortunately, it is difficult for humans to write such bookkeeping code correctly. If the guarded code contains any control-flow (e.g., allocating a list of `ResultSets`), that flow must be duplicated in the `finally` clause.

The Ohioedge CRM code highlights a number of observations. First, programmers are aware of the safety policies: `close` is common. Second, programmers are aware of possibility of run-time errors: language-level exception handling (e.g. `try` and `finally`) is used prominently. Third, there are many paths where exception handling is poor and resources may not be dealt with correctly. Finally, fixing the problem typically has software engineering disadvantages: the distance between any resource acquisition and its associated release increases, and extra control flow used only for exception-handling must be included. In addition, if another procedure wishes to make use of `Connections`, it must duplicate all of this exception handling code. Duplication is frequent in practice: the Ohioedge source file containing our example also contains two similar procedures with the same mistakes. Developers have cited this required repetition to explain why exception handling is sometimes ignored [5]. In general, correctly dealing with N resources requires N nested `try-finally` statements or a number of run-time checks. Such problems are error-prone in practice.

3 Bug-Finding Dataflow Analysis

We now present an analysis to find error-handling bugs. The 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 chose to take a fully static approach to avoid the problem of test case generation.

The analysis uses standard finite state machine specifications [6,7,8,9] to describe proper resource and API usage. We use a Java-specific fault model [4] to construct a control-flow graph where method invocations can raise declared `checked` exceptions. We chose Java because experiments show that its exceptions and run-time errors are correlated [1] and because method signatures include exception information. The analysis itself is language-independent. It is path-sensitive because we want to consider control flow and because the abstract state of a resource (e.g., "opened" or "closed") depends on control flow. It is intraprocedural for scalability. This leads to false positives, which we eliminate via heuristics that may mask real errors. The analysis abstracts data values, keeping sets of outstanding resource states as per-path dataflow facts. This abstraction can 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. We report a violation when a path leaves a method with a resource that is not in an accepting policy state.

3.1 Analysis Details

Our analysis sybmolically executes all code paths in each method body, abstracting data values but tracking control flow, exceptions and the specification.

Given the control-flow graph, our flow-sensitive, intraprocedural dataflow analysis [6,7,10] finds paths where programs violate the specification (typically by forgetting to discharge obligations) in the presence of run-time errors. We retain as dataflow facts paths through the program and a multiset of resource safety policy states for each path. That is, rather than tracking which variables hold resources we track a set of acquired resource states. We begin the analysis of each method body with an empty path and no obligations.

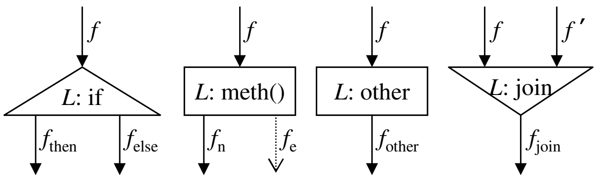
The analysis is given with respect to a safety policy specification $\langle \Sigma, S, s_0, \delta, F \rangle$. Given such a policy we must determine what state information to propagate on the CFG by giving flow and grouping functions. Each path-sensitive dataflow fact f is a pair $\langle \mathcal{T}, L \rangle$. The first component $\mathcal{T}$ is a multiset of specification states. So for each $s \in \mathcal{T}$ we have $s \in S$. We use a multiset because it is possible to have many obligations for a single resource type (e.g., to have two open `Socket`s). The second component L is a *path*, used when reporting violations, that lists program points between the start of the method and the current CFG edge.

3.2 Flow Functions

The analysis is defined in terms of the flow functions given in Figure 2. The four types of control flow nodes are branches, method invocations, other statements and join points. Because our analysis is path-sensitive and does not always merge dataflow facts at join points, each flow function formally takes a single incoming dataflow fact and yields a set of outgoing facts.

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

A method invocation may terminate normally, represented by the f_n edge in Figure 2. If the method is not covered by the policy (i.e., meth $\notin \Sigma$) then we propagate the symbolic state f directly. If the method is in the policy and the incoming dataflow fact f contains a state s that could transition on that method we apply that transition δ and then append the path label L. This is similar to the way tracked resources are handled in the Vault type system [11].



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

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

Fig. 2. Analysis flow functions

A method may also create a new policy resource. For example, the first time `new Socket` occurs in a path we create a new instance of the policy state machine to model the use of that `Socket` object.

The final case for a successful method invocation indicates a potential program error. In this case we have an event covered by the specification but the analysis is not tracking any resource that is in a state for which that event is valid. For the common two-state, two-event, open-close safety policies these violations correspond to "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`). We report such potential violations and stop processing along that path (i.e., the outgoing fact is the empty set) to avoid cascading error reports.

A method invocation may also raise a declared exception, represented by the f_e edge in Figure 2. Our fault model is any method can either terminate normally or raise any of its declared checked exceptions. It is this assumption that allows us to simulate faults and find error-handling mistakes. Unlike the successful invocation case, we do not update the specification state in the outgoing dataflow fact. This is because the method did not terminate successfully and thus presumably did not perform the operation to transform the resource's state. However, as a special case we allow an attempt to "discharge an obligation" or move a resource into an accepting state to succeed even if the method invocation fails. Thus we do not require that programs loop around `close` functions and invoke them until they succeed — that 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 [7] to grouping sets of symbolic states. We merge facts with identical obligations by retaining only the shorter path for error reporting purposes (modeled here as $\mathsf{shorter}(s_1, s_2)$). We may visit the same location many times to analyze paths with different sets $\mathcal{T}$.

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). In our experiments 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 loop. The analysis is exponential in the worst case but quite efficient in practice. Analyzing the 57,000-line `hibernate` program, including parsing, typechecking and printing out the resulting violations, took 104 seconds and 46 MB of memory on a 1.6 GHz machine.

The analysis goal is to find a path to the end of a method where a safety policy resource does not end in an accepting state. That is, for each $f = \langle \mathcal{T}, L \rangle$ that enters the end node of the CFG, if $\exists s \in \mathcal{T}.\ s \notin F$ the analysis reports a candidate violation along path L.

It is desirable to use heuristics in a post-processing step to filter candidate violations [4,12]. In this case 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.

4 Poor Handling Abounds

In this section we apply the analysis from Section 3 and common specifications to show that many programs make mistakes in handling exceptional situations. We consider a diverse body of twenty-seven Java programs totaling four million lines of code. The programs include databases, business software, networking applications and software development tools.

Figure 3 shows results from this analysis. The “Bugs” columns show the number of methods that violate at least one policy. We applied four library-resource policies (i.e., `Sockets`, `Streams`, file handles and JDBC database connections) to all programs. In addition, a total of 69 program-specific policies (found via specification mining [13]) were also used where applicable.

All of the reported methods were then manually inspected to verify that they contained at least one error along a reported path. Simple heuristics eliminated all false positive reports for these programs.

All paths in Figure 3 arose in the presence of exceptions the program did not handle correctly. More than half of these paths featured some exception

Program	Lines of Code	Bugs	Program	Lines of Code	Bugs
`javad 2000`	4k	**1**	`hibernate 2.0b4`	57k	**106**
`javacc 3.0`	13k	**4**	`jaxme 1.54`	58k	**6**
`jtar 1.21`	17k	**5**	`axion 1.0m2`	65k	**60**
`jatlite 3.5.97`	18k	**6**	`hsqldb 1.7.1`	71k	**53**
`toba 1.1c`	19k	**6**	`cayenne 1.0b4`	86k	**25**
`osage 1.0p10`	20k	**3**	`sablecc 2.17.4`	99k	**3**
`jcc 0.02`	26k	**0**	`jboss 3.0.6`	107k	**134**
`quartz 1.0.6`	27k	**17**	`mckoi-sql 1.0.2`	118k	**106**
`infinity 1.28`	28k	**21**	`portal 1.8.0`	162k	**39**
`ejbca 2.0b2`	33k	**31**	`pcgen 4.3.5`	178k	**17**
`ohioedge 1.3.1`	40k	**15**	`compiere 2.4.4`	230k	**322**
`jogg 1.1.3`	47k	**7**	`aspectj 1.1`	319k	**27**
`staf 2.4.5`	55k	**12**	`ptolemy2 3.0.2`	362k	**99**
			`eclipse 5.25.03`	1.6M	**126**

Fig. 3. 1251 Error handling mistakes in 3.9 million lines of code. The "Bugs" columns indicate the number of distinct *methods* that contain violations of various policies.

handling (i.e., the exception was caught), but the resource was still leaked. This result demonstrates that existing exception handlers contain mistakes.

4.1 The Importance of the Detected Bugs

Beyond raw numbers, the utility of the bugs found also matters. Even if a bug represents a violation of the policy, it may not be worth the developer's time to fix. Commercial software ships with known bugs. Bugs that are viewed as unlikely to affect real users go unfixed at many points in the development cycle.

Our analysis finds bugs that show up in the presence of exceptional situations by reporting resource leaks along paths that contain one or more run-time errors. We must show that these bugs are a serious problem "in the real world". A thorough evaluation of the importance of a bug is situation-specific and beyond the scope of this chapter. Aspects such as the performance or security impact of a bug or the cost of fixing it are difficult to measure quantitatively. We present some evidence to suggest that the bugs we report are important.

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, and thus unimportant, code, 44% of them rated a 3 out of 5 for taking place in "little used, experimental code", 19% of them rated a 4 out of 5 and were "definitely a bug in code that is used more often", and 26% of them rated a 5 out of 5 and were "definitely a bug in code that is used often." The 45% of the 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.

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. We thus performed a *time travel* experiment to determine whether the bugs found by our analysis were important enough to fix.

We used version control systems to obtain a snapshot of `eclipse 2.0.0` from July 2002 as well as one of `eclipse 3.0.1` from September 2004. We analyzed `eclipse 2.0.0` and noted the first 100 bugs reported. Without reporting any bugs to developers we looked for 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 analysis in `eclipse 2.0.0` had been fixed by `eclipse 3.0.1`. 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. Combined with our zero effective 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 anyway.

This section presents a static dataflow analysis that can locate software errors in a program's exception handling with respect to specification of correct behavior. The analysis examines each method in turn and tracks resources governed by the specification along all paths. Implementation details of the analysis presented here were previously discussed in earlier works [4,13].

5 An Attempt to Use Existing Language Features

Based on the mistakes found by our analysis, we claim that `try-finally` blocks are ill-suited for handling certain classes of resources during run-time errors. In essence, exceptions create hidden control-flow paths that are difficult for programmers to reason about. Before proposing a new language feature to simplify resource reclamation in exceptional situations, we must consider the advantages of two existing features: destructors and finalizers.

A *destructor* is a special method associated with a class. Destructors are used in C++ as well as other languages like C#. When a stack-allocated instance of that class goes out of scope, either because of normal control flow or because an exception was raised, the destructor is invoked automatically. Destructors are tied to the dynamic call stack of a program in the same way that local variables are. Destructors provide guaranteed cleanup actions for stack-allocated objects, even in the presence of exceptions. However, for heap-allocated objects the programmer must 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 [14]). 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. A destructor can be invoked early, but would typically have to use a flag to ensure that actions are not redone when it is called again. We want such bookkeeping to 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.

A *finalizer* is another special method associated with a class. Finalizers are available in Java as well as other languages like C#. A finalizer is invoked on an instance of a class when that instance is reclaimed by the garbage collector. The collector is not guaranteed to find any particular object and need not find garbage in any 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. Even that apparently-innocuous use of finalizers is often discouraged because programs have a limited number of file descriptors and can "race" with the collector to exhaust them [15]. In contrast, resources like JDBC database connections 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. Like destructors, finalizers can be invoked early but doing so typically requires additional bookkeeping.

More importantly, finalizers in Java come with no order guarantees. For example, a `Stream` built on (and referencing) a `Socket` might be finalized after that `Socket` if they are reclaimed in the same collection pass. We desire an error handling mechanism that can strictly enforce dependencies and provide a more intuitive ordering for cleanup actions. In addition, finalizers must be asynchronous, which complicates how they can be written.

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 [16], 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. Standard libraries might make good 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 we considered 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 rely on the library to do so. We want to make something like a finalizer more useful to Java programmers by making it easier to use and giving it 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 deal with important resources is difficult. In the next section, we describe a language mechanism that makes it easy to do the right thing: all of the bugs presented here could have been avoided using our proposed language extension.

6 Compensation Stacks

Based on existing mistakes and coding practices, 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* [17] and *linear sagas* [18]. 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. The compensations are applied in reverse order. This model is a good fit for run-time resource error handling. Many program actions 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 associated compensation will be also executed.[1] 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 are usually run 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.[2] 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

[1] We do not model abnormal program termination. Whether functions like `exit(1)` cause pending compensations to be executed or not is implementation-specific.

[2] 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 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 verify that a compensation cannot raise an exception. This option is not desirable for Java programs. 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`).

contents are eventually executed. They are as convenient as destructors when lexical and lifetime scoping coincide and are flexible enough to handle resources when they do not. 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 (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 (based on the type system described later in Section 8) to show that programs that make use of compensation stacks do not forget obligations.

7 Compensation Stack Pragmatics

We implemented compensation stacks via a source-level transformation for Java programs. We defined a `CompensationStack` class, added support for closures [19], and added syntactic sugar for lexically-scoped compensation stacks.

Consider again the client code from Figure 1. Our first step is to annotate the interfaces of methods that acquire important resources. For example, we associate with the action `getConnection` the compensation `close` at the interface level so that all uses of `Connections` are affected. Consider this definition:

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

We would change it so that a `CompensationStack` argument is required. The new 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 in Section 5, 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();  // execute all accrued compensations
}
```

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

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

All of the release actions are handled automatically, even in the presence of run-time errors. An implicit `CompensationStack` based on the method scope is being used and the resource-acquiring methods have been annotated to use such stacks. Note that annotating interfaces with compensating actions does not force a choice between lexically-scoped and heap-oriented resource management: when an object is first created using an interface its associated obligations can be put on any compensation stack.

Compensations can contain arbitrary code, not just method calls. For example, consider this code fragment adapted from [5]:

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

Traditional destructors are tied to objects, and there are many cases where a program would want to change the state of the object rather than destroying it. Destructors could be used here by creating "artificial objects" that are stack-allocated and perform the appropriate state changes on the enclosing object. However, such a solution would not be natural. For example, the program from which the last example was taken had 17 unique compensations (i.e., error-handling code that was site-specific and never duplicated) with an average length of 8 lines and a maximum length of 34 lines. Creating a new 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 become extra arguments to a 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 [11] or transactions [20]) 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 Compensation Stack Static Semantics

We provide a simple static type system for the correct use of explicitly-declared compensation stacks. This allows us to highlight the differences between our system and a full linear type system for tracking resources. It also provides a framework in which to describe the ordering guarantees provided by our system.

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

Fig. 4. A simple expression language with compensation stacks

Figure 4 shows a simple expression language involving compensation stacks. Normal program variables (e.g., integers) and objects (e.g., `Sockets`) are abstracted away. The join points after the non-deterministic conditional and loop model arbitrary control flow. This draconian simplification is sufficient for modeling compensating actions along all paths: gotos, while loops and exceptions merely provide additional control-flow branches and join points. Exceptions are tricky for programmers to deal with because they introduct control-flow that is invisible to human eyes. Such control flow paths are examined by our analysis (see Section 3); the type system presented here works similarly.

The compensation expressions are as described in Section 7. Each static $\mathsf{let}\ c_i =\ \mathsf{new\ CompStack()}\ \mathsf{in}\ e$ in the program is annotated with a fresh i for bookkeeping purposes. The $\mathsf{store}\ c_i$ expression represents storing a compensation stack in a global variable and setting a finalizer to run its compensations. Perhaps the most important detail is that $\mathsf{run}\ c_i$ and $\mathsf{runEarly}\ a_j\ \mathsf{from}\ c_i$ remove compensations from stacks after executing them at run-time. Compensations are removed from stacks even if those stacks are stored in memory or global variables. There is no danger of a "double-free" in calling $\mathsf{run}\ c_i$ multiple times.

Each compensation stack in this system is similar to a tracked resource in a linear type system [11]. Whenever a compensation stack is in scope, we know statically at what location i it was allocated (or loaded). The typing rules for compensation stacks are orthogonal to the typing rules for normal program objects. This type system rejects programs in which it cannot be guaranteed that all compensations will be executed.

A compensation stack, which might store un-executed compensations, may only go out of scope if it is stored in a global or if we can prove statically that

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

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

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

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

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

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

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

$$\frac{C_1 \cup \{i\}, D_1 \vdash e : C_2, D_2 \quad C_3 = C_2 \setminus \{i\} \quad i \in C_2 \qquad D_3 = D_2 \setminus \{i\} \quad i \in D_2}{C_1, D_1 \vdash \mathsf{let}\ c_i = \ \mathsf{load\ in}\ e : C_3, D_3}\ \mathit{load}$$

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

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

Fig. 5. Expression language static semantics

all of its compensations have been executed. We approximate this by requiring that run c_i or store c_i occur after the last compensate a_j with b_j using c_i before c_i goes out of scope. Our typing judgment maintains two *disjoint* sets: C, a set of "live" compensation stacks that may have un-executed compensations, and D, a set of "dead" compensation stacks on which all compensations have been executed or stored in memory. Adding a compensation to a dead stack makes it live. Thus we propose an effect type system for compensation stacks.

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

Figure 5 shows the typing rules for the language in Figure 4. The *seq* rule shows that this is a flow-sensitive type system for compensation stacks. The conservative *if* rule describes the effects of a conditional. Recalling the invariant $C \cap D = \emptyset$, at the conditional join point the resulting live set C_3 contains all stacks that might be live after either branch and the dead set D_3 contains all stacks that are dead after both branches. The $C_2 \cup D_2 = C_3 \cup D_3$ requirement prevents programs from creating a new compensation stack on one branch of the conditional. This is impossible in our example language where newly-created compensation stacks have local scope, but is possible in our Java implementation.

The *while* rule is also conservative. If the loop body can make a stack live, we assume that it does. If body might make a stack dead, we assume that it does not (a program must execute those stacks again later).

The *let* rule makes a new compensation stack and requires that it be dead as it goes out of scope. The *comp* rules are relatively simple since stack management happens at run-time. Adding a compensation to a dead stack makes it live and compensations can only be added to valid, in-scope stacks.

The *store* rules simulate storing a compensation stack in a global variable and consigning ultimate care of it to a finalizer. When it is finalized the run-time system will execute any remaining compensations associated with it. There is rarely a reason to store a stack with no outstanding obligations; the *storeD* is provided for completeness. The *load* rule is similar to the *let* rule but the stack need not be locally dead as it goes out of scope since it is still live in memory.

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

The *early* rule models our syntax for allowing the user to execute certain compensations early. If the particular compensation has already been executed or is otherwise no longer on the appropriate stack, nothing happens at run-time. Regardless, the *early* rule cannot make a stack dead.

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

This formal model does not track whether individual elements in a compensation stack have been executed. In practice, especially for lexically-scoped compensation stacks that do not escape their scope, a static analysis similar to the one in Section 3 can often determine exactly what the elements of such a stack might be. In such cases the implementation can optimize away the dynamic compensation stack object and insert the compensation code directly (effectively writing correct nested `finally-close` blocks for the programmer). We do not model such performance optimizations here.

We do not discuss method calls and returns here. An annotation system similar to the one described in Vault [11] suffices: each function type also specifies its requirements for compensation stacks and how it transforms them. The type system is also amenable to the standard extension for handling exceptions. For example, "`try` A B C `catch` D", where B may raise an exception, can be modeled as if $*$ then A ; B ; C else A ; B ; D. Type checking can be done by dataflow analysis (as in Section 3) without such code transformations.

9 Case Studies

We hand-annotated two programs to show that: (1) the run-time overhead is low; (2) it is easy to modify existing programs to use compensation stacks; and (3) it would not be difficult to write a new program from scratch using them. Guided by the dataflow analysis in Section 3, the programs' error handling was modified to use compensation stacks; no truly new error handling was added and the behavior was otherwise unchanged. This commonly amounted to removing a `close` call (and its guarding `finally`) and using a `CompensationStack` instead (possibly with a method that had been annotated to take a compensation stack parameter). The overhead of maintaining the stack was dwarfed by the I/O latency in our case studies. As a micro-benchmark example, a program that creates hundreds of `Socket`s and connects each to a website is 0.7% slower if a compensation stack tracks obligations to close the `Socket`s.

The first case study, Aaron Brown's undo-able email store [5], is a mail 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 to use compensations as well as annotating the interfaces for some 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. Compensating actions ranged from simple `close` calls to 34-line code blocks with internal exception handling and synchronization. The annotated program's performance was almost identical to the original on fifty micro-benchmarks and one example workload (all provided by the original author). Performance was measured to be within one standard deviation of the original, and was generally within one half of a standard deviation; the overhead of tracking 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 injecting a run-time error in the same cleanup code in both versions of the program, we were able to cause the unmodified version to drop all SMTP requests. The version using compensations handled that cleanup failure correctly and proceeded normally. While targeted fault injection is hardly representative, it does show that the errors addressed with compensations can have an impact on reliability.

The second case study, Sun's `Pet Store 1.3.2` [21], 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 [1,22], 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.[3] The annotated version of the program raises no such exceptions: compensation stacks ensure that the database objects are handled correctly. The average response times for the original program (over multiple runs) is 52.06 milliseconds (ms), with a standard deviation of 100 ms. The average response time for the annotated program is 43.44 ms with a standard deviation of 77 ms. The annotated program is both more consistent, because less middleware intervention was necessary, and also 17% faster.

Together, these case studies suggest that compensation stacks 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 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 reasonable. Finally, the checks ensure that cleanup code is invoked correctly along all paths through the program.

10 Conclusion

Software reliability remains an important and expensive issue. This chapter presents an approach for addressing a certain class of software reliability problems. We focus on exceptional situations, an aspect of software reliability that remains under-investigated.

First, we presented a static dataflow analysis for finding bugs in how programs deal with important resources in the presence of exceptional situations. The flow-sensitive, context-insensitive analysis scales well to large programs. The analysis found over 1,200 methods with mistakes in almost 4 million lines of Java code.

Second, based on those exception-handling bugs we designed a language feature to make it easier to fix such mistakes. We characterized why existing language features were insufficient. We proposed that programmers keep track of important obligations at run-time in special compensation stacks We provide a static semantics for compensation stacks to highlight their differences from previous approaches like pure linear type systems. In two case studies we showed that it is easy to apply compensation stacks to existing Java programs and that they can be used to make programs simpler and, in some cases, more reliable.

We find this work to be a successful step toward making software more reliable in the presence of exceptional situations. Using our analysis we can analyze programs to find error-handling mistakes. Once mistakes have been located we provide programmers with an easy-to-use tool for fixing them. All of this can be done cheaply, before the program is deployed. We hope that this approach, or approaches like it, will be more frequently adopted in the future.

[3] 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.

References

1. Candea, G., Delgado, M., Chen, M., Fox, A.: Automatic failure-path inference: A generic introspection technique for internet applications. In: IEEE Workshop on Internet Applications, San Jose, California (2003)
2. Fu, C., Ryder, B., Milanova, A., Wannacott, D.: Testing of java web services for robustness. In: International Symposium on Software Testing and Analysis. (2004)
3. Goodenough, J.B.: Exception handling: issues and a proposed notation. Communications of the ACM **18** (1975) 683–696
4. Weimer, W., Necula, G.C.: Finding and preventing run-time error handling mistakes. In: Object-oriented programming, systems, languages, and applications. (2004) 419–431
5. Brown, A., Patterson, D.: Undo for operators: Building an undoable e-mail store. In: USENIX Annual Technical Conference. (2003)
6. Engler, D., Chelf, B., Chou, A., Hallem, S.: Checking system rules using system-specific, programmer-written compiler extensions. In: Operating Systems Design and Implementation. (2000)
7. Das, M., Lerner, S., Seigle, M.: ESP: path-sensitive program verification in polynomial time. SIGPLAN Notices **37** (2002) 57–68
8. Ball, T., Rajamani, S.K.: Automatically validating temporal safety properties of interfaces. In: SPIN 2001, Workshop on Model Checking of Software. Volume 2057 of Lecture Notes in Computer Science. (2001) 103–122
9. Chen, H., Dean, D., Wagner, D.: Model checking one million lines of C code. In: Network and Distributed System Security Symposium, San Diego, CA (2004)
10. Kildall, G.A.: A unified approach to global program optimization. In: Principles of Programming Languages, ACM Press (1973) 194–206
11. DeLine, R., Fähndrich, M.: Enforcing high-level protocols in low-level software. In: Programming Language Design and Implementation. (2001) 59–69
12. Kremenek, T., Ashcraft, K., Yang, J., Engler, D.: Correlation exploitation in error ranking. In: Foundations of software engineering. (2004) 83–93
13. Weimer, W., Necula, G.C.: Mining temporal specifications for error detection. Volume 3440 of Lecture Notes in Computer Science. (2005) 461–476
14. Gay, D., Aiken, A.: Memory management with explicit regions. In: Programming Language Design and Implementation. (1998) 313–323
15. O'Hanley, J.: Always close streams. In: http://www.javapractices.com/. (2005)
16. Boehm, H.J.: Destructors, finalizers and synchronization. In: Principles of Programming Languages, ACM (2003)
17. Korth, H.F., Levy, E., Silberschatz, A.: A formal approach to recovery by compensating transactions. In: The VLDB Journal. (1990) 95–106
18. Alonso, G., Kamath, M., Agrawal, D., Abbadi, A.E., Gunthor, R., Mohan, C.: Failure handling in large-scale workflow management systems. Technical Report RJ9913, IBM Almaden Research Center, San Jose, CA (1994)
19. Odersky, M., Wadler, P.: Pizza into Java: Translating theory into practice. In: Principles of Programming Languages. (1997) 146–159
20. Alonso, G., Hagen, C., Agrawal, D., Abbadi, A.E., Mohan, C.: Enhancing the fault tolerance of workflow management systems. IEEE Concurrency **8** (2000) 74–81
21. Sun Microsystems: Java pet store 1.1.2 blueprint application. In: http://java.sun.com/blueprints/code/. (2001)
22. Chen, M.Y., Kiciman, E., Fratkin, E., Fox, A., Brewer, E.: Pinpoint: Problem determination in large, dynamic Internet services. In: International Conference on Dependable Systems and Networks, IEEE Computer Society (2002) 595–604