Proceedings of the International Conference on Software Maintenance, Washington, D.C, November, 1998, IEEE Copyright

Analysis of Programs With Exception-Handling Constructs

Saurabh Sinha and Mary Jean Harrold

Department of Computer and Information Science
Ohio State University
Columbus, OH
E-mail: {`sinha,harrold`}`@cis.ohio-state.edu`

Abstract

Analysis techniques, such as control-flow, data-flow, and control-dependence, are used for a variety of maintenance tasks, including regression testing, dynamic execution profiling, and static and dynamic slicing. To be applicable to programs in languages, such as Java and C++ however, these analysis techniques should, to the extent possible, account for the effects of exception occurrences and exception-handling constructs. This paper presents techniques to construct intraprocedural and interprocedural representations on which existing techniques can be performed, and demonstrates their applicability to several maintenance tasks.

Keywords: *Exception handling, program analysis.*

1 Introduction

Many software testing and maintenance techniques, such as data-flow testing, test case generation, regression testing, dynamic execution profiling, and static and dynamic slicing (e.g., [4, 7, 8, 13]), require control-flow information. Much research has addressed the problems of computing such analysis information for individual procedures (*intraprocedural*) (e.g., [3]) and for interacting procedures (*interprocedural*) (e.g., [9]). Some of this research has addressed the problems of performing analyses for programs with transfers of control, such as `continue` and `goto` statements, that can affect the analyses at the intraprocedural level (e.g., [1]). Other research has addressed the problems of performing analyses for programs with transfers of control, such as **exit()** statements, that can affect the analyses at the interprocedural level [6]. To be applicable to programs in languages, such as Java [5] and C++[1], however, these analysis techniques should, to the extent possible, account for the effects of exception occurrences and exception-handling constructs.

Exception occurrences are either execution states that violate the semantic constraints of the language or explicit *throw* points that cause transfer of control to a location where the thrown exception is handled. *Exception handling* is a mechanism that provides a flexible framework within which programs may choose to handle such situations. Failure to account for the effects of exception occurrences and exception-handling constructs in performing analyses can result in incorrect analysis information, which in turn can result in unreliable software tools: a branch-coverage testing tool for Java that fails to recognize the flow of control among exception-handling statements cannot adequately measure the branch coverage of a test suite; a slicing tool for Java that fails to recognize the flow of control among exception-handling statements cannot accurately compute control-/data-dependence, which may result in incorrect slices.

The additional expense that is required to perform analyses that account for the effects of exception-handling constructs may not be justified unless these constructs occur frequently in practice. To determine the frequency with which Java programs use exception-handling statements, we conducted an initial study. In the study, we examined a variety of Java applications, and obtained a count of programs, from each application, that contained either a `try` statement or a `throw` statement. Table 1 summarizes the results of the study; for each subject, it describes the application, and lists the number of programs examined, and the usage of exception-handling statements.

As the table illustrates, 23.3% and 24.5% of the examined programs contained `try` and `throw` statements, respectively; within a subject, ignoring the values for `jlex`, these percentages varied from 12.6% to 37.5% for `try` statements, and 15.3% to 62.8% for `throw` statements. Over all subjects, 497 programs contained both `try` and `throw` statements. Therefore, there were 982 programs, which comprise 31.7% of all examined programs, that con-

[1] See `http://www.cygnus.com/misc/wp/` for ISO/ANSI C++ standard.

Table 1. Presence of Exception-Handling Statements in Java Programs.

Subject		Number of	Programs with	
Name	Description	Programs	`try`	`throw`
jacorb	ORB implementation	1062	271	229
javacup	LALR-parser generator	34	5	17
jdk	Sun's JDK 1.1.5	1256	342	372
jlex	Lexical-analyzer generator	1	1	1
swing	Sun's Swing API 1.0.2	692	87	106
tdb	Debugger for Java	8	3	5
toba	Java-to-C translator	43	13	27
Total		3096	722	757

tained either a `try` statement or a `throw` statement. This study supports our belief that the use of exception-handling constructs in practice is significant enough that it should be considered during various analyses.

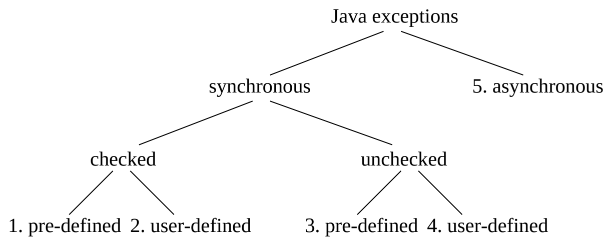
A number of existing techniques, such as those that perform type-inference for Java programs (e.g., [2]), use control-flow, data-flow, or control-dependence information. None of these techniques, however, presents representations that accommodate exception-handling — representations on which maintenance activities, such as regression test selection and slicing, can be performed. This paper presents techniques to construct both intraprocedural and interprocedural control-flow representations for programs that contain exeception-handling constructs. These representations can be used to perform maintenance activities such as regression test selection. These representations can also be used to perform other analyses, such as data-flow and control-dependence, which are required for maintenance activities such as slicing. We restrict our discussion to representations for programs that contain Java-like exception-handling constructs.

The main contributions of this paper are:

- An analysis of the effects of exception-handling constructs on control flow for Java programs (Sections 2 and 3).
- Techniques for creating control-flow graphs for individual and interacting methods that can be used for maintenance activities and other analyses (Section 3).
- A discussion of applications, such as structural testing and regression test selection, that use the representations for maintenance tasks (Section 4).

2 Exception Occurrences and Constructs

In Java, exceptions can be synchronous or asynchronous. *Synchronous* exceptions occur at particular program points and are caused by expression evaluation, statement execution, or explicit `throw` statements. Synchronous exceptions can be checked or unchecked: for *checked* exceptions, the compiler must find a handler or a signature declaration for the method that raises the exception; for *unchecked* exceptions, the compiler does not attempt to find such an associated handler or a signature declaration. Synchronous exceptions are further classified as pre-defined or user-defined: *pre-defined* exceptions are defined by the language; *user-defined* exceptions are defined by users of the language. For example, the `write()` method defined in `java.io.DataOutputStream` can raise a pre-defined checked exception, `IoException`, whereas, the method `pop()` defined in `java.util.Stack` can raise a pre-defined unchecked exception, `EmptyStack-Exception`. Users define a checked exception by extending `java.lang.Exception` or `java.lang.-Throwable`; similarly, users define an unchecked exception by extending `java.lang.RuntimeException` or `java.lang.Error`. *Asynchronous* exceptions occur at arbitrary, non-deterministic points in a program's execution, and are unchecked. Asynchronous exceptions occur when either the Java Virtual Machine raises an instance of `InternalError` (because of faults in the virtual-machine software, the host-system software, or the hardware), or a thread invokes the `stop()` method that raises an instance of `ThreadDeath` in another thread. The graph at the top of Figure 1 shows the types of Java exceptions.



```
try {
// guarded section
}
catch (ExceptionType1 t1) {
// handler for ExceptionType1
}
catch (ExceptionType2 t2) {
// handler for ExceptionType2
}
catch (Exception e) {
// handler for all exceptions
}
finally {
// cleanup code
}
```

Figure 1. Java Exception types (top), and the syntax of Java exception-handling constructs (bottom).

In Java, all synchronous, pre-defined exceptions (types 1 and 3) are raised as a result of expression evaluations, statement executions, or `throw` statements, whereas synchronous, user-defined exceptions (types 2 and 4) are raised by `throw` statements only.

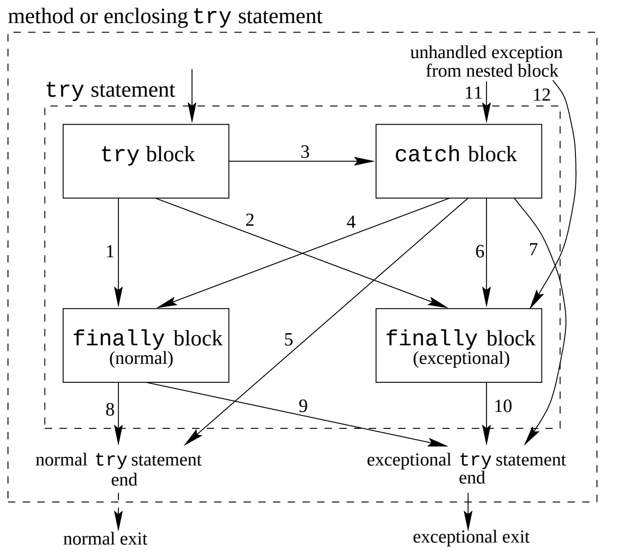
In Java, a `try` statement is the exception-handling construct. A `try` statement consists of a `try` block and, optionally, a `catch` block and a `finally` block — the legal constructs for a `try` statement are `try-catch`, `try-catch-finally`, and `try-finally`. The code at the bottom of Figure 1 shows a typical `try` statement. A `try` block (called the *guarded section*) is the set of statements whose execution must be monitored for exception occurrences. A `catch` block, which may be associated with each `try` block, is a sequence of `catch` clauses that specify exception handlers. Each `catch` clause defines the type of exception it handles, and contains a block of code that is executed when an exception of that type is caught. Each `try` statement can have a `finally` block that contains cleanup code. This cleanup code is always executed, regardless of the way in which control transfers out of the `try` block: by reaching the last statement in the try block (normal exit); through an exception that may or may not be handled in the associated catch block (exceptional exit); or by using `break`, `continue`, or `return` statements (normal exit). A `finally` block can itself raise an exception, which masks out the previous exception.

3 Control-Flow Analysis

In this section, we first examine the issues related to control-flow analysis for individual Java methods that contain exception-handling constructs, and describe the control-flow graph (CFG) for such methods. We next examine the impact of exception-handling constructs on interprocedural analysis. In discussing both intraprocedural and interprocedural control flow, we omit details of the construction algorithms; these can be found in [14]. Instead, we illustrate our representations with examples. We then discuss how type inferencing can affect the precision of representations, and the ways in which pre-defined exceptions can be analyzed. Finally, we present some preliminary results about the frequency with which exception-handling statements occur within individual Java programs.

3.1 Intraprocedural Analysis

When an exception is raised in a statement within a `try` block or in some method called within a `try` block, control transfers to the `catch` block associated with the last `try` block in which control entered, but has not yet exited. This `catch` block is the nearest *dynamically-enclosing* `catch` block [5], and can be in the same `try` statement, in an enclosing `try` statement, or in a calling method. If a matching `catch` handler is found, the parameter of that handler is initialized with the thrown object and the handler code executes; following the execution of the handler code, normal execution resumes at the first statement following the


method or enclosing try statement
unhandled exception from nested block
try statement
11
12
try block
3
catch block
2
4
1
6
7
5
finally block (normal)
finally block (exceptional)
8
9
10
normal try statement end
exceptional try statement end
normal exit
exceptional exit

1. no exceptions raised in try block
2. exception raised in try block; no catch block
3. exception raised in try block; catch block specified
4. exception handled; finally block specified
5. exception handled; no finally block
6. exception not handled; finally block specified
7. exception not handled; no finally block
8. no exceptions raised in finally block
9. exception raised in finally block
10. finally block propagates previous exception or raises another exception
11. unhandled exception from nested block; catch block specified
12. unhandled exception from nested block; no catch block; finally block specified

Figure 2. Control flow in Java exception-handling constructs.

`try` statement where the exception was handled — control does not return to the point where the exception was raised. If no matching `catch` handler is found in the nearest dynamically-enclosing `catch` block, the search continues in the `catch` block of the enclosing `try` statement, and subsequently in some calling method. Before control exits a `try` statement, the `finally` block of the `try` statement is executed, if it exists, regardless of whether control exits the `try` statement with an unhandled exception.

The block-level control-flow graph, shown in Figure 2, illustrates the control flow into and out of a `try` statement. The figure shows a `try` statement and its component blocks; the conditions triggering the control flow between the blocks are numbered and listed next to the figure. In the following, each type of path within a `try` statement is referenced by the edges that it traverses in the block.[2]

Path 1-8 is taken if the `try` and the `finally` blocks raise

[2]The partial Java program shown in Figure 3 illustrates two of the paths within a `try` statement: an execution sequence that covers lines 4, 5, 6, 8, 9, 16, and 17 of the program illustrates *Path 1-8* within a `try` statement; an execution sequence that covers lines 4, 5, 6, 7, 14, 15, 16, and 17 illustrates *Path 3-4-8* within a `try` statement.

no exceptions.

Path 1-9 is taken if the `try` block completes normally, but the `finally` block raises an exception.

Path 2-10 (12-10) is taken if the `try` block raises an exception (or a nested block raises an exception that is not handled in that block — a nested block *propagates* an exception) and there is no corresponding `catch` block, which causes control to flow directly to the `finally` block.

Path 3-4-8 (11-4-8) is taken if the `try` block raises an exception (or a nested block propagates an exception), the `catch` block handles the exception, and the `finally` block raises no additional exceptions.

Path 3-4-9 (11-4-9) is taken if the `try` block raises an exception (or a nested block propagates an exception) that is handled in the `catch` block, and then the `finally` block raises another exception.

Path 3-5 (11-5) is taken if the `try` block raises an exception (or a nested block propagates an exception) that is handled in the `catch` block, and no `finally` block is specified.

Path 3-6-10 (11-6-10) is taken if the `try` block raises an exception (or a nested block propagates an exception) and the `catch` block fails to handle the exception, or the handler code raises a new exception — the raised exception is pending when control enters the `finally` block. The `finally` block either propagates this exception or raises another exception that masks out the previous one.

Path 3-7 (11-7) is taken if the `try` block raises an exception (or a nested block propagates an exception) and the `catch` block fails to handle the exception or the handler code raises a new exception, and no `finally` block is specified.

Within a `catch` block, all handlers are examined, in the order in which they appear, for a handler that is a supertype of a raised exception; no priority is given to an exact match over one requiring the application of an inheritance relationship. A raised exception, E, matches a catch handler, T, if (1) E and T are of the same type, or (2) T is a superclass of E.

In generating control-flow representations for exception-handling constructs, we connect a throw-statement node directly to the catch-clause node for each `catch` handler that may handle the raised exception. Therefore, the search process for a matching handler described earlier in this section, which is reflected in the actions of the Java Virtual Machine when it encounters an exception, is not reflected in our control-flow representation; rather the decisions involved in the handler search are made implicitly during the graph construction process, so that the graph representation is a result of those decisions. We call such a representation a *precise* representation. An alternative representation, which can be considered *imprecise*, may explicitly represent the handler-matching process in the graph representation. Such a graph, therefore, would have additional predicate nodes that are used to match the type of a thrown exception with the type of a `catch` handler. Although the imprecise representation is less expensive to generate, it can create spurious dependencies that do not exist in Java programs, and may increase the graph size; we, therefore, do not consider the imprecise representation further in this paper.

Two implications of generating a precise representation are that (1) if the statement where an exception is raised is to be directly connected to a matching handler, the types of exceptions raised at that statement must be known, and (2) because an exception can be handled outside the method where it is raised, a throw-statement node in the CFG for the method may have no out-edges; such edges are created when individual CFGs are connected to form an interprocedural control-flow graph.[3]

Our CFG represents each `catch` clause by a catch-assignment node that initializes the formal parameter of the `catch` clause with the actual object mentioned in a `throw` statement; the catch-assignment node is followed by nodes that represent statements in the handler code.

To illustrate, consider the partial Java program shown in Figure 3. This code opens a connection to a database (line 3), selects a row from the database (line 5), updates values for some fields (line 8), and commits the updated values back to the database (line 9). If the select command fails, an exception is thrown. The `catch` block lists this exception along with other update exceptions. The `finally` block closes the database connection irrespective of whether an exception is raised. The relationships among the exception types mentioned in the `catch` clauses are as follows: `UpdateException` and `SelectException` are siblings in the exception-inheritance hierarchy (with parent `java.lang.Exception`); `PkConstraint-Exception` is a child of `UpdateException`. In Figure 3, nodes 10, 12, and 14 represent catch-assignment nodes for the three `catch` handlers. One of the catch-assignment nodes, node 14, has an in-edge because the corresponding exception, `SelectException`, is raised within the same method, in line 7; the other two catch-assignment nodes, nodes 10 and 12, have no in-edges because the corresponding exceptions are not raised in the same method. In Section 3.2, we present an implementation of the `update()` method invoked in line 9 that raises exceptions caught by the handlers in lines 10 and 12.

The `finally` block is replicated in Figure 2 to illustrate that control can reach it under normal or exceptional circumstances. In our representation, we treat a `finally`

[3]Analogously, in the CFG for a method, a catch-clause node may have no in-edges if all exceptions handled by that `catch` clause are raised in methods other than the one containing the `catch` clause; for example, see Figure 3.

```
1  DatabaseConnection dbConn = new DatabaseConnection();
2  String sqlCmd; Array row;
3  dbConn.open();
4  try {
5    row = dbConn.select(sqlCmd);
6    if (row == null) {
7      throw new SelectException();
     }
8    sqlCmd = updateFields(row);
9    dbConn.update(sqlCmd);
   }
10 catch (PkConstraintException pce) {
11   showMessage(pce);
   }
12  catch (UpdateException ue) {
13   showMessage(ue);
   }
14 catch (SelectException se) {
15   showMessage(se);
   }
16 finally {
17   dbConn.close();
   }
18 . . .
```

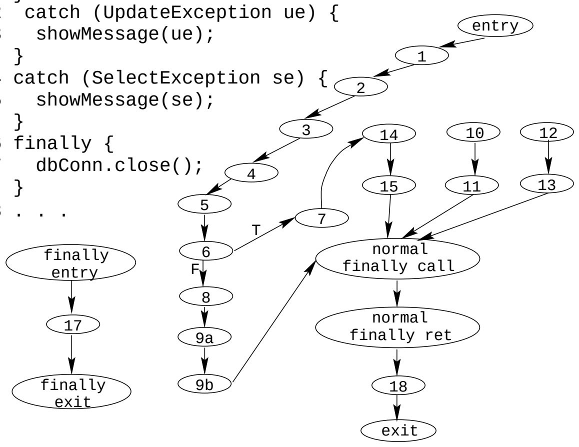


Figure 3. Partial Java program and its control-flow graph.

block as a separate method (with a distinct CFG) that is invoked under several conditions: (1) when control reaches the end of a `try` block or the end of a `catch` clause, (2) when control leaves a `try` statement because of an unhandled exception, and (3) when control leaves a `try` statement because of a transfer-of-control statement, such as `break`, `continue`, or `return`. The invocation of `finally` blocks under various conditions results in the addition of several finally-call nodes to the CFGs of the methods that contain those `finally` blocks: we refer to the finally-call nodes created for condition (2) as *exceptional*-finally-call nodes, and those for conditions (1) and (3) as *normal*-finally-call nodes.[4]

For condition (1), the CFG has a call site to the `finally` block; as illustrated in Figure 3, this normal-finally-call node for condition (1) is reached at the end of a `try` block (node 9b), and at the end of each associated `catch` clause (nodes 11, 13, and 15). The number of normal-finally-call nodes created in a CFG for condition (1) is the same as the number of `finally` blocks in the corresponding method.

For condition (2), the CFG for a method may contain several exceptional-finally-call nodes; these call nodes are reached along paths where an unhandled exception is propagated. The number of exceptional-finally-call nodes created for a method depends on both the number of exception types raised by a method and the number of `finally` blocks in a method. A method propagates an exception if it raises but does not handle the exception; an invocation of such a method can, therefore, result in an exception in the calling method. A method raises an exception if it either contains a `throw` statement or contains a call site where the called method propagates an exception; in the former case the method *directly* raises an exception, whereas in the latter case, it *indirectly* raises an exception.[5] The number of exceptional-finally-call nodes created for a method M is bounded by the product of number of the `finally` blocks in M and distinct types of exceptions raised by M. For each distinct type of exception raised by a method, we create separate exceptional-finally-call nodes corresponding to `finally` blocks that enclose the exception-raise point. The reason for creating separate exceptional-finally-call nodes for distinct exception types is that we intend to directly connect a statement raising an exception to the handler that catches that exception, and distinct exception types may be caught by distinct handlers. In general, the number of exceptional-finally-call nodes in a CFG is less than the above product; two factors contribute to this: first, a raised exception may be caught before the handler-matching process results in invocation of `finally` blocks in the method raising the exception, and second, an exception may be raised at a point in a method where it is not enclosed by a `finally` block in that method.

For condition (3), the CFG for a method contains several normal-finally-call nodes; these call nodes appear along paths where a transfer-of-control statement, such as `break`, `continue`, or `return`, is encountered within a `try` statement that is enclosed by a loop. In such cases, following the CFG node for the transfer-of-control statement, we create zero or more normal-finally-call nodes, and connect the last normal-finally-call node to the destination of the transfer-of-control statement. The number of normally-finally-call nodes created in a CFG for condition (3) depends on the number of transfer-of-control statements that appear within `try` statements (that are enclosed by loops), and the number of `finally` blocks in the corresponding method. A transfer-of-control statement within a `finally` block produces a similar behavior as a transfer-of-control statement within a `try` block or a `catch` block in that one or more enclosing `finally` blocks may be called; following these calls, however, the edge that leads to the the destination of the transfer-of-control statement is an interprocedural edge because we create a separate CFG for a `finally` block, and that edge is, therefore, created by a

[4]For each finally-call node, we also create a corresponding finally-return node, just as we would for a regular call site.

[5]Similarly, we can define the exceptions propagated by a method as those that are *directly* propagated by the method, and those that are *indirectly* propagated by the method.

separate algorithm (that builds interprocedural representations) rather than the CFG-construction algorithm [14].

As Figure 2 illustrates, there are two modes of exit for every `try` statement and every method: *normal exit*, where control leaves with no pending exceptions, and *exceptional exit*, where control leaves with a pending exception. If a `try` statement exits normally (edges 5 or 8 of Figure 2), control flows to the next sequential statement following the `try` statement, whereas if a `try` statement exits exceptionally (edges 7, 9, or 10 of Figure 2), control flows to an exceptional method exit. An exceptional exit from a method signifies that the method propagates an exception; therefore, from an exceptional method exit, control flows to the closest `catch` block that dynamically encloses a call to such a method. In the CFGs, throw-statement nodes or exception-finally-return nodes that have no out-edges indicate exceptional exits from the corresponding methods; following such nodes, we create exceptional-exit nodes. A CFG for a method contains as many exceptional-exit nodes as there are distinct types of exceptions that are propagated by that method.

3.2 Interprocedural Analysis

In considering interprocedural control flow for Java programs, our concern is to characterize the interprocedural control flow induced by exception-handling constructs only; therefore, we do not address issues relating to interprocedural representations of dynamically-dispatched function calls. Such function calls can be handled, for example, by creating decision vertices as described in [10].

An interprocedural control-flow graph (ICFG) for a program $\mathcal{P}$ consists of CFGs for every procedure P in $\mathcal{P}$. At each call site, the call node is connected to the entry node of the called procedure by a call edge, and the exit node of the called procedure is connected to the corresponding return node by a return edge. Exception propagation on the call stack from a called method to a calling method introduces additional interprocedural edges between methods in an ICFG, apart from the usual call and return edges. Our CFG-construction algorithm saves pertinent information about call sites that helps in the creation of these additional interprocedural edges [14].

Figure 4 shows an interprocedural block-level control-flow graph (similar to Figure 2) that shows the called method, B, at the top, and its caller, A, below it. The call to B within A's `try` block is shown by a call edge. If B completes normally, control returns to A through a normal exit from B. However, if B propagates an exception, control returns from B through an exceptional exit. Following an exceptional exit from B, control flows to one of three places: (1) if A's `try` block has an associated `catch` block where a handler catches the exception raised in B, control flows to the catch-assignment node for that handler; (2) otherwise, if

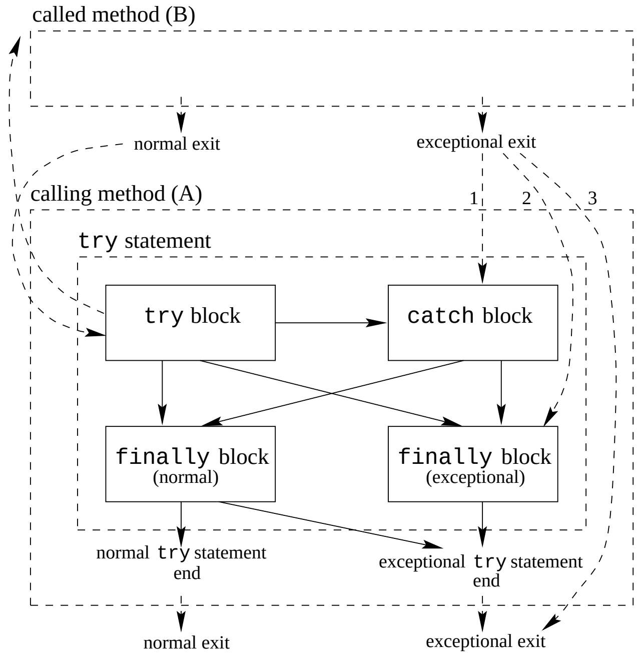


Figure 4. Interprocedural control flow in exception-handling constructs.

A's `try` block has a corresponding `finally` block, control flows to that `finally` block (in the ICFG this is represented as an exceptional-finally-call node; (3) if neither of the above is true, method A also exits exceptionally, and the search for a handler continues in a caller of A.

Thus, in the presence of exception-handling constructs, an ICFG contains additional interprocedural edges that originate at an exceptional-exit node of a called method, and terminate at a catch-assignment, an exceptional-finally-call, or an exceptional-exit node in a caller of that method. We call such edges *interprocedural-exception* edges.

Another kind of interprocedural edge that appears in an ICFG for a program with exception-handling constructs results from our treatment of `finally` blocks as separate methods: if a `finally` block contains a transfer-of-control statement, an *interprocedural-transfer* edge connects the node for that transfer-of-control statement to a node in the CFG for the method that contains that `finally` block.

Figure 5 shows an implementation of the `update()` method for the partial Java program. This method accepts an SQL command as a parameter, and executes it in the database. If the command fails, `update()` throws an exception based on the cause of the failure. The ICFG for the partial Java program is also shown in Figure 5. The `throw` statement in line 26 of `update()` causes an exceptional exit from `update()`; because the `throw` statement in line 26 can potentially raise two types of exceptions, `PkConstraintException` and

```
19 update(String cmd) throws UpdateException {
20   UpdateException u;
21   if ( (status=executeCmd(cmd)) != 0 ) {
22     if (status == 1) {
23       u = new PkConstraintException();
       }
24     else {
25       u = new UpdateException();
       }
26     throw u;
     }
   }
```

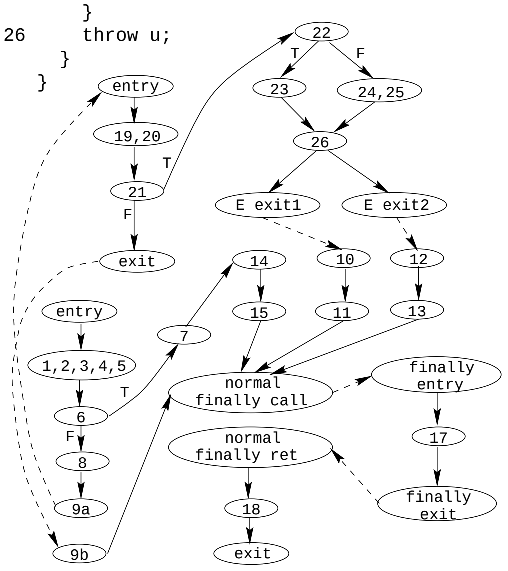


Figure 5. `update()` method and interprocedural control-flow graph for partial Java program.

`UpdateException`, node 26 in the ICFG is connected to two exceptional-exit nodes — the one labeled 'E exit1' corresponds to `PkConstraintException`, and the other, labeled 'E exit2', corresponds to `UpdateException`. Using the link information saved for the call site that calls `update()`, the ICFG-construction algorithm creates interprocedural-exception edges to connect exceptional-exit nodes in `update()` to catch-assignment nodes in the caller of `update()`.

3.3 Precision of Control-flow Representations

Our control-flow representations for exception-handling constructs assume that some suitable algorithm is used to infer the types of exceptions that can be raised at various program points. The precision of our representations depends to an extent on the precision of such a type-inference algorithm. In our representations, each throw-statement node has as many out-edges as the types of exceptions that can be raised at the corresponding `throw` statement — there is one out-edge for each exception type. Therefore, an imprecise (but safe) estimation of exception types would result in additional edges emanating from throw-statement nodes, such that paths starting along such edges are infeasible.

Type-inference algorithms (e.g., [11, 12]) attempt to determine the types for each expression in a program by solving type constraints, or propagating local type information through a program. Such techniques have traditionally been applied to optimization of dynamically-dispatched function calls. Recent work [2] uses context-sensitive points-to analysis to infer types in programs that contain exception-handling constructs.

Our approach to determining types for exceptions avoids exhaustive propagation of type information through a program. Rather, at the point in the CFG construction when an AST node representing a `throw` statement is encountered, our CFG-construction algorithm invokes a routine that computes types for exceptions raised at the corresponding `throw` statement. Our type-inference algorithm [14] performs a reverse data-flow analysis starting from the throw-statement node in the partially-constructed CFG; on reaching a call site or the entry node of the CFG, the algorithm makes a conservative estimate of possible types and does not continue analysis into a called or a calling method. Our approach is, thus, demand-driven, and precise in the current method — it performs reverse data-flow analysis from a point where the desired information is required, as opposed to an exhaustive analysis, but it only performs this analysis in the current method. Our type-inference algorithm is also flow sensitive because it performs precise analysis in the method containing a `throw` statement, but is context insensitive because it makes conservative assumptions at a call site, a `catch` handler, or the method entry.

Some initial data that we gathered leads us to believe that in practice, an exhaustive type-inference algorithm may not be required to determine types for exceptions. We examined each `throw` statement in programs from the groups listed in Table 1: out of 2752 `throw` statements that appeared in the subject programs, 2618 mentioned a new-instance expression (for example, line 7 in Figure 3); of the remaining 134 `throw` statements, 114 mentioned a variable (for example, line 26 in Figure 5), whereas 20 mentioned a method call. Therefore, for 95.1% of the `throw` statements from our subjects, type inference is trivial.

Based on further experimentation, which would reveal the level of imprecision in the types inferred by by our type-inference algorithm (for cases where a `throw` statement does not mention a new-instance expression), we can consider extending the algorithm so that it performs additional analyses on reaching the entry node, a call site, or a catch-assignment node.

3.4 Handling Pre-Defined Exceptions

The preceding discussions apply to programs in which the exception-raise point can be statically determined from the source code; thus the discussions cover exceptions that are raised explicitly, by `throw` statements, rather than implicitly, by an expression evaluation, a statement execution, or by the Java Virtual Machine.

Exception types 1, 3, and 5 from Figure 1 may be raised without `throw` statements: exception types 1 and 3 may be raised by messages sent to classes defined in the Java API, whereas exception type 5 may be raised by a message send (a `stop()` message from one thread to another) or by the Java Virtual Machine. To generate control-flow representation for exception types 1 and 3, we must determine those messages that can potentially raise an exception. Following such a message send, we can create a predicate node that decides whether that message raises an exception or not. The true branch from this predicate node is connected to the statement immediately following the message send, whereas the false branch is processed similar to a `throw` statement, by creating a dummy throw-statement node.

A simple, and conservative approach to determine those messages that can raise an exception may treat every message as potentially exception-raising, and create a predicate node after it. This approach would increase the CFG size significantly, however, given that object-oriented programs contain frequent message sends.

To restrict the number of messages considered as candidates for potentially raising-exceptions, we can consider two techniques. First, we can use a static approach that inspects the signature of the method called at a message send, and determines whether or not the invoked method declares any exceptions. Such a static approach will account for all pre-defined checked exceptions because such exceptions are declared in the signature of standard library methods that raise those exceptions. Thus, for example, the `write()` method in `java.io.DataOutputStream` (in the example from Section 2) declares `IoException` in its signature.

Second, we can use a dynamic-feedback approach to identify messages that raise exceptions. Data about exceptions raised at every method call could be gathered during executions of a program, and then used to generate a more precise control-flow representation for that program. The code to gather such data may have to be implemented in the Java Virtual Machine; this needs to be investigated. A dynamic approach can enable us to identify messages that can raise pre-defined unchecked exceptions. Such an approach, however, is unsafe: it will identify only those methods that were found to raise exceptions on particular executions of a program; thus, there can be methods that, although they can potentially raise pre-defined unchecked exceptions, are not identified by the dynamic-feedback approach.

An asynchronous exception can potentially be raised after any statement in a program and this may vary from one execution of the program to the next. Asynchronous exceptions typically have a bounded delay associated with their detection; for example, the virtual machine may poll for such exceptions at the point of each control-transfer instruction (at the bytecode level) [5]. A conservative estimate of exception-raise points for asynchronous exceptions, therefore, determines a very large number of such points in the program.

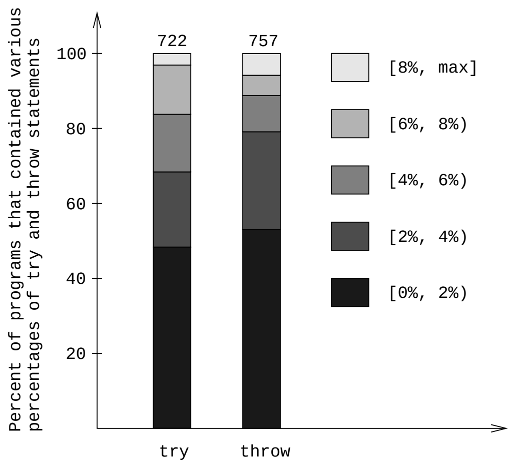


Figure 6. Frequency of occurrence of `try` and `throw` statements in programs that contained `try` and `throw` statements, respectively.

3.5 Preliminary Results

Table 1 shows how often our subjects contain `try` and `throw` statements, but it does not indicate how frequently such statements occur within specific Java programs. Thus, for example, Table 1 shows that 722 Java programs from our subjects contain `try` statements, but it does not indicate how many `try` statements appear in those 722 programs; gathering such data was the purpose of our second study.

Figure 6 shows the frequency of occurrence of `try` and `throw` statements as a segmented bar graph; the vertical axis shows the percentages of programs, and the segments show the various percentages of `try` and `throw` statements. The segmented bar for `try` statements illustrates the distribution of the percentages of `try` statements in programs. For example, in approximately 13% of the 722 programs (that contained `try` statements), `try` statements accounted for 6% to 8% of the statements in those programs. Similarly, the segmented bar for `throw` statements illustrates the distribution of the percentages of `throw` statements in programs. The number of `try` statements in our subjects was 3038, whereas the number of `throw` statements was 2752. The comparison of the two segmented bars in Figure 6 is not meaningful.

Figure 7 gives averages of the percentages of `try` and `throw` statements by each subject; the averages are over programs from a subject that contained `try` statements, or those that contained `throw` statements, and not over all

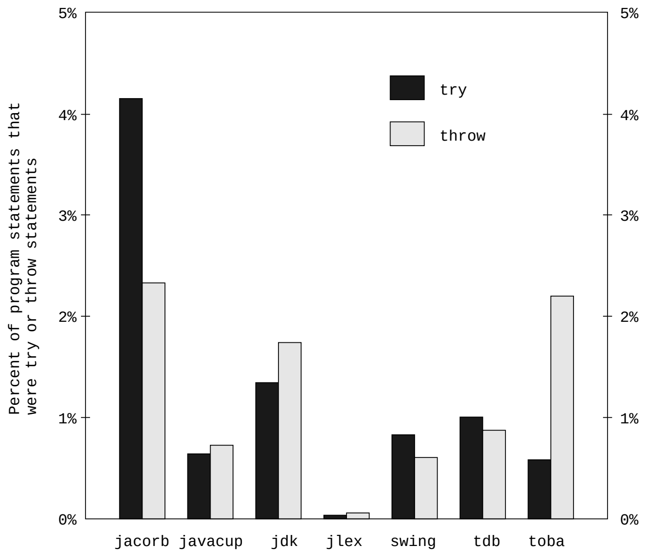


Figure 7. Frequency of occurrence of `try` and `throw` statements in programs from each subject.

programs from that subject. The figure shows that in the `jacorb` programs that contained `try` statements, on average, there was one `try` statement for every 24 statements, and in the programs that contained `throw` statements, there was a `throw` statement for every 43 statements. When averaged across all subjects, a `try` statement appeared in every 58 statements, and a `throw` statement appeared in every 68 statements.

Although these results in themselves are not indicative of the extent to which exception-handling statements may impact analysis data, they strengthen our belief that the impact may be significant. We are currently implementing a prototype to generate control-flow representations for programs that contain exception-handling constructs. Such a tool would facilitate future experiments to study the impact of exception-handling statements on program-analysis data.

4 Applications of the Representations and Analyses

The representations that we presented in the previous sections can be used for many different testing and maintenance tasks. This section discusses briefly two such tasks: structural testing and regression test selection; for brevity, we omit the details of these tasks. For each task, we first describe the task, and then illustrate how using an alternative control-flow representation (that ignores control-flow among exception-handling statements) affects the task. We assume that the alternative control-flow representation does not connect a throw-statement node to an appropriate node.

4.1 Structural Testing and Coverage

Structural testing techniques use a program's structure to guide the selection of test cases. For example, in branch testing, test cases are developed by considering inputs that cause certain branches in the program under test to be executed; similarly, path testing considers test cases that execute certain paths in the program. *Structural coverage* techniques give a measure of how well the structural elements of the program are executed by a given test suite. For example, the branch-coverage measure of a test suite is the percentage of branches in the program that are executed by the test cases in the test suite.

Certain coverage criteria, such as branch coverage, basis-path coverage, and path coverage, consider edges in a CFG to build test suites and to measure the coverage provided by a test suite. Such criteria are, therefore, affected by edges that appear in a CFG. Our CFGs represent execution paths from `throw` statements to `catch` handlers, and paths that lead to the execution of `finally` blocks. Therefore, using our representation for structural testing provides an accurate measure of the adequacy of a test suite. The alternative representation, however, can provide misleading information about the coverage provided by a test suite, because it does not consider all branches and paths in the program.[6] Other coverage criteria, such as statement coverage, that do not consider edges in a CFG, are not affected by the representation for exception-handling statements.

The precision of the type-inference algorithm used to generate our CFGs has an impact on structural testing: an imprecise type-inference algorithm results in infeasible branches in a program, and therefore, no test suite for edge-based coverage would provide 100% coverage for the resulting representation.

4.2 Regression Test Selection

Regression testing is an expensive testing procedure that attempts to validate modified software and ensure that new errors are not introduced into previously tested code. One method for reducing the cost of regression testing is to save the test suites that are developed for a product, and reuse them to revalidate the product after it is modified. A *retest-all* approach reruns all such test cases, but may consume excessive time and resources. *Regression test selection techniques*, in contrast, attempt to reduce the cost of regression testing by selecting a subset of the test suite that was used during development and using that subset to test the modified program.

[6]For branch coverage, if a test suite provides 100% coverage using the alternative representation, that test suite also provides 100% coverage in our representation. This is not true for path coverage and basis-path coverage.

Rothermel and Harrold developed a regression test selection technique [13] that uses a program's CFG. Their technique first creates the CFG of the original program. Then, it uses this CFG to insert probes into the program so that, when the instrumented program is executed with a test case in the test suite, it reports those edges that were executed. After this instrumented version of the program is executed with all test cases in the test suite, the technique creates a test history that associates with each edge, those test cases that executed it. When the program is modified, the technique creates the CFG for the modified program. The technique then walks the CFGs for the original and modified programs in a depth-first manner until, upon considering like-labeled edges in these CFGs, it finds that the text associated with the sink of the edge in the original program differs from the text associated with the sink of the edge in the modified program. At this point, the technique selects all test cases that, according to the test history, are associated with that edge.

This technique can be used on our CFGs to select appropriate tests when exception-handling statements are modified. Using the alternative control-flow representation, however, the technique would fail to detect modifications made to `catch` handlers because in the alternative representation catch-assignment nodes do not have in-edges. Although the problem can be alleviated by specifying regression-testing criteria specific to exception-handling statements, such as select tests that exercise a deleted or a modified `catch` handler, such an approach may select tests that will not reveal faults.

5 Conclusions

We have presented techniques to create intraprocedural and interprocedural representations for Java programs that contain exception-handling constructs. We discussed ways in which precision of representations can be improved, and program points where pre-defined exceptions are raised can be determined. Our study suggests that in practice there may be little gain in precision by an exhaustive type-inference algorithm to determine types for exceptions. We also gave an overview of how these representations can be used to perform other analyses such as structural testing and regression test selection.

Our study indicates frequent usage of exception-handling statements in Java programs. Further experimentation is required, however, to determine the impact of exception-handling statements on control flow and other applications of control-flow representations. Such experimentation can also examine the trade-offs involved in representing or ignoring exception-handling statements, and the precision of such representations.

6 Acknowledgements

This work was supported in part by NSF under NYI Award CCR-9696157 and ESS Award CCR-9707792 to Ohio State University. Chaitanya Kodeboyina contributed to the intraprocedural control flow analysis. Gerald Baumgartner, Gregg Rothermel, Jim Jones, and Donglin Liang made many helpful suggestions that improved the presentation of this work.

References

[1] H. Agrawal. On slicing programs with jump statements. In *Proc. of the ACM SIGPLAN '94 Conf. on Prog. Lang. Design and Impl.*, pages 302–12, June 1994.

[2] R. K. Chatterjee, B. G. Ryder, and W. A. Landi. Complexity of concrete type-inference in the presence of exceptions. *Lecture Notes in Computer Science*, 1381:57–74, Apr. 1998.

[3] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. on Prog. Lang. and Sys.*, 9(3):319–349, July 1987.

[4] P. Frankl and E. J. Weyuker. An applicable family of data flow criteria. *IEEE Trans. on Softw. Eng.*, 14(10):1483–1498, Oct. 1988.

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

[6] M. J. Harrold, G. Rothermel, and S. Sinha. Computation of interprocedural control dependence. In *Proc. of the ACM Int'l. Symp. on Softw. Testing and Analysis*, March 1998.

[7] M. J. Harrold and M. L. Soffa. Selecting data for integration testing. *IEEE Softw.*, pages 58–65, Mar. 1991.

[8] B. Korel. Automated software test data generation. *IEEE Trans. on Softw. Eng.*, 16(8):870–879, Aug. 1990.

[9] W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proc. of SIGPLAN '92 Conf. on Prog. Lang. Design and Implem.*, pages 235–248, June 1992.

[10] L. Larsen and M. J. Harrold. Slicing object-oriented software. In *Proc. of 18th Int'l Conf. on Softw. Eng.*, pages 495–505, Mar. 1996.

[11] J. Palsberg and M. Schwartzbach. Object-oriented type inference. In *Proc. of Object-Oriented Prog. Sys., Lang. and Appl.*, pages 146–161, Oct. 1991.

[12] J. Plevyak and A. Chien. Precise concrete type inference for object-oriented languages. In *Proc. of Object-Oriented Prog. Sys., Lang. and Appl.*, pages 324–340, Oct. 1994.

[13] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Trans. on Softw. Eng.and Meth.*, 6(2):173–210, Apr. 1997.

[14] S. Sinha and M. J. Harrold. Static analysis of programs with exception-handling constructs. Technical Report OSU-CISRC-7/98-TR25, The Ohio State University, 1998.