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
{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, 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 ranging 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 difficult to understand and foresee. Abuses or misuses of such features can thus result in code that is faulty or difficult to understand, verify, and maintain [2, 9].

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 [11, 13].

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 static analyses include traditional techniques, such as type inference and data-flow analysis, and newly developed techniques. The dynamic analyses consist of new types of coverage analysis specifically defined to capture implicit control flow.

This paper presents our approach as it applies to exception handling. During development, our technique provides information about exception-related code entities (e.g., throw and 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 [9].

During testing, our approach provides guidance and support for 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 providing information about the requirements at each level.

Proceedings of the 26th International Conference on Software Engineering (ICSE'04)
0270-5257/04 $20.00 © 2004 IEEE
IEEE COMPUTER SOCIETY

To evaluate our approach, we implemented it in a prototype and performed empirical studies. The first set of studies assessed the presence of coding patterns that lead to complex implicit control flow in a set of real Java programs. The results show that such patterns are present in all the subjects we studied. The second set of studies 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/guide development and maintenance in the presence of implicit control flow based on both new and existing program-analysis techniques.
- A description of a prototype that demonstrates the feasibility of automating the approach.
- 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 Support for Software-Engineering Tasks

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

2.1 Definitions

Exceptions in Java. In Java, there are two main kinds of exceptions: checked and unchecked. A *checked exception* is explicitly thrown in the code. An *unchecked exception* is generated by the runtime system (e.g., access to an out-of-bounds array element). In Java, exceptions are regular objects and are raised (in the case of checked exceptions) using the `throw` statement (e.g., line 23, Figure 1). Exception handling in Java is supported by `try-catch-finally` sequences. A *try block* consists of a group of statements followed by one or more catch blocks, a finally block (e.g., lines 10–15, Figure 1), or both. A try block is executed until completion or until an exception is thrown. A *catch block* is associated with a try block and consists of a type and a set of statements (e.g., lines 12–13, Figure 1). A *finally block* is also associated with a try block and consists of a set of statements (e.g., lines 14–15, Figure 1). If an exception of type E is thrown within a try block, the catch blocks associated with it are checked for a type match. (A type match occurs when the type of the catch is E or a superclass of E.) If a match occurs, the body of the catch block is executed and the execution continues with the statement following the try block. Otherwise, the call stack is unwound until a catch block with a matching type is found or until the call stack is empty—in which case, the program terminates. If a finally block is present, its code is always executed after control leaves its associated try block, whether the try block terminates normally or is interrupted by an exception.

```
 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.

Finally Context. 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 `Sum.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.

IEEE COMPUTER SOCIETY

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) is reached within the 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. 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.

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 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. 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.

Precision and Safety. In Java, exceptions can be declared in the method interfaces using the `throws` clause—each method can declare the exceptions that can propagate outside 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 than the `throws` declarations are safe; however, even for checked exceptions, the `throws` declarations can be imprecise.

2.2 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[1] or Idea.[2] 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 [9] mention several inappropriate usage patterns of exception handling and provide evidence that the presence of those patterns complicates software reliability and maintainability. Unfortunately, such usage patterns are difficult to identify by simply examining the code or by using a purely syntactic analysis. 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 blocks, the developer can inspect them to determine whether they 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 2, 3, and 4 show a schematic view of the kinds of information provided at throw points, catch points, and throws declarations. (Although the approach is meant to be integrated within an IDE (*Integrated Development Environment*), 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 11 inappropriate coding patterns, related to exception handling (see Table 1). For space limitations, we discuss only four of these patterns in this paper—the ones in bold font. Reference [14] discusses all patterns.

[1] `http://www.eclipse.org/`
[2] `http://www.intellij.com/idea/`

0270-5257/04 $20.00 © 2004 **IEEE**
IEEE COMPUTER SOCIETY

Table 1. Inappropriate coding patterns.

Inappropriate coding pattern	Construct at which info. 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

Unreachable catch handlers. The presence of a statically unreachable catch handler may indicate a fault: the handler type may be incorrect, which could cause it to catch no exceptions. If this is not the case, the handler is simply useless 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 reachable—a catch handler that has no reaching exception types is statically unreachable.

Figure 2 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 type `java.lang.IllegalArgumentException` and `java.lang.RuntimeException`.

To determine whether a catch handler is unreachable because of an incorrect type (and is thus failing to handle exceptions), the developer can change the type of the handler to `java.lang.Throwable` (which, because all exceptions are subtypes of `java.lang.Throwable`, would cause the handler to catch any exceptions) and then compute the information about reaching types. If no types reach the handler, it can be removed. However, if some types reach the handler, using information about where those exception 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. In these cases, developers can, after identifying the ignored exceptions, modify the deactivation points.

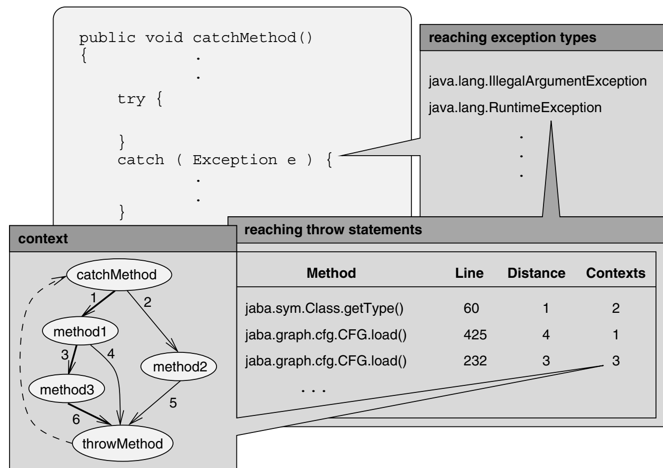

Figure 2. Schematic view of information about catch handlers that is provided using our approach.

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 3 shows the developer's interaction with an IDE to get this information: after selecting a throw statement, the IDE provides 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, for exception `e` of type `java.lang.RuntimeException`, one of which ignores the exception.

Large distance between throw and catch. In general, if exceptions are propagated a long way on the call stack, the exception handling may be less meaningful and debugging more difficult [9]. 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 abstraction. To avoid this problem, higher layers should map lower-level exceptions to exceptions that are explainable in terms of the higher-level abstraction [2].

Our approach supports developers in identifying throw–catch pairs whose distance may complicate maintenance

[3] In 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()`).

0270-5257/04 $20.00 © 2004 IEEE
IEEE COMPUTER SOCIETY

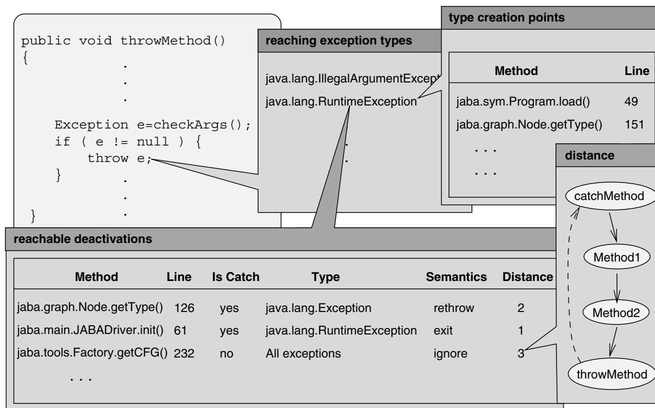

Figure 3. Schematic view of information about throw statements that is provided using our approach.

and testing. The approach supports this task by providing information about the distance between a reaching throw statement and a catch handler (Figure 2) and between a reachable deactivation and a throw statement (Figure 3).

Box reachable throw statements in Figure 2 shows how the information could be presented to developers and how they could further inspect throw–catch pairs with great distance. Developers can select each such throw–catch 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 great, 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 [9].

Imprecise or unsafe `throws` declarations. In Java, `throws` declarations alert a method's users to exceptions that can propagate out of the method. The presence of imprecise/unsafe exception declarations can provide misleading information to the API's clients: an imprecise exception declaration can cause the client code to contain unreachable catch handlers; an unsafe exception declaration can cause the client code to propagate 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 4 presents the method-interface information that can be provided using our approach. Using this information, 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.

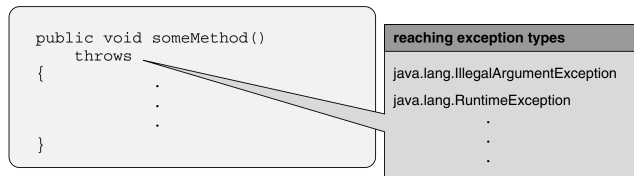

Figure 4. Schematic view of information about method interfaces that is provided using our approach.

2.3 Testing

During testing, our approach provides support in identifying test requirements[4] for exception handling and generating test data to satisfy the requirements. In this section, we first present techniques for generating test requirements to obtain levels of exception-handling coverage. Then, we discuss how our approach presents information to the testers to guide them in testing exception handling constructs.

2.3.1 Test requirements for exception handling

The presence of exception handling causes interactions that can be verified at different levels of thoroughness. Figure 5 presents levels of coverage for exception handling organized hierarchically—the higher levels require more thorough testing (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 simplest exception-handling coverage level, throw-statement coverage, requires coverage of the statements that contain a throw instruction. At this level, the test requirements ignore 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 coverage level, called `(throw)` in Figure 5, is shown at the bottom-left corner in the figure. Throw-statement coverage can be strengthened by (1) considering the types of exceptions that can be raised at a throw statement, and (2) considering the catch handlers that are reachable from a throw statement. These levels of coverage, called `(throw,type)` and `(throw,catch)` in Figure 5, require more thorough testing of exception handling than throw-statement coverage.

The arrows in Figure 5 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. `(throw,type,catch)` is a stronger level of coverage that subsumes both `(throw,type)` and `(throw,catch)`; it requires exercising a throw statement for each type

[4] A *test requirement* consists of an item (for code-based testing, one or more code entities) that must be exercised during testing and some constraints on how it must be exercised. For example, a test requirement for a (throw, catch) pair requires, to be satisfied, the execution of both the throw statement and the catch statement, such that the exception caught by the catch is the one raised by the throw.

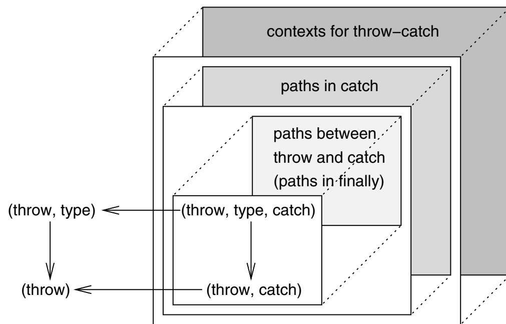

Figure 5. Levels of coverage for exception handling.

of exception that can be raised at the statement and for each catch handler that is reachable from the throw statement. (throw,catch) and (throw,type,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 within the catch blocks, and (3) by considering the relevant contexts for reaching throw statements.

2.3.2 Support for testing

Our approach guides testers in (1) exploring test requirements to select a suitable coverage level, and (2) generating test data to exercise the selected test requirements.

Select the level of coverage

Our approach uses static analysis to let testers interactively assess the effort required to attain various levels of exception-handling coverage. The approach presents the hierarchy (Figure 5) and provides information about 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 of test requirements that are covered at level n, the approach uses static analysis to compute a safe estimate of the percentage of test requirements that would be covered at level $n + 1$. Figure 6 presents the information provided to help 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 and, using the information provided by our approach, can select the appropriate level of coverage for the program under test.

Figure 7 presents a schematic view of the types of information that can be presented in an IDE during testing. 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

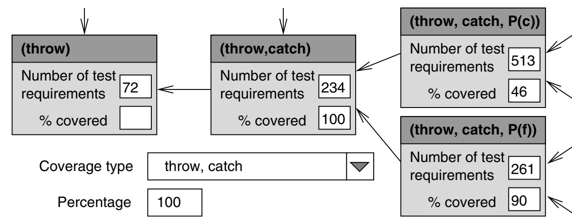

Figure 6. Information provided to testers to guide them in selecting a level of coverage.

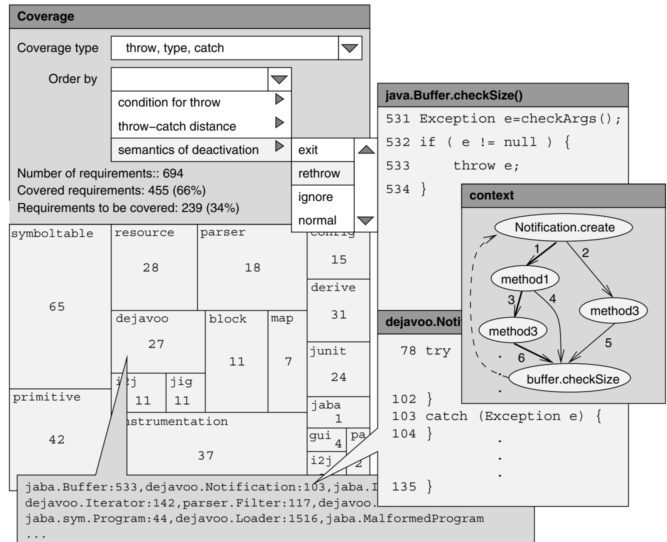

Figure 7. Schematic view of testing-requirements information for exceptions that is provided using our approach.

uncovered requirements. The lower part of the box on the left shows a treemap view[5] 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 7 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 develop test cases.

[5]The *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.

0270-5257/04 $20.00 © 2004 **IEEE**
IEEE COMPUTER SOCIETY

Table 2. Subject programs.

Subject	Description	Methods
DAIKON[6]	Dynamic invariant detector	7887
JABA[1]	Program analysis framework	2673
NANOXML[12]	XML parsing library	232
SIENA[3]	Publish subscribe system	195

2.4 Underlying Analyses

In this section, we briefly discuss the underlying analyses that lets us compute the information described in Sections 2.2 and 2.3. Reference [14] provides further details.

Exception flow analysis. We use exception-flow analysis to compute reaching exception types and throw statements at catch handlers and reachable deactivations at throw statements. The analysis first builds an interprocedural control flow graph (ICFG) that represents both intraprocedural and interprocedural control flow caused by exceptions [13]. Then, it traverses forward in the ICFG starting at each throw statement and computes reachable catch handlers and reachable deactivations within finally blocks.

Context analysis. To determine relevant contexts for a reaching throw statement, our approach builds the call graph of the program being analyzed. Using the call graph, we determine the set of methods that are both reachable from the method containing the catch and are reaching at the method containing the throw, and construct a reduced call graph. Finally, to enumerate the contexts, we compute acyclic paths in the reduced call graph. By examining the lengths of these paths, we compute the distance between a throw statement and reaching deactivation.

Type-inference analysis. Type-inference analysis (e.g., [8]) is required to determine the types of exceptions that can be raised at throw statements. 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 [13].

Coverage analysis. We use coverage analysis to identify which requirements are satisfied during testing. To this end, we instrument the code during execution so as to trace (1) execution of throw statements and type of exception(s) thrown, and (2) execution of catch blocks and type of exception(s) caught. This information is matched during executions to compute (throw,type,catch) tuples and identify which test requirements have been satisfied.

3 Empirical Results

To evaluate our approach, we built a prototype tool that identifies several of the inappropriate coding patterns mentioned in Section 2.2. The tool also computes test requirements and coverage according to the four basic coverage levels in Figure 5. 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 studied the coverage of exception handling by test suites generated using traditional testing techniques. For both studies, we used the subject programs listed in Table 2.

Table 3. Unreachable catch handlers.

Subject	Number of occurrences by considering Checked 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%)

3.1 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 because we are familiar with its code—the scenario in which 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) checked exceptions, and (2) all exceptions. Column 2 of the table shows the number of catch handlers that may be intended to handle only unchecked exceptions; such handlers are, therefore, unreachable if we analyze only checked 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 unchecked 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 DAIKON, contained handlers for only unchecked 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 unchecked exceptions and found that it was handling unchecked exceptions by accident; it was actually meant to handle exceptions propagated by another method in JABA. However, according to the analysis, the called method propagated no exceptions. On examining the called method, we realized that the exception declaration of the method was imprecise, that is, it listed exceptions that were not propagated by the method. Probably, the programmer who coded the calling method relied on the API documentation to determine potential exceptions and added a handler for those exceptions. Thus, using this information, we removed the unnecessary handler.

0270-5257/04 $20.00 © 2004 IEEE

IEEE COMPUTER SOCIETY

Table 4. Ignored exceptions.

Subject	Empty catch handlers	Deactivations in finally
DAIKON	19	0
JABA	2	0
NANOXML	2	0
SIENA	0	0

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 conditional control flow. In one case, a method that propagated an exception was being called from several different contexts and in all but one of those contexts the propagated exception represented an error. In that one context, the exception represented a condition that was expected, required no error recovery, and could be safely ignored. In another case, the exception being ignored should not have been ignored and 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. 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 they are 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 8 presents data about the occurrences of unsafe or imprecise exception declarations in method interfaces. For this study, we computed the data by considering only checked 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 unsafe and imprecise, respectively. 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.

Table 5. Distance between throws 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

The data in Figure 8 show that 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, because they provide misleading information to 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. However, listing all potential exception types for each method may be cumbersome, and this decision should be based on the visibility of the method to external clients and the ability of the clients to recover from the propagated exceptions. Using our approach, this process can be automated to present the list of potential exceptions to the developer, who can then decide which exceptions to list in the interface.

3.2 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 devel-

IEEE COMPUTER SOCIETY

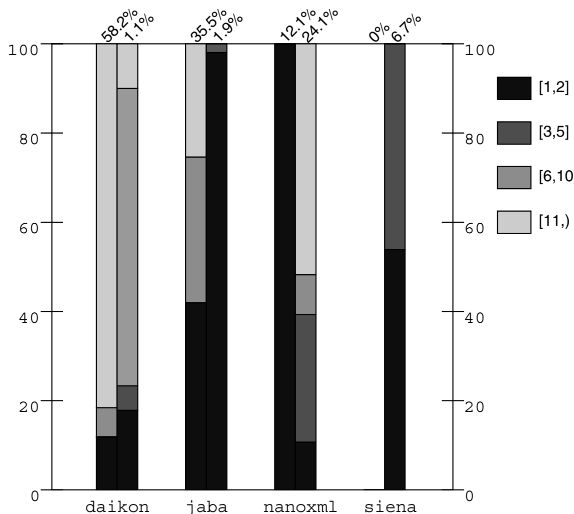

Figure 8. Imprecise or unsafe exception declarations in method interfaces.

opment of those applications or, for NANOXML, post-development, based on functional specifications. 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)` coverage level, for the subjects. Column 2 of Table 6 lists the number of test requirements for each subject. Next, we eliminated from the requirements all (throw, type, catch) tuples for which the test suites covered neither the throw method nor the catch method. In other words, we considered only requirements such that the test suite achieved 100% method coverage for the methods involved in the requirements (thus eliminating the risk of getting low coverage just because of a poor test suite). Column 3 of Table 6 lists the number of considered test requirements. As the data shows, few test requirements were eliminated—17% for DAIKON, 13% for SIENA, less than 1% for JABA, and none for NANOXML. Finally, we examined the percentage of considered (throw, type, catch) tuples that were covered by the tests (column 4 of Table 6). 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 explicitly considered to ensure adequate coverage.

4 Related Work

Several researchers have investigated the analysis of exception-handling constructs for development and maintenance. Robillard and Murphy [10] developed a tool that analyzes exception flow in Java programs to provide information for program understanding and detection of coding

Table 6. Coverage of throw-catch pairs.

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%)

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. Also, they do not investigate how the information about exception flow can be used to identify and eliminate inappropriate coding patterns. Finally, they do not address testing of exception handling.

Yi and Chang [4, 16] present a set-constraint-based approach for analyzing exception flow in Java programs to enable the identification of uncaught exceptions, and unreachable 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 [9] 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 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 of understanding exception flow. Also, our approach provides a more extensive classification of inappropriate coding patterns than their approach. Finally, they do not address testing of exception handling.

Other researchers have investigated the testing of exception-handling constructs. Chatterjee and Ryder [5] 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 [15] 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 sub-

0270-5257/04 $20.00 © 2004 **IEEE**
IEEE COMPUTER SOCIETY

set of those in Java. Additionally, their coverage criteria do not consider the interactions among the statements to generate the test requirements.

Fu and colleagues [7] present an approach for exercising catch blocks based on compiler-directed fault injection. They identify 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. Their approach is orthogonal to our approach for testing 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.

5 Summary and Future Work

We have presented an integrated and systematic approach for providing automated support for the development, maintenance, and testing 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 testing, the approach guides the testers in computing test requirements for exception handling and generating test data to satisfy those requirements. We have also 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 which exception handling could be improved. Our study on the coverage of exception handling using existing test suites indicated that test suites that are not explicitly designed to cover exception handling will likely achieve poor coverage of exception handling. Our approach for computing test requirements for exception handling constructs can be used to help achieve adequate coverage of such constructs.

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 Eclipse. Second, future work could investigate ways to generalize or modularize the underlying analysis so that the set of patterns can be easily extended to include user-defined patterns. Third, future work could investigate different approaches that would be useful for generating test data for covering exception handling. Fourth, future work could investigate ways to support automated or semi-automated code refactoring to eliminate inappropriate coding patterns. Finally, future work could investigate how the exception structure of a program evolves over time and affects the maintainability of the program.

Acknowledgments

This work was supported in part by National Science Foundation awards CCR-0306372, CCR-0205422, CCR-9988294, CCR-0209322, and SBE-0123532 to Georgia Tech. The anonymous reviewers provided useful feedback that helped us improve the paper. Michael Ernst provided access to DAIKON's CVS.

References

[1] Aristotle Research Group. JABA: Java Architecture for Bytecode Analysis. `http://www.cc.gatech.edu/aristotle/Tools/jaba.html`, 2003.

[2] J. Bloch. *Effective Java: Programming Language Guide*. Addison-Wesley, Reading, MA, 2001.

[3] 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.

[4] 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, Mar. 2001.

[5] 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, Mar. 1999.

[6] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. 27(2):1–25, Feb. 2001.

[7] 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.

[8] J. Palsberg and M. Schwartzbach. Object-oriented type inference. In *Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications*, pages 146–161, Oct. 1991.

[9] 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.

[10] 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, Sept. 1999.

[11] 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, Apr. 2000.

[12] M. D. Scheemaecker. NANOXML: A small XML parser for Java. http://nanoxml.n3.net, 2002.

[13] S. Sinha and M. J. Harrold. Analysis and testing of programs with exception-handling constructs. *IEEE Trans. Softw. Eng.*, 26(9):849–871, Sept. 2000.

[14] S. Sinha, A. Orso, and M. J. Harrold. Analysis and testing of programs with exception-handling constructs. Technical Report GIT-CC-03-48, Georgia Institute of Technology, Sept. 2003.

[15] N. Tracey, J. Clark, K. Mander, and J. McDermid. Automated test-data generation for exception conditions. *Software—Practice and Experience*, 30(1):61–79, Jan. 2000.

[16] 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.

0270-5257/04 $20.00 © 2004 **IEEE**

IEEE COMPUTER SOCIETY