

Automated Support for Development, Maintenance, and Testing in the Presence of Implicit Control Flow

Saurabh Sinha, Alessandro Orso, and Mary Jean Harrold
College of Computing
Georgia Institute of Technology
801 Atlantic Drive
Atlanta, GA 30332 USA
{sinha,orso,harrold}@cc.gatech.edu

Abstract

Although object-oriented languages can improve programming practices, their characteristics may introduce new problems for software engineers. One important problem is the presence of implicit control flow caused by exception handling and polymorphism. Implicit control flow causes complex interactions in programs, and can thus complicate software-engineering tasks. To address this problem, we present a systematic and structured approach for supporting these tasks, based on the static and dynamic analyses of constructs that cause implicit control flow. Our approach provides software engineers with information for supporting and guiding development and maintenance tasks. We also present empirical results to illustrate the potential usefulness of our approach. Our studies show that, for the subjects considered, complex implicit control flow is always present and is generally not adequately exercised.

1 Introduction

The use of object-oriented languages in industry has grown considerably in the last decade. Today, object-oriented languages, such as Java, C++, and C# are commonly used for developing a wide spectrum of software that ranges from desktop applications to web services.

Object-oriented languages can improve development activities by enforcing good programming practices such as encapsulation, information-hiding, and modularity. However, some features of these languages cause new problems for software engineers. In particular, features such as exception handling and polymorphism can result in software behaviors that are hard to understand and foresee. Abuses or misuses of such features can thus result code that is faulty or difficult to understand, verify, and maintain [3, 13].

The main problem with exception handling and polymorphism is that they introduce *implicit control flow*—control flow that is not obvious in the source code. Although implicit control flow can occur also in procedural programs (e.g., in C, with the use of `setjump-longjump` constructs or function pointers), these occurrences are typically rare. In contrast, all but trivial object-oriented programs contain implicit control flow. In two independent studies, conducted on a variety of real Java programs, researchers found that exception-handling constructs occurred, on average, in up to 16% of the methods [15, 17].

Because of the complexity introduced by implicit control flow in object-oriented code, developers may overlook important interactions in the programs. For example, the propagation of an exception from a called method to a calling method several levels up the call chain can create unintended data dependences in the program. As a result, the effectiveness of software-engineering tasks such as code development and maintenance can be adversely affected if the presence of implicit control flow is not taken into account.

To address this problem, we have developed an approach that leverages static and dynamic analyses to provide support and guidance to software engineers during development and maintenance. The analyses include traditional techniques, such as type inference, data-flow analysis, and coverage analysis, and newly developed techniques.

During development, our technique provides information about exception-related code entities (e.g., throw statements, catch statements) and their interactions. Using this information, developers can avoid inappropriate coding patterns and produce code that is easier to understand and maintain. During maintenance, the same information can be used to restructure and refactor the program to remove inappropriate coding patterns [13].

During verification, our approach provides guidance and support in identifying test requirements and generating test data to satisfy such requirements. Testing requirements are expressed in terms of interactions among exception-related entities and are defined at different levels—higher-level requirements are harder to satisfy but result in more thoroughly tested code. The approach guides the tester in selecting a suitable level of testing by (1) providing information about the requirements at each level and (2) guiding the selection of the appropriate verification technique for the requirements.

To evaluate our approach, we implemented it in a prototype and performed empirical studies. In the first set of studies, we assessed the presence of complex implicit control flow in a set of real Java programs. The results show that such control flow is present in all the subjects we studied. In the second set of studies, we investigated the level of coverage of exception-related test requirements achieved by test suites developed for and distributed with the subjects. The results of this study show that coverage of exception-related constructs is generally low. Both results motivate the use of suitable approaches that provide automated support for development and maintenance of object-oriented programs in the presence of implicit control flow.

The main contributions of this paper are:

- An approach to support and guide development and maintenance in the presence of implicit control flow.
- A description of a prototype that demonstrates the feasibility of automating the approach and integrating into a tool such as an IDE (*Integrated Development Environment*).
- Two sets of studies that show that the presence of implicit control flow in object-oriented programs is common and that exercising interactions caused by such implicit control flow is a non-trivial task.

2 Background

2.1 Exception handling in Java

In Java, an exception is an instance of a class that is a subtype of `java.lang.Throwable`. An exception can be raised at any point in the program through a `throw` statement. A `try` statement provides the mechanism for associating exception handlers with the code. A `try` statement consists of a `try` block, and optionally, a `catch` block and a `finally` block. The legal instances of a `try` statement are `try-catch`, `try-catch-finally`, and `try-finally`. A `try` block contains statements whose execution is monitored for exception occurrences. A `catch` block specifies the type of exception it handles and contains a block of code that is executed when an exception of that type (or a subtype) is raised in the associated `try` block. A `try` statement can have a `finally` block. The code in a `finally` block is always executed, regardless of how control transfers out of the `try` block.

Java follows the *termination* model of exception handling: after an exception is handled, control does not return to the point at which the exception was raised, but continues at the first statement following the `try` statement where the

```

public class Sum {
    private static int i, j, sum, n;
    public static void main() {
1.  n = readInt();
2.  j = readInt();
3.  i = sum = 0;
        try {
4.      while ( i < n ) {
5.          add();
        }
    }
6.  catch ( ValueExceededException ve ) {
7.      System.out.println( ``value exceeded`` );
8.      return;
    }
9.  System.out.println( sum );
    }
    private static void add() throws
        ValueExceededException {
        try {
10.     checkValue();
11.     sum = sum + j;
        }
12.     catch ( NegativeValueException iv ) {
13.         sum = sum - j;
        }
14.     finally {
15.         System.out.println( ``current sum = ``+sum );
        }
16.     j = readInt();
17.     i = i + 1;
    }
    private static void checkValue() throws
        NegativeValueException,
        ValueExceededException {
        AddException e;
18.     if ( j < 0 )
19.         e = new NegativeValueException();
20.     else if ( sum + j > MAXVAL )
21.         e = new ValueExceededException();
22.     if ( e != null )
23.         throw e;
    }
}

```

Figure 1: Program Sum illustrates exception handling.

exception was handled. A Java exception can be propagated on the call stack: if a method raises but does not handle an exception, the exception is reraised in the context of the caller of that method.

Figure 1 presents a program to illustrate exception-handling constructs in Java. The program consists of a class `Sum`, which contains three methods: `main()`, `add()`, and `checkValue()`. The program reads in integers and adds the absolute values of those integers until either a user-specified number of integers have been summed or the sum exceeds `MAXVAL`. Methods `main()` and `add()` contain a `try` statement each. Method `checkValue()` checks the value of `j`, the integer that is to be summed, and raises either a `NegativeValueException`, if the value is negative, or a `ValueExceededException`, if the value would cause the sum to exceed `MAXVAL`. Neither of these exceptions is handled within `checkValue()`: `NegativeValueException` propagates up to `add()`, whereas `ValueExceededException` propagates up to `main()`. The `try` statement in `add()` contains a `finally` block; this `finally` block is executed irrespective of how control transfers out of the corresponding `try` block—by reaching the end of the `try` block, through a `NegativeValueException` that is handled in the corresponding `catch` block, or through a `ValueExceededException` that is not handled in the corresponding `catch` block.

In Java, a `finally` block executes in a *normal context* if (1) control reaches the end of a `try` block or a `catch` block, or (2) control leaves a `try` statement because of an unconditional transfer statement (e.g., `break`, `continue`, or `return`).

A finally block executes in an *exceptional context* if control leaves a try statement due to an unhandled exception. For example, the finally block in method `add()` (Figure 1) executes in normal context when either no exception or a `NegativeValueException` is raised in the try block; the finally block executes in an exceptional context if a `ValueExceededException` is raised in the try block. The context of execution of a finally determines where control flows after the execution of the finally. For example, if the finally in `add()` executes in a normal context, control flows to statement 16, whereas, if the finally executes in an exceptional context (for exception `ValueExceededException`), control exits `add()` and goes to statement 6 in `main()`.

Exceptions in Java can be classified according to different criteria. For example, a Java exception can be synchronous or asynchronous. A *synchronous exception* occurs at a particular program point; it is caused by an expression evaluation, a statement execution, or an explicit throw statement. Most exceptions in Java occur synchronously. An *asynchronous exception*, on the other hand, can occur at arbitrary, nondeterministic points in the program. For example, one thread can stop the execution of another thread by invoking the `stop()` method, which causes an instance of `java.lang.ThreadDeath` to be thrown in the target thread. A synchronous exception can be checked or unchecked. For a *checked exception*, the compiler must find a handler or a signature declaration for the method that raises the exception. Checked exceptions are designed to reduce the number of exceptions that are not handled properly in a Java program [9]. For an *unchecked exception*, the compiler does not attempt to find such a handler or a signature declaration. The unchecked exception classes are the class `java.lang.RuntimeException` and its subclasses and the class `Error` and its subclasses. Such exceptions are exempted from compile-time checks because (1) recovery from such exceptions is often difficult or (2) declaring such exceptions would not aid significantly in establishing the correctness of Java programs [9]. A synchronous exception is *explicitly raised* if the exception is raised at a throw statement in the application being analyzed. A synchronous exception is *implicitly raised* if the exception is raised in a library method (*library exception*) or by the runtime environment (*runtime-environment exception*). The source of an implicitly raised exception, therefore, lies outside the application being analyzed. For example, a call to the Java API method `java.util.Stack.pop()` can raise a `java.util.EmptyStackException`; an expression that dereferences an object reference can cause the Java runtime environment to raise a `java.lang.NullPointerException`.

2.2 Control-flow representation for exception handling

Our graph-construction techniques for exception-handling constructs model both intraprocedural and interprocedural control flow caused by exceptions [17]. The techniques construct *control-flow graphs* (CFGs) that contain nodes to represent throw statements, catch handlers, and finally blocks, and edges to represent the normal and exceptional control flow caused by these constructs. To determine the successors of a throw node, our CFG-construction algorithm performs type inference to determine the types of exceptions that can be raised by the corresponding throw statement. The algorithm then creates outgoing edges from the throw node for those exception types and labels each edge with the type of exception that causes that edge to be traversed. To model propagation of exceptions out of a method, the CFG contains exceptional exit nodes.

Figure 2 shows the CFGs for the methods in `Sum` (CFG edges are shown as solid lines). The CFG for method `checkValue()` contains a throw node, which has an outgoing edge for each type of exception that can be raised at the corresponding throw statement. The CFG for `checkValue()` also contains two exceptional exit nodes, one each for exceptions `NegativeValueException` and `ValueExceededException` that are propagated out of `checkValue()`.

Our representations provide two alternative ways of modeling the execution of finally blocks. The first approach

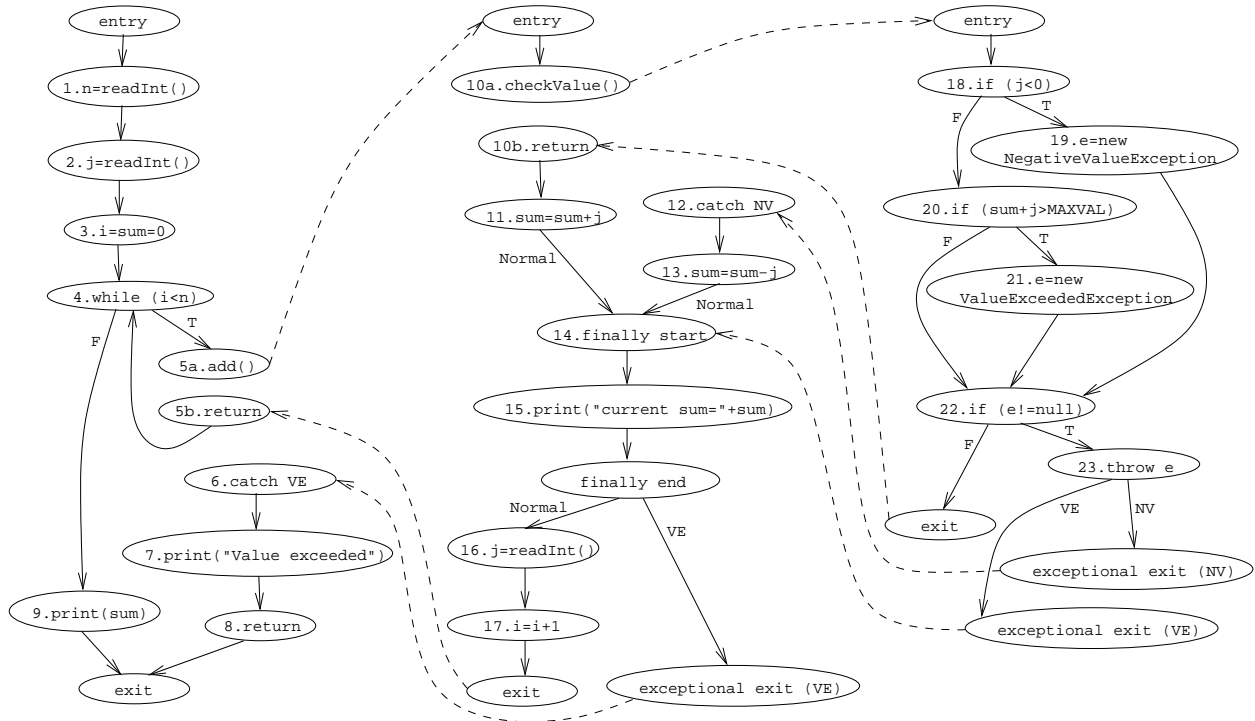


Figure 2: Interprocedural control-flow graph for Sum.

creates a separate CFG for each finally block and creates call and return nodes at appropriate points for calls to each finally block. The second approach embeds the control flow among the statements in a finally block in the CFG of the method that contains the finally. The approach also creates a *finally start node* to represent entry to the finally and a *finally end node* to represent exit from the finally. In Figure 2, the finally block in `add()` is represented using the second approach. The CFG for `add()` contains finally start and end nodes for the finally. Such a representation can contain unrealizable paths.¹ For example, path (11, 14, 15, finally end, exceptional exit (VE)) is unrealizable because control enters the finally in a normal context but leaves it in an exceptional context. To enable analysis techniques to traverse realizable paths through finally blocks, our representation stores attributes at in edges to finally start nodes and out edges from finally end nodes; these attributes let an in edge (out edge) to be matched correctly with the corresponding out edge (in edge).

To model interprocedural control flow, the algorithm constructs an *interprocedural control-flow graph* (ICFG) by connecting CFGs using call and return edges: a *call edge* connects a call node to the entry node of the called method, and a *return edge* connects the exit node of the called method to the corresponding return node. To represent interprocedural exceptional control flow, the algorithm creates exceptional return edges. An *exceptional return edge* connects an exceptional exit node in the CFG of a called method to a node, that represents the program point where control returns from the called method, in the CFG of the calling method. In Figure 2, call, return, and exceptional return edges are shown as dashed lines. As the figure illustrates, the CFG for `checkValue()` contains two exceptional exit nodes corresponding to the two types of exceptions propagated by `checkValue()`. Both of these nodes are connected, by exceptional return edges, to nodes in the CFG for `add()`: the exceptional exit node for type `NegativeValueException` is connected to catch node 12 in the CFG for `add()`, whereas the exceptional exit node for

¹A path is *unrealizable* if it contains invalid call–return sequences. A path that traverses through a finally block is unrealizable if it contains invalid entry–exit sequences

type `ValueExceededException` is connected to a finally call node in the CFG for `add()`.

3 Support for Software-Engineering Tasks

Our approach is based on performing static and dynamic analyses of constructs that cause implicit control flow, and using the results to support software engineering tasks. The approach provides information to software engineers to help (1) identify and eliminate inappropriate coding patterns during development and maintenance, and (2) exercise constructs that cause implicit control flow during verification. After defining terms used in the rest of the paper, we describe how our approach supports development, maintenance, and verification tasks, and discuss the analysis techniques on which the approach is based.

3.1 Definitions

Exception Deactivation. A catch handler *deactivates* a thrown exception. In Java, a finally block can also deactivate a thrown exception when that block executes in an exceptional context, and a throw statement, a return statement, or a break or continue statement (that transfers control outside the finally)² is reached within the finally block. For example, if the finally block in method `add()` in `Sum` (Figure 1) contained a return statement, a `ValueExceededException` that reached the return statement would be deactivated; the code in the catch handler in method `main()` would not be executed in response to that exception.

A deactivation can have different execution semantics. In this paper, we differentiate the following *semantics of a deactivation*: (1) ignore exception, (2) log error message and exit, (3) map and rethrow exception, and (4) take recovery action. A deactivation that occurs within a finally block or in an empty catch handler ignores the thrown exception—no recovery code executes in response to the handler.

Exception Flow. A *reaching throw statement* s_t is defined with respect to a deactivation s_d such that there exists an execution path from s_t to s_d and no statement along the path deactivates the raised exception. A *reaching exception type* s_d for exception type T is defined with respect to a deactivation s_d such that there exists an execution path from a throw statement that can raise an exception of type T to s_d and no statement along the path deactivates the raised exception. A *reachable deactivation* s_d is defined with respect to a throw statement s_t such that there exists an execution path from s_t to s_d and no statement along the path deactivates the raised exception.

The *distance* between a throw statement and a reachable deactivation is the maximum number of methods through which a thrown exception can propagate before reaching the deactivation. For example, in program `Sum`, the throw statement in line 23 has two reachable catch handlers—in lines 12 and 6. The distance from the throw statement to the catch handler in line 12 is 1, whereas the distance from the throw statement to the catch handler in line 6 is 2.

Calling Context. A throw statement s_t can execute in a number of different calling contexts. These contexts can be computed by identifying the different calling sequences from the entry of the program that cause the method containing s_t to be reached. For a catch handler s_c that is reachable from throw statement s_t , the calling contexts that are relevant for s_t-s_c are the calling sequences that start within the lexical scope of s_c and that cause the method containing s_t to be reached. The *relevant contexts* for a reaching throw statement at a catch handler are the sequences of calls that originate in the try block for the catch handler and cause the method containing the throw to be reached.

²A break or continue statement that does not transfer control outside the finally does not deactivate a pending exception.

Precision and Safety. The declared type of a reference is *imprecise* if it is not the least common super type of all possible runtime types of the objects that can be bound to that reference. In `Sum`, the declared type of variable `e` is precise: `AddException` is the least common super type of `NegativeValueException` and `ValueExceededException`.

A catch handler can potentially handle any subtype of the declared type of the catch handler. However, the handler may actually handle only a subset of the potential exception types. The type of a catch handler is *imprecise* if the exception type declared in the handler is not the least common super type of all statically reaching exception types. In `Sum`, the types of both the catch handlers are precise. However, if the type of either one were changed, for example to `AddException`, the types of those handlers would become imprecise.

In Java, exceptions can be declared in the method interfaces using the `throws` clause—each method can declare the exceptions that can propagate out of the method. For such checked exceptions, the method must declare the exceptions; for such unchecked exceptions, the method may or may not declare them. Therefore, for unchecked exceptions, the `throws` declarations can miss exceptions that can actually be propagated (i.e., they can be *unsafe*) and they can list exceptions that cannot be propagated (i.e., they can be *imprecise*). For checked exceptions, the compiler ensures that the `throws` declarations are safe; however, even for checked exceptions, the `throws` declarations can be imprecise.

In general, the precision of type-inference analysis determines the extent to which declared types of references, types of catch handlers, and exception declarations are imprecise. Type-inference algorithms (e.g., [11, 12]) attempt to determine the types for each expression in a program by solving type constraints or by propagating local type information throughout a program. Type-inference analysis can be performed at different levels of precision—a more precise algorithm can compute fewer spurious types than a less precise algorithm. Thus, using a more precise type-inference algorithm can cause fewer declared types of references and fewer types of catch handlers to be identified as imprecise; moreover, such an algorithm can cause the extent of imprecision (measured in terms of the number of redundant types) to be less. However, for exception declarations, using a more precise type-inference algorithm can cause an opposite effect—the number of declarations that are marked imprecise and the extent of imprecision in those declarations can increase. Moreover, for exception declarations, a more precise type-inference analysis can cause fewer exception declarations to be identified as unsafe and a lower extent of unsafety (measured in terms of the number of missing types) in those declarations.

3.2 Code development and maintenance

For development and maintenance, we address the scenario in which a developer is inspecting the code either to improve it, through refactoring, or to understand it, for example while investigating a failure or planning a change. Our approach supports developers in performing such tasks by providing them with relevant information interactively.

There are several techniques for refactoring programs and for providing developers with useful information about a program. Some of these techniques are implemented in widely distributed tools, such as Eclipse or Idea. Most of these techniques and tools, however, provide limited information. For example, a typical use of such tools is to identify syntactic errors while coding. These approaches, although useful for developers and maintainers, do not leverage the power of existing program-analysis techniques.

In general, misuses of mechanisms, such as exception handling, can make programs difficult to understand and maintain. In some cases, they may also result in faults and cause failures that are difficult to investigate. Reimer and Srinivasan [13] mention several inappropriate usage patterns of exception handling that can make programs difficult to maintain. Such patterns are difficult to identify by simply examining the code or by using a purely syntactic analysis.

Table 1: Inappropriate coding patterns related to exception handling and polymorphism identified using our approach and the syntactic construct at which relevant information can be provided.

Inappropriate coding pattern	Syntactic construct at which information can be provided
Unreachable catch handlers	catch
Ignored exceptions	throw, catch, finally
Large distance between throw and catch	throw, catch
Imprecise or unsafe throws declarations	method interface
Imprecise types of catch handlers	catch
Imprecise declared types of references	throw, polymorphic call
Inappropriate contexts	throw, catch
Unhandled exceptions	throw, method interface
Mapping multiple exception types to the same type	catch

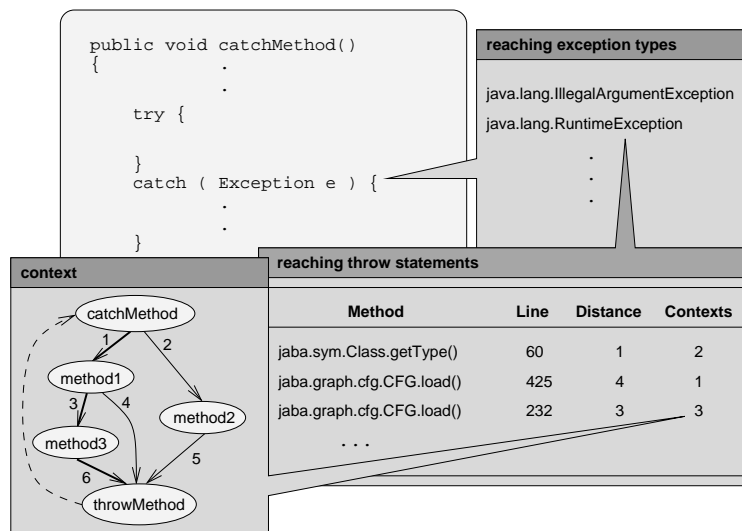


Figure 3: Schematic view of information about catch handlers that is provided using our approach.

However, by leveraging program-analysis techniques, our approach can identify such patterns in the code and report them to developers. For example, if our analysis reports the presence of unreachable catch handlers, the developer can inspect them to determine whether such handlers indicate programming errors (e.g., the wrong types of exceptions being caught) or can be removed from the program.

In this section, we describe the inappropriate coding patterns addressed by our approach and discuss how the approach lets developers identify and eliminate, or prevent, such patterns. The approach is based on providing developers with information that is derived from the analysis of the program and that can be interactively navigated. Figures 3, 4, 5, and 6 show a schematic view of the kinds of information provided at throw statements, catch blocks, finally blocks, and small throws declarations. (Although the approach is meant to be integrated within an IDE, in this paper, we present it without referring to any specific implementation.) We refer to these figures in the rest of the section while discussing inappropriate coding patterns. Our technique identifies several inappropriate coding patterns, related to exception handling and polymorphism; Table 1 lists these patterns.

Unreachable catch handlers. The presence of a statically unreachable catch handler may indicate a fault: the handler type may be incorrect which might be causing the handler to catch no exceptions. If not, such a catch handler

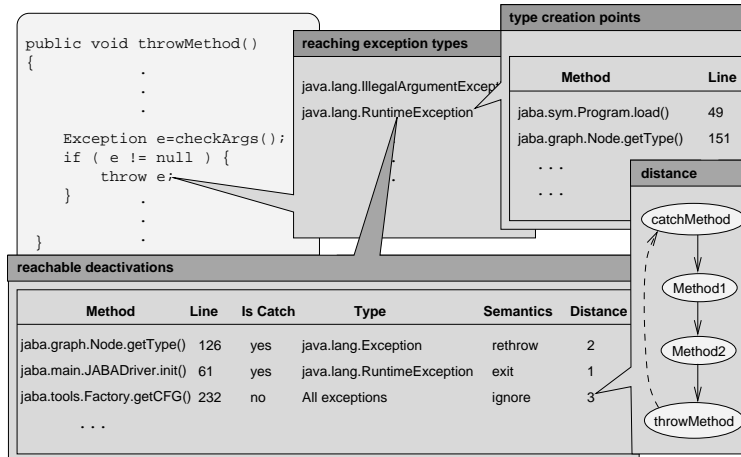


Figure 4: Schematic view of information about throw statements that is provided using our approach.

represents dead code—it serves no purpose and can be removed from the program.

To let developers identify unreachable catch handlers, our approach provides them with information, for each catch block, about reaching exception types. Developers can use this information to check whether a catch handler is statically unreachable—a catch handler that has no reaching exception types is statically unreachable.

Figure 3 shows the developer’s interaction with an IDE to get this information: after selecting a catch handler, the reaching exception types are presented in a box (reaching exception types). In the figure, the box shows that the catch handler can be reached by exceptions of types `java.lang.IllegalArgumentException` and `java.lang.RuntimeException`.

To determine whether a catch handler is unreachable because of an incorrect type (and is therefore failing to handle exceptions that it would otherwise handle), the developer can change the type of the handler to `java.lang.Throwable` and then compute the information about reaching types. If no types reach the handler, the handler can be removed. However, if some types reach the handler, using information about where those exceptions originate (box reaching throw statements in the figure), the developer can determine whether the type of the handler should be changed to handle some or all of those exceptions.

Ignored exceptions. In Java, an exception can be ignored in one of two ways: either the exception is handled by an empty catch handler or the exception is deactivated within a finally block. In most cases, ignoring exceptions is undesirable: no corrective action or logging code executes in response to the exception. The deactivation of exceptions in finally blocks may be unintentional and, therefore, erroneous. Even if the presence of such deactivations is intentional, it can make the code difficult to understand and maintain. In these cases, developers can, after identifying the ignored exceptions, modify the deactivation points.

If exceptions are ignored at an empty catch handler, it indicates that the caught exceptions require no recovery action or even logging. Thus, such an exception might be used only to alter the control flow; its occurrence may not really indicate an exceptional condition. In such cases, it may be preferable to code such conditions using constructs for normal conditional control flow. By doing so, the program can avoid the performance penalty that is usually associated with the execution of exception-handling code. Moreover, usage of exception-handling constructs can prevent code optimizations that could have been performed in the absence of such constructs.

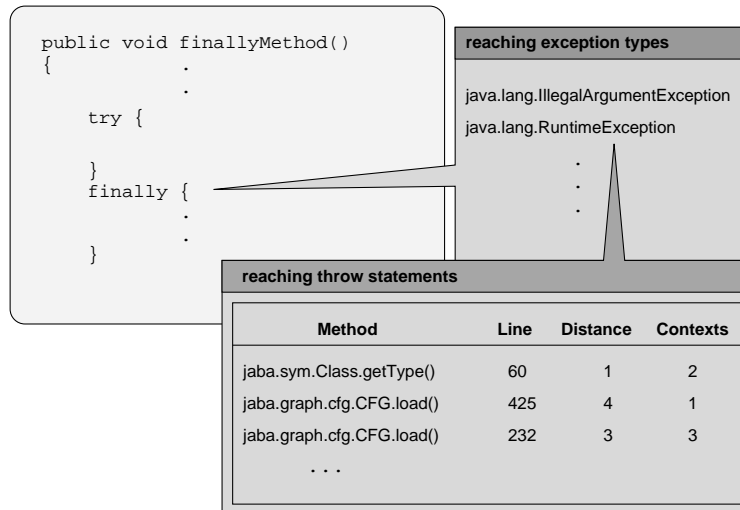


Figure 5: Schematic view of information about finally blocks that is provided using our approach.

To support developers in identifying ignored exceptions, our approach provides them with information, for each throw statement, about reachable deactivations and the properties of those deactivations. These properties include the semantics of the deactivation, which can be used to identify points where exceptions are ignored—all deactivations whose semantics is “ignore exception.”

Figure 4 shows the developer’s interaction with an IDE to get this information: after selecting a throw statement, the developer is provided with information about all types that may be thrown at a statement (box reaching exception types).³ For each type, box reachable deactivations shows the information about reachable deactivations. The box shows, for each deactivation, (1) where the deactivation occurs, (2) whether the deactivation occurs at a catch handler or a finally block, (3) the type of the deactivation, if the deactivation occurs at a catch handler, (4) the semantics of the deactivation, and (5) the distance between the throw and the deactivation. In the figure, the box shows that there are three deactivations, in method `java.graph.cfg.CFG.load()`, one of which ignores the exception.

Information about ignored exceptions can also be provided at catch blocks and finally blocks. For example, at an empty catch handler, the reaching exception types (see Figure 3) represent exceptions that are ignored. Using such information, the developer can reason whether ignoring those exceptions is appropriate. Similarly, our approach can provide information about reaching exception types at finally blocks. Figure 5 presents the developer’s interaction with an IDE to access such information. At a finally block that contains statements that can potentially deactivate reaching exceptions, the developer can request information about the reaching exception types. If no exception types reach the finally block, the statements in the finally block do not deactivate exceptions and, therefore, need not be modified. However, if some exceptions do reach the finally block, such exceptions can potentially be deactivated within the finally. For each reaching exception type, the developer can request information about reaching throw statements. Using such information, the developer can decide whether the exceptions should be deactivated within the finally.

Distance between throw and catch. In general, if exceptions are propagated a long way on the call stack, exception handling may be less meaningful and debugging more difficult [13]. As lower-level exceptions propagate to higher-level methods, they can cause the higher-level methods to raise exceptions that are inappropriate to the higher-level

³In most cases, the exception being thrown is created at the throw statement and there is only one type of exceptions that can be thrown (i.e., the throw is in the form `throw new E()`).

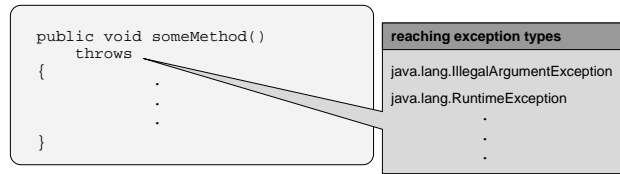


Figure 6: Schematic view of information about method interfaces that is provided using our approach.

abstraction. To avoid this problem, higher layers should map lower-level exceptions to exceptions that are explainable in terms of the higher-level abstraction [3].

Our approach supports developers in identifying throw–catch pairs whose distance may complicate maintenance and verification. The approach supports this task by providing information about the distance of a reaching throw statement at a catch handler (Figure 3) and the distance of a reachable deactivation at a throw statement (Figure 4).

Box reachable deactivations in Figure 3 shows how the information could be presented to developers and how they could further inspect throw–catch pairs with high distance. Developers can select each such throw–catch pair and be provided with information about the call paths along which the throw–catch can occur (box context). For the example, in the figure, developers would see that the throw statement occurring at line 232 of `jaba.graph.cfg.CFG.load()` and the selected catch statement are separated by a call chain of maximum length three—the one consisting of methods `method1`, `method3`, and `throwMethod`.

In cases in which developers consider the distance too high, they can reduce it by refactoring the code. For example, developers may decide to add an intermediate catch handler, in one or more methods within the call chain, that logs relevant information and remaps the exception [13].

Imprecise or unsafe throws declarations. In Java, the `throws` declarations alert users of a method to the exceptions that can propagate out of the method; users can then take corrective action. The presence of imprecise or unsafe exception declarations can provide misleading information to clients of the API. An imprecise exception declaration informs the clients of exceptions that can actually not be propagated out of the method; such information can cause the client code to contain unreachable catch handlers. An unsafe exception declaration can cause the clients to be unaware of exceptions that can potentially be propagated out of the method; thus, the client code may fail to take recovery actions in response to such exceptions and, instead, propagate the exceptions unintentionally.

To aid developers in identifying unsafe or imprecise exception declarations, our approach provides information about the types of exceptions that can reach method exits. Figure 6 presents the method-interface information that can be provided using our approach. Using this information, shown in the figure in box **reaching exception types**, the developer can ensure that the exception declarations in method interfaces are safe and precise. Moreover, this information can be useful for verifying whether all exceptions that can propagate out of a method should in fact propagate.

Although unchecked exceptions need not be declared using the `throws` clause, such exceptions should be documented—a description of exceptions that propagate out of a method is an important part of the documentation required to use a method properly [3]. Our approach provides developers with information that can help them efficiently create such documentation.

Imprecise types of catch handlers. The presence of catch handlers that have imprecise types can make the code difficult to understand and modify. The code in such a handler may be tailored for more general exception types that, in fact, cannot be caught by the handler. By making the type of the handler precise, the code in the handler can be improved to take recovery action that is more appropriate for the specific exceptions.

The information presented in Figure 3 can help the developer determine whether the type of a catch handler is imprecise. The example in Figure 3 illustrates that the catch handler can handle exceptions of type `java.lang.IllegalArgumentException` and `java.lang.RuntimeException`. However, the type of the catch handler is `java.lang.Exception`, which is not the least common super type of the reaching exception types and is, therefore, imprecise. Therefore, the type of the handler can be changed to `java.lang.RuntimeException`.

Imprecise declared types of references. Imprecise declared types of references can make the code difficult to understand; it can also cause the presence of unnecessary type casts in a program.

The information about reaching exception types at a throw statement (Figure 4) can help the developer determine whether the declared type of a reference at a throw statement is imprecise. For example, in the code shown in Figure 4, the declared type of `e` is `java.lang.Exception`. The possible runtime types of `e` are `java.lang.IllegalArgumentException` and `java.lang.RuntimeException`. Therefore, the declared type of `e` can be made precise by changing it to `java.lang.RuntimeException`. In Sum (Figure 1), the declared type of variable `e` is precise. However, if the declared type of `e` were `java.lang.Exception`, the declared type would have been imprecise and would have allowed the possibility of several other types of exceptions being raised at statement 23, although none of these types would have actually been raised. Therefore, in this scenario, the declared type could have been changed to make it precise.

Similarly, imprecise declared types of references at polymorphic call sites can make the program difficult to understand and maintain.

Inappropriate context of error recovery. Our approach can be used to provide developers with information about relevant contexts of a reaching throw statement at a catch handler. With this information, the developer can ensure that the recovery code in the handler is appropriate for each of those contexts. If common error recovery for all relevant contexts is not possible, the developer can add different catch handlers for contexts that require different error recovery.

In Figure 3, box context shows the relevant contexts for the reaching throw statement from line 232 of `java.graph.cfg.CFG.load()` at the catch handler. The box shows that, although there are three relevant contexts for the reaching throw statement, there is only one catch handler for all those contexts. Using this information, the developer can check whether this is appropriate. For example, it could be the case that the catch handler in `catchMethod` is appropriate when control flows along path (1, 3, 6) in Box 3, but not when control flows along path (1, 4). In this case, the developer can add a catch handler in `method1` that performs the necessary context-dependent error recovery.

Alternatively, developers can request context information at a throw statement for a specific reachable deactivation. Although not shown in Figure 4, our approach can be used to provide such information at throw statements.

Unhandled exceptions. Applications should preferably not propagate exceptions—this can cause the application to terminate with unhandled exceptions and without meaningful error messages. This should especially be true for checked exceptions, which are intended to be used for recoverable errors [3].

To check what exceptions propagate out of an application, developers can request the information about method interface (shown in Figure 6) for the “main” method of the application. Using information about reaching exception types, developers can reason whether the checked exceptions that are propagated out of the application should be handled within the application. If such exceptions cannot be handled within the application, developers can possibly change those exceptions from checked to unchecked.

Alternatively, for an exception raised at a throw statement, developers can request information about reachable method exits—these are the methods through which that exception can propagate. If the list of reachable method exits, includes the “main” method of the application, the exception can propagate out of the application. This information is not shown in Figure 4 but can easily be provided using our approach.

Mapping multiple exception types to the same type. Exception mapping is useful to ensure that different layers of software raise exceptions that are appropriate to the abstractions at those layers. Using information about reaching exception types at a catch handler (Figure 3), the developer can map those lower-level exceptions to ones that are appropriate for higher-level abstractions. However, exception mapping can make the source of an exception difficult to identify during debugging. The problem can be further compounded if a program maps multiple exception types to the same exception type [13]. To avoid this problem with exception mapping, developers can use exception chaining [3]: that is, encapsulate the lower-level exceptions into the higher-level exceptions. By using exception chaining, the complete stack trace from the occurrence of the last exception back to the occurrence of the original exception is available for debugging. Using the information about reaching exception types and throw statements, the developer can decide whether exception mapping is appropriate or exception chaining is preferable.

3.3 Testing and verification

During verification, our approach provides support in identifying test requirements for exception handling and generating test data to satisfy the requirements. In this section, first, we present techniques for generating test requirements to obtain different levels of coverage of exception handling and polymorphism. Then, we discuss how our approach presents information to the testers to guide them in verifying exception handling.

3.3.1 Test requirements for exception handling

The presence of exception handling causes interactions that can be verified at different levels of thoroughness. Figure 7 presents levels of coverage for exception handling organized hierarcically—the higher levels require stricter verification (at a higher cost), and thus provide more confidence in the correctness than lower levels. The levels of coverage can be used to generate test requirements, which can be verified using testing or inspection. The requirements can also be verified as properties that should or should not hold. Moreover, the requirements can be used to understand the interactions caused by exception handling.

The simplest level of coverage of exception handling is throw-statement coverage. At this level, the verification ignores the different types of exceptions that can be raised at a throw statement and the different catch handlers that are reachable from a throw statement. This level of coverage, called `(throw)` in Figure 7, is shown at the bottom-left corner in the figure. This level of coverage can be strengthened in two ways: (1) by considering the types of exceptions that can be raised at a throw statement, and (2) by considering the catch handlers that are reachable from a throw statement. These levels of coverage, called `(throw, type)` and `(throw, catch)` in Figure 7, require stricter

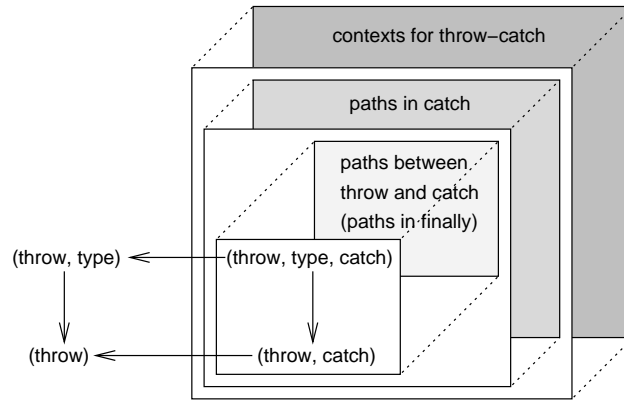


Figure 7: Levels of coverage for exception handling.

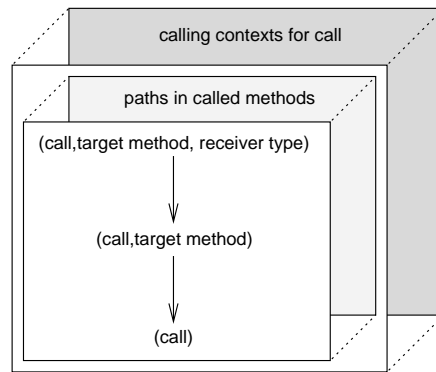


Figure 8: Levels of coverage for polymorphism.

verification of exception handling. For example, $(\text{throw}, \text{type})$ requires the verification of a throw statement for each type of exception that can be raised at the throw statement.

The arrows in Figure 7 represent the subsumption relations among the levels. A level of coverage *subsumes* another if the test requirements generated at the first level include the test requirements generated at the second level. $(\text{throw}, \text{type}, \text{catch})$ is a stronger level of coverage that subsumes both $(\text{throw}, \text{type})$ and $(\text{throw}, \text{catch})$; it requires the verification of a throw statement for each type of exception that can be raised at the statement and for each catch handler that is reachable from the throw statement. $(\text{throw}, \text{catch})$ and $(\text{throw}, \text{type}, \text{catch})$ can be strengthened further along three dimensions: (1) by considering the paths between a throw statement and a reachable catch handler (i.e., paths in finally blocks that occur between the throw and the catch), (2) by considering the paths in the catch handlers, and (3) by considering the relevant contexts for reaching throw statements.

These additional dimensions need not be considered in the order just listed and shown in Figure 7; the dimensions are independent of each other and can be considered in any order. For example, if paths in catch handlers are considered first, $(\text{throw}, \text{catch})$ can be made stronger as $(\text{throw}, \text{catch}, P_c)$, which requires the verification of each throw statement for each catch handler reachable from the throw and for each path in the reachable catch handler.

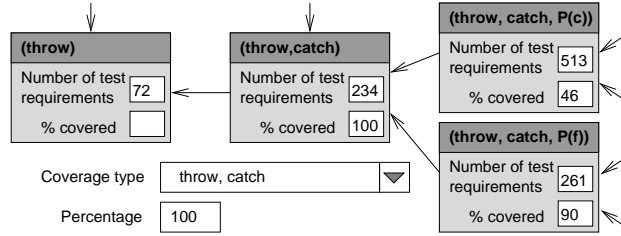


Figure 9: Information provided to testers to guide them in selecting a level of coverage.

3.3.2 Test requirements for polymorphism

Figure 8 presents the levels of coverage for polymorphism. Like the levels of coverage of exception handling, levels of coverage for polymorphism are organized hierarchically, based on subsumption relations. The simplest level of coverage of polymorphism is call-site coverage. At this level, the verification ignores the different methods that can be invoked at a polymorphic call site. This level of coverage, called `(call)` in Figure 8, can be strengthened by considering the methods that can be invoked at a polymorphic call site. The stronger `(call, target method)` measure requires the verification of a polymorphic call site for each method that can potentially be executed at that call site. This level can, in turn, be made stronger by considering not only the target methods but also the types of the receivers. The `(call, target method, receiver type)` level requires the verification of a polymorphic call site for each target method and each receiver type for the call.

Each of these three coverage levels can be made stronger along two dimensions: (1) by considering paths in the called methods, and (2) by considering the calling contexts for the call. These dimensions are independent of each other and can be considered in any order. For example, `(call, target method)` can first be strengthened as `(call, target method, P_{cm})`, which requires the verification of each path through each target method at a polymorphic call site. `(call, target method, P_{cm})` can then be strengthened by considering the calling contexts in which a polymorphic call site can be executed.

3.3.3 Support for verification

Our approach guides the tester in (1) exploring the test requirements to select a suitable level of coverage, (2) ordering the test requirements to determine the appropriate verification technique, and (3) generating test data to exercise the selected test requirements.

Select the level of coverage

Our approach uses static analysis to let the tester interactively assess the effort required to attain various levels of coverage of exception handling and polymorphism. The approach presents the hierarchy (Figures 7 and 8) to the tester and provides information about the number of test requirements that need to be covered to move from one level of coverage to a higher one. If no additional test requirements need to be covered to move to a higher level, the two levels of coverage are equivalent.

The approach lets the tester select the percentage of test requirements that might be targeted for coverage at a given level of coverage (level n). Given the percentage cov_n of test requirements that are covered at level n , the approach uses static analysis to compute the statically-maximal percentage cov_{n+1} of test requirements that would definitely be covered at level $n + 1$. To compute cov_{n+1} , the analysis selects $cov_n\%$ of test requirements at level n , such that the coverage of each selected test requirement would cause the definite coverage (statically) of a test requirement at

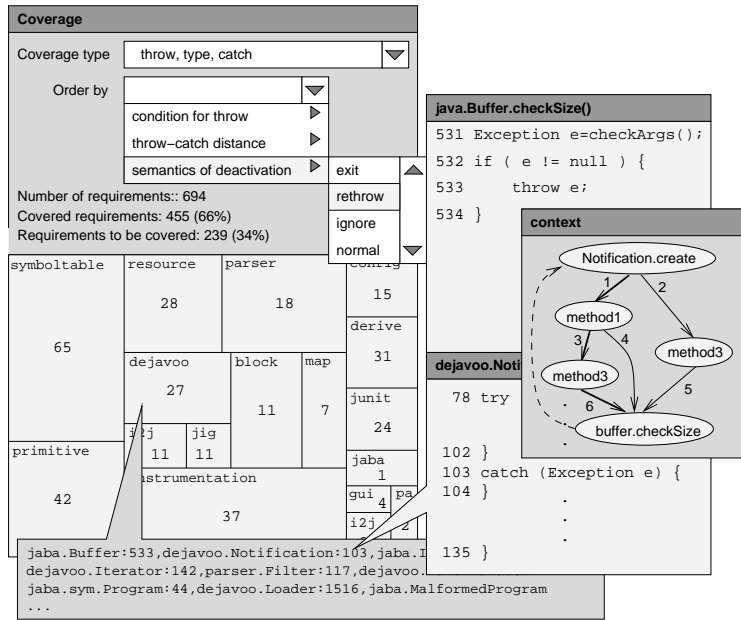


Figure 10: Schematic view of testing-requirements information for exceptions that is provided using our approach.

level $n + 1$. Thus, the analysis omits those test requirements at level n whose coverage does not (statically) imply the coverage of a test requirement at level $n + 1$. For example, suppose that there are 10 test requirements at `(throw)` level and 20 test requirement at `(throw, catch)` level. Further, suppose that five of the test requirements at `(throw)` level are such that their coverage definitely causes the coverage of a test requirement at `(throw, catch)` level (i.e., the exceptions raised at these five throw statements are not propagated out of the program along any path). If the tester selects 40% coverage at `(throw)` level, the tool reports that 20% (4 out of 20) of test requirements can be covered at `(throw, catch)` level. If the tester increases the targeted coverage at `(throw)` level to 50%, the tool reports that, at this higher coverage percentage, 25% (5 out of 20) of the test requirements can be covered at `(throw, catch)` level. At this point the analysis has considered all the five throw statements whose execution results in the definite execution of some catch handler. If the tester increases the coverage percentage at `(throw)` level to more than 50%, the tool does not report a higher coverage (than 25%) at `(throw, catch)` level because none of remaining five throw statements cause the definite execution of a catch handler (because there exist execution paths along which the exceptions raised at these statements can propagate out of the program).

Figure 9 presents the information provided to testers to help them select a level of coverage. The figure shows four types of coverage, and for each type, the number of test requirements. The tester can select a level of coverage and a target coverage percentage for the test requirements. For example, 100% `(throw, catch)` coverage would imply 46% `(throw, catch, P_c)` coverage and 90% `(throw, catch, P_f)`. Using such information, the tester can select the appropriate level of coverage for a program.

Order test requirements for coverage; generate test data

Once the tester selects a level of coverage, our approach lets them order the test requirements based on different characteristics of the test requirements. For example, for exception handling, the approach can let the tester order test requirements based on (1) conditions for reaching the throw statement, (2) distance between throw and catch, and (3) semantics of deactivation. Thus, the user can order the test requirements by decreasing order of the distance

between throw and catch. Using this information, the tester can focus the verification effort on those exceptions that can propagate several levels up the call chain, and determine the appropriate verification technique. Similarly, using information about the semantics of deactivation, the tester can group and order test requirements and select different verification techniques for different types of deactivations. For example, for those throw-catch pairs for which the deactivation semantics is “log error message and exit”, the tester might select inspection as the appropriate verification technique. For other deactivation semantics, the tester might select testing or formal verification as the appropriate technique. We are currently investigating how these and other characteristics can be used to order test requirements.

Figure 10 presents a schematic view of the types of information that can be presented in an IDE during verification. The upper part of the box on the left shows the type of coverage selected by the tester (`(throw, type, catch)` in the example) and statistics about them, in terms of covered and uncovered interactions. The lower part of the box on the left shows a treemap view⁴ of the code, in which each node represents a module and the numbers in the nodes indicate the number of uncovered throw-catch pairs involving that module (a throw-catch pair involves a module if the throw, the catch, or both occur in the module).

The IDE can also provide detailed information for a specific module. The box in the lowest left part of Figure 10 lists the set of throw-catch pairs for module `dejavoo`. As shown in the right part of the figure, for each entry, the user can get information about the location of the throw (method `java.Buffer.checkSize`, in the example), the location of the catch (method `dejavoo.Notification.create`, in the example), and the possible calling paths between them (the **context** box). Such information supports users while they inspect the pairs and test cases.

3.4 Underlying Analyses

In this section, we discuss the underlying analyses that enable us to compute the information described in Sections 3.2 and 3.3. First, we discuss type-inference analysis, which lets us compute potential types at throw statements and polymorphic call sites. Second, we describe in detail exception flow analysis, which lets us compute (1) reaching exception types and throw statements at deactivations, (2) reachable deactivations and program exits at throw statements, and (3) `(throw, catch)` pairs and `(throw, type, catch)` tuples. Third, we discuss context analysis, which lets us compute (1) relevant contexts for reaching throws at deactivations and (2) distance for throw-catch pairs. Finally, we discuss the analysis which we use to compute catch handler semantics.

3.4.1 Type-inference analysis

Type-inference analysis (e.g., [11, 12]) is required to determine the types of exceptions that can be raised at throw statements and the types of receivers at polymorphic call sites. Type-inference analysis can be performed at different levels of precision—a more precise algorithm can compute fewer spurious types than an imprecise algorithm. For the empirical results reported in this paper, we used class hierarchy analysis, augmented with local flow-sensitive analysis [17] for computing types of exceptions. For computing types of receivers at polymorphic call sites, we used class hierarchy analysis. In future work, we intend to use more precise type-inference analyses based on algorithms described in References [1, 10].

⁴The *treemap* visualization is a two-dimensional, space-filling approach to visualizing a tree structure in which each node is a rectangle whose area is proportional to some attribute of that node.

```

algorithm ComputeReachableDeactivations
input  throw           throw statement for which to compute reachable deactivations
        excpType        exception type at throw for which to compute reachable deactivations
output reachableDeacts deactivations reachable from throw
        reachableExits  program exits reachable from throw
declare worklist       Nodes in the ICFG that are traversed
        visited         boolean indicating whether a node had been visited
        n, e            ICFG node and edge, respectively
global ICFG           ICFG for program
begin ComputeReachableDeactivations
1.  initialize visited to false for each node in ICFG
2.  e = out edge from throw corresponding to exception excpType
3.  addToWorklist(e, worklist, visited)
4.  while worklist  $\neq \phi$  do
5.    remove edge e from worklist
6.    n = sink of edge e
7.    case n is catch node:
8.      add n to reachableDeacts
9.    case n is exceptional exit node:
10.     if method containing n is a “main” method then
11.       add method containing n to reachableExits
12.     endif
13.     foreach out edge e of n do
14.       addToWorklist(e, worklist, visited)
15.     endfor
16.   case n is finally start node:
17.     e = corresponding out edge from finally end node
18.     addToWorklist(e, worklist, visited)
19.     findDeactivationsInFinally(n, reachableDeacts)
20.   endcase
21. endwhile
22. return reachableDeacts, reachableExits
end ComputeReachableDeactivations

```

Figure 11: Algorithm for computing reachable deactivations for a throw statement.

3.4.2 Exception flow analysis

We use exception flow analysis to compute reachable deactivations and reachable program exits from throw statements. The analysis first builds the ICFG of the program, as described in Section 2. Then it traverses forward in the ICFG starting at each throw statement and computes reachable catch handlers and reachable deactivations within finally blocks.

Figure 11 presents the algorithm for computing reachable deactivations for a throw statement. The algorithm takes as inputs (1) a throw statement and (2) an exception type (that can be raised at the throw statement), and returns the set of reachable deactivations and reachable program exits for the throw statement and exception type. The algorithm uses a worklist approach to traverse forward, along realizable paths, in the ICFG starting at the throw statement.

To avoid visiting a node more than once, the algorithm associates a boolean *visited* with each node in the ICFG. This boolean is initialized to false and is set to true when a node is added to the worklist. The algorithm initializes *visited* for each node (line 1) and adds the out edge, corresponding to relevant exception type, from the throw node to the worklist (line 3). Function `addToWorklist`, not shown in Figure 11, adds the specified edge to the specified worklist if the sink of the edge has not been visited. After initializing the worklist, the algorithm removes an edge from the worklist and processes it until the worklist becomes empty (lines 4–21).

For each edge, the algorithm checks the type of the sink node. If the sink node is a catch node, the algorithm has identified a reachable deactivation; therefore, the algorithm adds the sink node to *reachableDeacts* (lines 7–8). If the

```

function findDeactivationsInFinally
input  finallyStartNode  start node for finally
        reachableDeacts  set of reachable deactivations
declare worklist        nodes traversed in ICFG while identifying deactivations in finally
        visited          boolean indicating whether a node had been visited
        finallyNodes     ICFG nodes corresponding to statements in finally
        n, e             ICFG node and edge, respectively
global ICFG           ICFG for program
begin findDeactivationsInFinally
1.  finallyNodes = nodes in ICFG that correspond to finally
2.  initialize visited to false for each node in ICFG
3.  e = out edge from finallyStartNode
4.  addToWorklist(e, worklist, visited)
5.  while worklist ≠ ∅ do
6.    remove edge e from worklist
7.    n = sink of edge e
8.    case n is throw node or return statement:
9.      add n to reachableDeacts
10.   case n is break or continue statement:
11.     if successor of n is not in finallyNodes then
12.       add n to reachableDeacts
13.     endif
14.   case n is finally start node:
15.     e = corresponding out edge from finally end node
16.     addToWorklist(e, worklist, visited)
17.     foreach out edge e of n do
18.       addToWorklist(e, worklist, visited)
19.     endfor
20.   case n is call node:
21.     e = out edge from corresponding return node
22.     addToWorklist(e, worklist, visited)
23.     foreach method m called at n do
24.       findReachableThrowsInMethod(m, reachableDeacts)
25.     endfor
26.   default
27.     if n is not exit, exceptional exit, or finally end node then
28.       foreach out edge e of n do
29.         addToWorklist(e, worklist, visited)
30.       endfor
31.     endif
32.   endcase
33. endwhile
end findDeactivationsInFinally

```

Figure 12: Function `findDeactivationsInFinally` that is called from `ComputeReachableDeactivations` (Figure 11) to compute deactivations that occur in finally blocks.

sink node is an exceptional exit node, the algorithm checks whether the method to which the node belongs is a “main” method; if it is, the algorithm adds the method to *reachableExits* (lines 9–12). The algorithm also adds the out edges of the node, if any, to the worklist (lines 13–15). If the sink node is a finally start node, to ensure traversal along realizable paths, the algorithm adds the corresponding out edge from the matching finally end node to the worklist (lines 16–18). Then, the algorithm calls function `findDeactivationsInFinally`, shown in Figure 12, to identify potential deactivations in the corresponding finally block (line 19). Apart from these three node types, the algorithm encounters no other node types along paths starting at throw nodes in the ICFG.

To illustrate with an example, consider the steps of the algorithm for computing reachable deactivations for the throw statement in line 23 of `Sum` (Figure 1) and exception type `ValueExceededException`. The algorithm initializes the worklist with the out edge labeled ‘VE’ from the throw node (see the ICFG for `Sum` in Figure 2). Next, the algorithm enters the loop in line 4: it removes edge (23, `exceptional exit (VE)`) from the worklist and examines

the sink of the edge. The sink is an exceptional exit node; therefore, the algorithm adds the only out edge of the node to the worklist. Because `checkValue()` is not a “main” method, the algorithm does not add it to *reachableExits*. In the second iteration, the algorithm removes edge (`exceptional exit (VE)`, 14) from worklist. In this iteration, the sink of the edge is a finally start node. Therefore, the algorithm first adds the corresponding out edge (`finally end, exceptional exit (VE)`) to the worklist, and then searches for deactivations (in this case, none) in the corresponding finally block. In the third iteration, the algorithm adds edge (`exceptional exit (VE)`, 6) to the worklist. In the fourth iteration, the algorithm removes that edge from the worklist and identifies the reachable deactivation—the catch handler in line 6 of `Sum`.

Figure 12 shows the pseudo code for function `findDeactivationsInFinally` that is called by `ComputeReachableDeactivations` in line 19. The function takes as inputs (1) a finally start node and (2) a set of reachable deactivations to which it adds any deactivations that it finds in the finally block. `findDeactivationsInFinally`, like `ComputeReachableDeactivations`, uses a worklist approach. It traverses the ICFG, along realizable paths, starting at the finally start node and identifies reachable (1) throw statements, (2) return statements, and (3) break and continue statements that transfer control outside the finally.

`findDeactivationsInFinally` first computes the set of nodes in the ICFG that corresponds to the finally block (line 1). We omit the details of that algorithm. Briefly, that algorithm walks the CFG of the method containing the finally, starting at each entry point in the CFG except the specified finally start node, and marks the reached nodes; all the unmarked nodes, which include nodes for nested finally blocks, correspond to the relevant finally block. Next, `findDeactivationsInFinally` initializes the visited flags (line 2) and adds the out edge from the finally start node to the worklist (lines 3–4). Then, the algorithm processes edges from the worklist until the worklist becomes empty (lines 5–33).

In each iteration, the algorithm removes an edge from the worklist and examines the type of the sink node. If the sink node represents a throw statement or a return statement (line 8), the algorithm adds the node to the set of reachable deactivations (line 9). If the sink node represents a break or continue statement (line 10), the algorithm checks whether the target of the break or continue statement is outside the finally; if it is, the algorithm adds the sink node to *reachableDeacts* (lines 11–13). If the sink node is a finally start node (line 14), the algorithm adds the corresponding out edge from the matching finally end node to the worklist (lines 15–16). By doing this, on reaching a finally end node, the algorithm need not add any nodes to the worklist and, therefore, traverses realizable paths through nested finally blocks. The algorithm also adds the successors of the finally start node to the worklist (lines 17–19). If the sink node is a call node (line 20), the algorithm adds the out edge from the corresponding return node to the worklist (lines 21–22). Also, the algorithm must traverse the called methods to find throw statements that may be reachable in the called methods—such throw statements can also deactivate a pending exception. However, the algorithm need not scan for return, break, or continue statements in the called methods because such statements do not deactivate exceptions when they occur in methods called within finally blocks. To identify reachable throw statements in called methods, the algorithm calls function `findReachableThrowsInMethod` (lines 23–25). That function performs a simple graph traversal, following realizable paths, from the entry of the specified method and identifies reachable throw statements; we omit the details of that function. For all other sink node types, except exit, exceptional exit, and finally end nodes, the algorithm adds all successors to the worklist (lines 26–31).

Our approach uses algorithm `ComputeReachableDeactivations` for each exception type that can be raised at each throw statement. After all reachable deactivations have been computed, using that information, our approach can easily compute reaching exception types and reaching throw statements at catch handlers, (`throw, type`) pairs, and

Table 2: Subject programs.

Subject	Description	Methods
daikon[7]	Dynamic invariant detector	7887
jaba[2]	Program analysis framework	2673
nanoxml[16]	XML parsing library	232
siena[4]	Publish subscribe system	195

(throw, type, catch) tuples.

3.4.3 Context analysis

To perform context analysis, our approach builds the call graph of the program being analyzed. A *call graph* for a program contains a node for each method in the program and an edge (n_i, n_j) for each call site in the method corresponding to n_i that calls the method corresponding to n_j . To determine the relevant contexts for a reaching throw s_t at a deactivation s_d , our approach first constructs a reduced call graph. The approach traverses the call graph twice. During the first traversal, the approach walks forward in the call graph, starting at the method containing s_d , and computes reachable nodes (methods). During the second traversal, the approach walks backward in the call graph, starting at the method containing s_t , and computes reaching nodes (methods). Next, the approach computes the intersection of the nodes visited during the two traversal; the nodes in the intersection constitute the nodes in the reduced call graph. Finally, to compute relevant contexts for s_t-s_d , the approach computes acyclic paths in the reduced call graph.

By examining the lengths of the paths, the approach computes the distance between the throw statement and the reachable deactivation.

3.4.4 Catch-semantics analysis

To compute catch handler semantics—that is, whether the semantics of a catch handler corresponds to (1) ignore exception, (2) log error message and exit, (3) map and rethrow exception, or (4) take recovery action—our approach computes the set of CFG nodes that correspond to a catch handler. By examining the size of the set of nodes, the approach determines whether a catch handler is empty and, therefore, whether that handler’s semantics is “ignore exception”. If the set of nodes contains a node that represents an exit statement such that the node postdominates⁵ all other nodes for the catch handler, the semantics of the catch handler is “log error message and exit”. Similarly, if the set of nodes for a catch handler includes a throw node that postdominates all other nodes for the handler, the semantics of that handler is “rethrow exception”. Finally, if none of the above is true for a catch handler, that handler’s semantics is the fourth type, that is, “take recovery action”.

4 Empirical Results

To evaluate our approach, we have built a prototype tool that identifies several of the inappropriate coding patterns mentioned in Section 3.2. The tool also computes the test requirements according to the four basic levels of coverage shown in Figure 7. In this section, we present the results of two empirical studies that we performed using the tool. In the first study, we examined the occurrences of bad coding patterns in real Java programs. In the second study, we

⁵A node n_i in the CFG *postdominates* a node n_j if all paths in the CFG from n_j to the exit node contain n_i .

Table 3: Unreachable catch handlers.

Subject	Number of occurrences by considering	
	Explicit and library exceptions	all exceptions
daikon	35/536 (6.5%)	19 (3.5%)
jaba	1/127 (0.8%)	0 (0%)
nanoxml	1/13 (7.7%)	0 (0%)
siena	0/29 (0%)	0 (0%)

studied the coverage of exception handling by test suites generated using traditional testing techniques. For both these studies, we used the subject programs listed in Table 2.

4.1 Occurrences of inappropriate coding patterns

We present empirical results to illustrate the occurrences of inappropriate coding patterns in Java programs. To show the usefulness of the information, we used it, where possible, to eliminate the patterns from JABA. We used JABA for this purpose because we are familiar with its code. This how the information would typically be used by developers.

Unreachable catch handlers. Table 3 presents data about the occurrences of unreachable catch handlers. It shows the number of occurrences of such catch handlers in the subjects by considering (1) implicit library and explicit exceptions, and (2) all exceptions. Column 2 of the table shows the number of catch handlers that are intended to handle only runtime-environment exceptions; such handlers are, therefore, unreachable if we analyze only explicit and library exceptions. Column 3 lists the number of catch handlers that either are truly unreachable (and, therefore, should be removed from the code) or have an incorrect type (which is causing them to be unreachable).

The data in Column 2 can be useful for applications that do not handle runtime-environment exceptions. For such applications, the handlers listed in Column 2 may actually have an incorrect type or can be removed. As the data in the table shows, all the subjects, except SIENA, contained handlers for only runtime-environment exceptions. From our knowledge of JABA, we expected it to have no such catch handler. Therefore, we examined the catch handler that was reported to handle only runtime-environment exceptions and found that it was handling runtime-environment exceptions by accident; it was actually meant to handle exceptions propagated by another method in JABA. However, the called method propagated no exceptions—otherwise, the handler would not have been unreachable. On examining the called method, we realized that the exception declaration of the method was imprecise—the declaration listed exceptions that were not propagated by the method. The calling method, which could have been written by a different programmer who, relying on the API documentation to determine potential exceptions, added a handler for those exceptions. Thus, using this information, we removed the handler because it was not needed.

Ignored exceptions. Table 4 presents data about the occurrences of ignored exceptions. It presents data about both ways that exceptions can be ignored: by being deactivated at an empty catch handler or within a finally block. The table shows the number of occurrences of such exception deactivations; for empty catch handlers, the table presents data about only those handlers that were reachable. None of our subjects contain deactivations in finally blocks, but all, except SIENA, contain empty catch handlers: DAIKON contains as many as 19 empty catch handlers, whereas JABA and NANOXML contain two each.

We examined the two empty catch handlers in JABA to determine whether the exceptions deactivated by those handlers should indeed be ignored and, if so, whether they could be programmed using constructs for normal condi-

Table 4: Ignored exceptions.

Subject	Empty catch handlers	Deactivations in finally
daikon	19	0
jaba	2	0
nanoxml	2	0
siena	0	0

Table 5: Distance between throw statements and catch handlers.

Subject	Range of distance	Throw-catch pairs with distance				
		0	1-2	3-5	6-10	> 10
daikon	0-(>30)	11	27	98	105	644
jaba	0-26	6	14	73	133	239
nanoxml	3-8	0	0	13	3	0
siena	2-4	0	31	16	0	0

tional control flow. In one case, we found that a method that propagated an exception was being called from several different contexts. The propagated exception represented an error in all but one of those contexts—in that one context, the exception represented a condition that was expected and required no error recovery. Thus, in this case ignoring the exception in that context was justified. However, programming this case using normal control flow would require the called method to return a boolean, which would have to be tested in all the contexts, and all but one of those contexts would have to throw an exception for the error. Thus, in this case, the alternative coding practice appears too cumbersome; instead, ignoring the exception in one context in which it does not represent an error seems reasonable. In the second case, we found that the exception being ignored should actually have not been ignored. Thus, in this case, we added a suitable logging message for the error.

Distance between throw and catch. Table 5 presents data about the distances between throw statements and catch handlers. The table shows, for each subject, the range of distances, and the number of throw-catch pairs that had different ranges of distances: 0, 1-2, 3-5, 6-10, and greater than 10. To limit the cost of the analysis, while gathering the distance, we excluded paths in the call graph with more than 30 nodes. Therefore, for DAIKON, we only report that the distance ranges from 0 to some value greater than 30. The data in the table show that the distances for NANOXML and SIENA, although fairly low in absolute terms, may be high relative to the sizes of those subjects. However, in JABA, exceptions can be propagated as many as 26 levels up the call chain. Also, for more than 50% of the throw-catch pairs, the distance is greater than 10. Although such large distances may be unreasonable in other applications, for JABA it is not. JABA parses class files to perform different analyses and most exceptions in JABA represent unrecoverable errors during parsing. Therefore, these exceptions are propagated all the way to the top of the call chain to the driver class that invoked the JABA API, and reported to the user.

Imprecise or unsafe throws declarations. Figure 13 presents data about the occurrences of unsafe or imprecise exception declarations in method interfaces. For this study, we computed the data by considering only explicit exceptions. The figure contains two segmented bars for each subject, the first for unsafe declarations and the second for imprecise declarations. The height of each bar represent 100% of the unsafe/imprecise declarations in a subject; the percentage values at the top of the bar indicate the percentage of methods whose throws declarations are un-

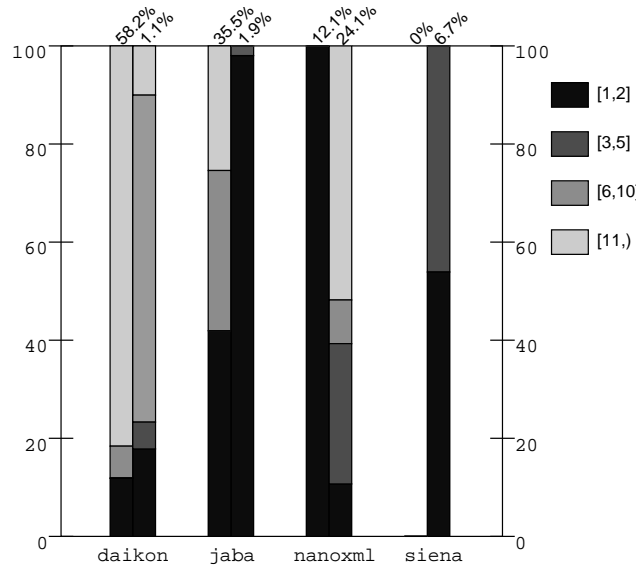


Figure 13: Imprecise or unsafe exception declarations in method interfaces.

safe/imprecise. For example, for JABA, 35.5% of the methods contain unsafe declarations and 1.9% contain imprecise declarations. The segments within a bar partition the unsafe/imprecise declarations by the extent of unsafety (missing exception types)/imprecision (redundant exception types). For example, for JABA, 42% of the unsafe declarations missed 1–2 exception types, 33% missed 6–10 types, and the remaining 25% missed more than 10 exception types. For DAIKON, 58% of the methods contain unsafe declarations, out of which 82% miss more than 10 exceptions. For SIENA, no methods contained unsafe declarations and about 7% contained imprecise declarations, all of which contained 1–2 redundant types.

The data in Figure 13 shows that the unsafe and imprecise declarations can be pervasive in Java programs. This is not surprising given that exception flow is complex; such declarations can easily degrade as the software evolves. Unsafe and imprecise declarations are undesirable, especially in methods that can be used externally (outside the application), because they provide misleading information to the clients of the API. Imprecise declarations can cause the client API to contain unreachable catch handlers. As discussed earlier, the unreachable catch handler in JABA was in fact caused by an imprecise `throws` declaration. Unsafe declarations can cause the client API to propagate exceptions unintentionally. Listing all potential exception types for each method may be cumbersome. In practice, this decision can be made based on the visibility of the method to external clients and the ability of the clients to recover from the propagated exceptions. Moreover, using our approach, the process can be automated to eliminate unnecessary burden on the developers to manually identify and list all exceptions. Automated analysis can present the list of potential exceptions to the developer, who can then decide which exceptions to list in the interface.

4.2 Coverage of exception handling using traditional testing techniques

In the second study, we examined the coverage of exception handling using existing test suites for our subjects. These test suites were generated either during the development of those applications or, in the case of NANOXML, post-development, based on functional specifications. In either case, the test suites were not designed to cover exception handling.

To determine the extent of exception coverage by these test suites, we generated test requirements, at the `(throw, type, catch)`

Table 6: Coverage of (throw, type, catch) tuples.

Subject	(throw, type, catch) tuples	Filtered tuples	Covered filtered tuples
daikon	3000	2498	7 (0.28%)
jaba	465	464	6 (1.29%)
nanoxml	16	16	4 (25.0%)
siena	47	41	0 (0%)

coverage level, for the subjects. Column 2 of Table 6 lists the number of test requirements for each subject. Next, we filtered the test requirements to eliminate those (throw, type, catch) tuples for which the test suites covered neither the throw method nor the catch method. Column 3 of Table 6 lists the number of filtered test requirements. As the data shows, few test requirements were filtered out—17% for DAIKON, 13% for SIENA, less than 1% for JABA, and none for NANOXML. Finally, we examined the percentage of filtered (throw, type, catch) tuples that were covered by the tests; Column 4 of Table 6 presents data for the covered (throw, type, catch) tuples. The data show that—except for NANOXML, which does not have many (throw, type, catch) tuples—very small percentages of the (throw, type, catch) tuples are covered.

The study, although limited in nature, indicates that test suites that are not designed for covering exception handling may not achieve good coverage of exception handling; test requirements for exception handling need to be considered to ensure adequate coverage.

5 Related Work

Several researchers have investigated the analysis of exception-handling constructs for development and maintenance. Robillard and Murphy [14] developed a tool that analyzes exception flow in Java programs to provide information for program understanding and detection of coding inconsistencies. The tool generates views that help a developer to understand the flow of exceptions across modules, and identify program points where exceptions are caught unintentionally, or where finer-grained exception handling may be possible. Their approach provides limited information about the flow of exceptions. They do not investigate how the information about exception flow can be used to identify and eliminate inappropriate coding patterns. Moreover, they do not address testing and verification of exception handling.

Yi and Chang [5, 19] present a set-constraint-based approach for analyzing the flow of exceptions in Java programs to enable the identification of uncaught exceptions, unreachable catch handlers, and imprecise catch handlers. Like Robillard and Murphy’s technique, their technique identifies a subset of the kinds of exception flow computed by our approach and provides an ad hoc, instead of an integrated, approach to the analysis. Additionally, unlike our approach, their approach does not identify inappropriate coding patterns.

Reimer and Srinivasan [13] identified several inappropriate exception usage patterns in large J2EE applications that have made maintenance of the applications difficult. They proposed several ways to identify and remove these inappropriate usage patterns, including integration of static analysis into an IDE. Our approach for supporting development and testing is similar to theirs in that it performs static analysis and uses the results to assist in locating and removing inappropriate exception use patterns. However, their approach targets logging and debugging activities instead of supporting code development and refactoring and understanding exception flow. Our approach provides a more extensive classification of inappropriate coding patterns than their approach. Also, they do not address testing

and verification of exception handling.

Other researchers have investigated the testing of exception-handling constructs. Chatterjee and Ryder [6] identify definition-use associations that are caused by exception variables or that arise along exceptional control-flow paths for use in data-flow testing. Unlike our approach, their approach does not investigate the interactions caused by exception-handling constructs or how to generate test requirements to exercise those interactions.

Tracey and colleagues [18] discuss the automated generation of test data for exceptions. They consider exception handling in Ada and target the coverage of each raise statement and each exception handler. They use genetic algorithms to generate test data automatically. Their technique is applicable to Ada, whose exception semantics are a subset of those in Java. Additionally, their coverage criteria—raise statement and exception handler—do not consider the interactions among the statements to generate the test requirements.

Fu and colleagues [8] present an approach for exercising catch blocks based on compiler-directed fault injection. Their approach identifies code blocks that are vulnerable to hardware and operating system faults, and injects faults at these locations to exercise the associated error recovery code. They also define a fault-catch coverage metric, which is the ratio of the number of faults for which a catch block has been exercised to the number of all possible faults for which the catch block can execute. Fu and colleagues' approach is orthogonal to our approach for verification of exception handling. They focus on the software's ability to handle hardware and operating system faults only; they do not focus more generally on the interactions caused by the presence of exception handling.

6 Summary and Future Work

We have presented an integrated and systematic approach for providing automated support for the development, maintenance, and verification of programs that contain implicit control flow. The approach uses static and dynamic program analyses to gather information that can be presented to developers in an IDE. The approach helps the developers in identifying and possibly removing inappropriate usage patterns of exception handling. During verification, the approach guides the testers in computing test requirements for exception handling and generating test data to exercise those interactions. We have presented empirical results to illustrate the occurrences of some inappropriate usage patterns of exception handling. In some of the cases, the approach helped us to identify parts of code in JABA where exception handling could be improved. Our study on the coverage of exception handling using existing test suites indicated that test suites that are not designed to cover exception handling will likely achieve poor coverage of exception handling. Our approach for computing test requirements specifically for exception handling can be used to help achieve adequate coverage of exception handling.

There are several directions for future work. First, future work could further investigate visualization techniques for presenting the information to developers. The schematic views presented in this paper represent one approach, which we are currently implementing in a tool. Second, future work could study the coverage of exception handling from other types of code-coverage-based test suites, such as test suites developed to cover statements or branches. Third, future work could investigate different approaches that would be useful for generating test data for covering exception handling. Future work could also evaluate our approach for ordering test requirements and identify additional parameters that can be used to make the ordering more effective. Finally, future work could investigate how the exception structure of a program evolves over time and affects the maintainability of the program.

References

- [1] L. O. Andersen. Program analysis and specialization for the C programming language. Technical Report 94-19, University of Copenhagen, 1994.
- [2] Aristotle Research Group. JABA: Java Architecture for Bytecode Analysis. <http://www.cc.gatech.edu/aristotle/Tools/jaba.html>, 2003.
- [3] J. Bloch. *Effective Java: Programming Language Guide*. Addison-Wesley, Reading, MA, 2001.
- [4] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, August 2001.
- [5] B. M. Chang, J. W. Jo, K. Yi, and K. M. Choe. Interprocedural exception analysis for Java. In *Proceedings of the 16th ACM Symposium on Applied Computing*, pages 620–625, March 2001.
- [6] R. Chatterjee and B. G. Ryder. Data-flow-based testing of object-oriented libraries. Technical Report DCS-TR-382, Department of Computer Science, Rutgers University, March 1999.
- [7] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. 27(2):1–25, February 2001.
- [8] C. Fu, R. P. Martin, K. Nagaraja, T. D. Nguyen, B. G. Ryder, and D. Wonnacott. Compiler directed program-fault coverage for highly available Java internet services. In *Proceedings of the 2003 International Conference on Dependable Systems and Networks*, pages 595–604, June 2003.
- [9] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, Reading, MA, 1996.
- [10] D. Liang and M. J. Harrold. Efficient computation of parameterized pointer information for interprocedural analyses. In *Proceedings of 2001 Static Analysis Symposium*, pages 279–298, July 2001.
- [11] J. Palsberg and M. Schwartzbach. Object-oriented type inference. In *Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications*, pages 146–161, October 1991.
- [12] J. Plevyak and A. Chien. Precise concrete type inference for object-oriented languages. In *Proceedings of the 9th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 324–340, October 1994.
- [13] D. Reimer and H. Srinivasan. Analyzing exception usage in large Java applications. In *Proceedings of the ECOOP 2003 Workshop on Exception Handling in Object-Oriented Systems: Towards Emerging Application Areas and New Programming Paradigms*, pages 10–19, July 2003.
- [14] M. P. Robillard and G. C. Murphy. Analyzing exception flow in Java programs. In *Proceedings of ESEC/FSE '99 7th European Software Engineering Conference and 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 1687 of *Lecture Notes in Computer Science*, pages 322–337. Springer-Verlag, September 1999.
- [15] B. G. Ryder, D. Smith, U. Kremer, M. Gordon, and N. Shah. A static study of Java exceptions using JSEP. In *Proceedings of the 9th International Conference on Compiler Construction*, volume 1781 of *Lecture Notes in Computer Science*, pages 67–81. Springer-Verlag, April 2000.
- [16] Marc De Scheemaeker. NanoXML: A small XML parser for Java. <http://nanoxml.n3.net>, 2002.
- [17] S. Sinha and M. J. Harrold. Analysis and testing of programs with exception-handling constructs. *IEEE Transactions on Software Engineering*, 26(9):849–871, September 2000.
- [18] N. Tracey, J. Clark, K. Mander, and J. McDermid. Automated test-data generation for exception conditions. *Software—Practice and Experience*, 30(1):61–79, January 2000.
- [19] K. Yi and B. M. Chang. Exception analysis for Java. In *Proceedings of the ECOOP '99 Workshop on Formal Techniques for Java Programs*, volume 1743 of *Lecture Notes in Computer Science*, pages 111–112. Springer-Verlag, June 1999.