Analysis of Programs with Exception-Handling Constructs

Saurabh Sinha and Mary Jean Harrold
Department of Computer and Information Science
The Ohio State University
395 Dreese Lab
Columbus, OH 43210 USA
TEL: (614) 292-2568 FAX: (614) 292-2911
{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 analyses.

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., [3, 6, 12, 13, 20]), require control-flow information. Much research has addressed the problems of computing such analysis information for individual procedures (*intraprocedural*) (e.g., [5, 21]) and for interacting procedures (*interprocedural*) (e.g., [15]). 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 [11]. To be applicable to programs in languages, such as Java [7] 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 C++ that fails to recognize the flow of control among exception-handling statements cannot adequately measure the branch coverage of a test suite; a

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

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 a small study. In the study, we examined a number of non-trivial, real-life Java programs from a diverse group of applications, and obtained a count of programs, from each group, that contained either a `try` statement or a `throw` statement. Table 1 summarizes the results of the study; for each program group, it describes the application, and lists the number of programs examined, and the usage of exception-handling statements.

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

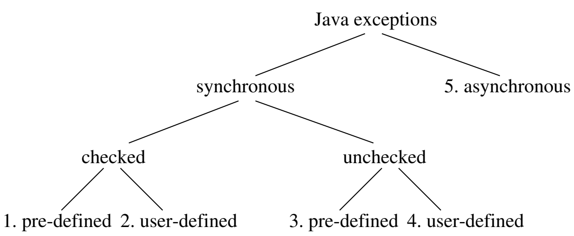
Program Group		Number of	Programs that Contain	
Name	Description	Programs	`try` Stmts	`throw` Stmts
jacorb	ORB implementation in Java	1062	271	229
javacup	LALR-parser generator for Java	34	5	17
jdk	Sun's JDK 1.1.5	1256	342	372
jlex	Lexical-analyzer generator for Java	1	1	1
swing	Sun's Swing API 1.0.2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

As the table illustrates, 23.3% and 24.5% of the examined programs contained `try` and `throw` statements, respectively; within a program group, 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. Several programs contained both `try` and `throw` statements, and over all program groups, there were 497 such programs. Therefore, there were 982 programs, which comprise 31.7% of all examined programs, that contained either a `try` statement or a `throw` statement. This study supports our belief that the use of exception-handling statements in real-life programs 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., [4]), 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 such representations, and discusses several applications that use the representations. We restrict our discussion to representations for Java programs that contain exception-handling constructs; representations for C++ programs can be constructed similarly.

The main contributions of this paper are:

- An analysis of the effects of exception occurrences and 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 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).




```
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: Exception types in Java, shown on the left, and the syntax of Java exception-handling constructs, shown on the right.

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, `EmptyStackException`. 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 on the left of Figure 1 shows the types of Java exceptions.

In Java programs, 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. All thrown exceptions in Java are instances of classes derived from `java.lang.Throwable`.

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 on the right 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 provide details of the corresponding graph-construction algorithms; we also 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 handled. Finally, we present some preliminary results about the frequency of occurrence of exception-handling statements.

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 [7], 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 `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 no exceptions.
Path 1-9 is taken if the `try` block completes normally, but the `finally` block raises an exception.

[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.

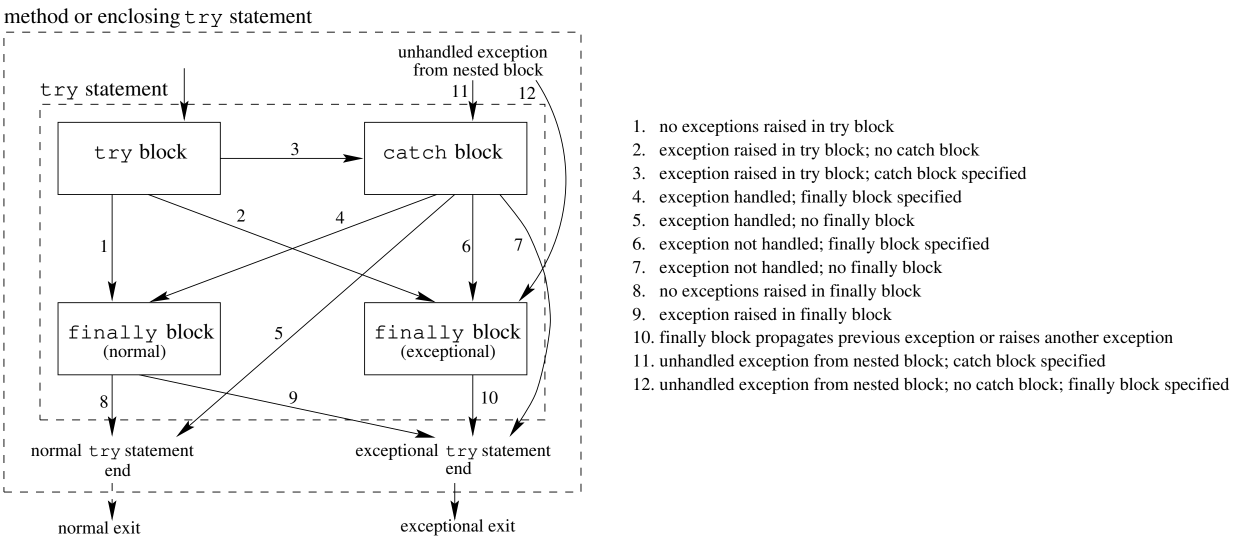


Figure 2: Control flow in Java exception-handling constructs.

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.[3]

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 its 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.[4]

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 then 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`); `PkConstraintException` 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 3.5, 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` 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] The compiled bytecodes for Java programs do not encode control flow for the handler-matching process. Instead, the Java compiler creates an exception table that has an entry for each handler mentioned in a method [7], and each such entry lists the address range of the code protected by the handler, the address of the handler, and the type of the handler. At run-time, when the Java Virtual Machine discovers that an exception is raised, it inspects the exception table, in the order that the handlers are listed, and transfers control to the first matching handler.

[4] 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 . . .
```

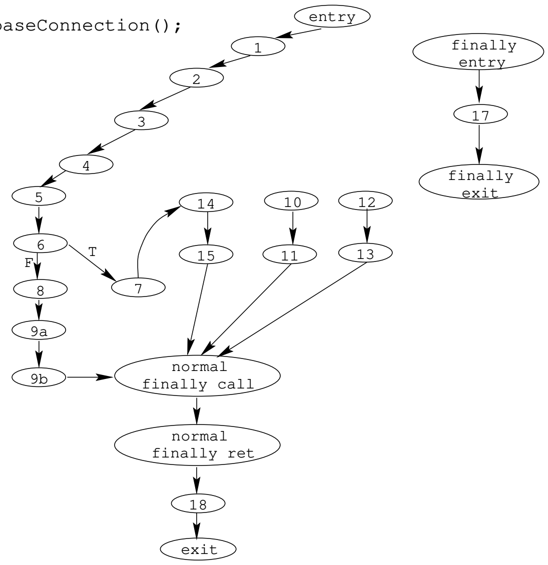


Figure 3: Partial Java program and its control-flow graph.

(3) as *normal*-finally-call nodes.[5]

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 that call `finally` blocks contained in that method (illustrated in the example considered in Section 3.2); these call nodes are reached along paths where unhandled exceptions are 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.[6] The number of exceptional-finally-call nodes created for a method M is bounded by the following product:

number of `finally` *blocks in M * 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

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

[6] 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.

may be raised at a point in a method where it is not enclosed by a `finally` block in that method.

We create exceptional-finally-call nodes in the CFG for a method in two passes. If a method M directly raises an exception, we create all required exceptional-finally-call nodes corresponding to `finally` blocks that enclose the exception-raise point while constructing the CFG for M. If, however, M indirectly raises an exception, we create required exceptional-finally-call nodes in a separate pass after the construction of individual CFGs for all methods; this is necessitated by a lack of information about exception types that can be raised at a call site (and that are propagated by the called method).

For condition (3), the CFG for a method contains several normal-finally-call nodes (none illustrated in the examples); 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; we create these normal-finally-call nodes during the CFG construction. 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 separate algorithm (that builds interprocedural representations) rather than the CFG-construction algorithm (Section 3.6).

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.

In the next subsection we illustrate the CFG-construction algorithm with a detailed example; following that, in Sections 3.3 and 3.4, we describe the CFG-construction algorithm.

3.2 Example to Illustrate CFG Construction

Our CFG-construction algorithm consists of two passes: the first pass constructs a CFG for each method in a program; the second pass iteratively determines the exception types that are indirectly raised and propagated by each method, and adds additional exceptional-finally-call and exceptional-exit nodes to the CFGs created during the first pass.

Consider the example shown in Figure 4. The figure shows five methods from a sample program, the

```
Method A

1. try {
2.   C()
   }
3. catch( Exception1 e1 ) {
   }
4. catch( Exception2 e2 ) {
   }
```

```
Method C

9.  try {
10.   try {
11.     if P1
12.       throw new Exception1()
      }
13.   finally {
      }
14.   E()
    }
15. finally {
    }
```

```
Method B

5. try {
6.   C()
   }
7. catch( Exception e ) {
   }
8. finally {
   }
```

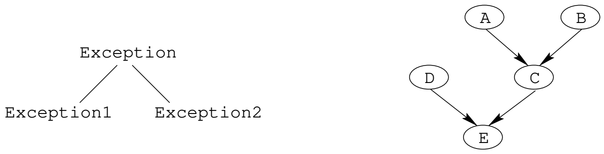
```
Method D

16. try {
17.   E()
    }
18. catch( Exception1 e1 ) {
    }
19. catch( Exception2 e2 ) {
    }
20. finally {
    }
```

```
Method E

21. try {
22.   if P2
23.     throw new Exception1()
24.   if P3
25.     throw new Exception2()
    }
26. finally {
    }
```

(a)



(b) (c)

Figure 4: Example to illustrate CFG-construction process: (a) sample program with five methods containing exception-handling constructs; (b) exception hierarchy; (c) call graph for sample program.

class hierarchy of the exception types raised in the methods, and the call graph[7] for the five methods. In the sample program, method E directly raises two exceptions: `Exception1` and `Exception2`; method C directly raises `Exception1`, through the `throw` statement in line 12, and indirectly raises `Exception1` and `Exception2`, through the call to method E in line 14. Our CFG-construction algorithm assumes that the exception-class hierarchy of the program being analyzed is known; this can easily be accommodated by including a preprocess pass before the CFG-construction algorithm that constructs the exception-class hierarchy for a program.

The first pass of our CFG-construction algorithm constructs CFGs for methods A, B, C, D, and E; these CFGs are illustrated in Figure 5. The nodes shown in the box are created in the second pass, after the initial CFG construction. In CFGs in Figure 5, nodes, such as 3, 4, 7, 18, and 19, that have no in-edges represent catch-assignment nodes. The presence of exceptional-exit nodes in a CFG indicates that the corresponding method propagates exceptions — the number of exceptional-exit nodes is the same as the number of distinct exception types propagated by the method. Thus, CFGs for methods A, B, and D do not have exceptional-exit nodes because those methods do not propagate any exceptions; CFGs for methods C and E contain two exceptional-exit nodes each because both C and E propagate two types of exceptions.

In each CFG, there is one pair of normal-finally-call/return nodes for each `finally` block in the corresponding method; this pair of nodes corresponds to the invocation of `finally` blocks under condition (1). There are two pairs of such nodes in the CFG for method C, and one each in the CFGs for methods B, D, and E. Although not shown in the figure, the CFGs for the `finally` blocks are also created in the

[7]A *call graph* for a program contains a node for every method in a program; there is an edge from the node representing a method M_1 to the node representing a method M_2 for every call site in M_1 that calls M_2.

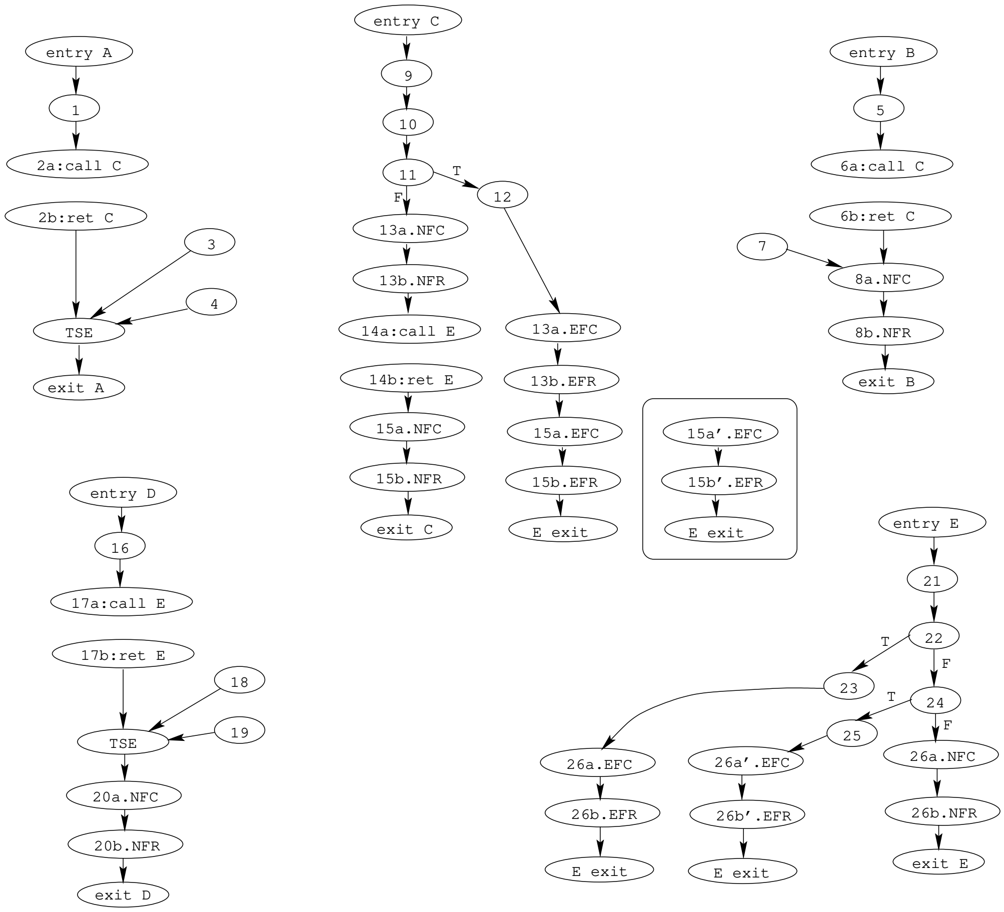


Figure 5: CFGs for methods from sample program; nodes labeled 'NFC' ('NFR') are normal-finally-call (normal-finally-return) nodes, nodes labeled 'EFC' ('EFR') are exceptional-finally-call (exceptional-finally-return) nodes, nodes labeled 'E Exit' are exceptional-exit nodes, and those labeled 'TSE' are try-statement-end nodes; nodes shown in the box are created during the second pass; the remainder of the nodes are created during the first pass of the CFG-construction algorithm. CFGs for the `finally` blocks are not shown in the figure.

---

first pass of the CFG-construction algorithm. The CFG for method E contains two exceptional-finally-call nodes because the `finally` block in line 26 encloses exception-raise points that raise two distinct types of exceptions. The CFG for C contains three exceptional-finally-call nodes, although C raises exceptions of two distinct types and contains two `finally` blocks, because the exception-raise point in line 14 is enclosed by only one of the `finally` blocks; this illustrates how the number of exceptional-finally-call nodes in a CFG can be less than the product of distinct exception types propagated by the corresponding method and the number of `finally` blocks in the corresponding method. The exceptional-finally-call nodes in the CFG for E are created during the first pass of the CFG-construction algorithm because E directly raises the two exceptions. In the CFG for C, however, exceptional-finally-call nodes for the exception raised directly in line 12 are created in the first pass, but those for the exception raised indirectly in line 14 are created during the second pass of the CFG-construction algorithm (after the exceptions propagated by E have been determined). The second pass of the algorithm creates additional exceptional-finally-call nodes (nodes 15a' and 15b' in Figure 5) for the `finally` block in line 15 corresponding to exception type `Exception2`; the second pass, however, does not create additional exceptional-finally-call nodes for the `finally` block in line

15 corresponding to exception type `Exception1` (that is also propagated by E) because such call nodes are already created during the first pass of the algorithm.

The individual CFGs shown in Figure 5 are connected using an algorithm that creates an interprocedural representation for a program (Section 3.6).

3.3 First Pass of CFG-construction Algorithm

The first pass of our CFG-construction algorithm, `ConstructCFG` (Figures 6 and 7), constructs a CFG for method M from M's abstract-syntax tree (AST), and gathers information about M that is required for construction of an interprocedural representation for the program that contains M.[8] The figure shows only those actions that are related to exception-handling constructs. We assume that the remainder of the CFG, unrelated to exception-handling constructs, is constructed in the usual way by a syntax-directed algorithm [2, 9]; thus, for example, at the point in the construction when the `try` statement is encountered, entry and exit nodes will have been created for the CFG.

`ConstructCFG` creates CFG nodes for `catch` blocks as illustrated in Figures 3 and 5. For each `finally` block that appears in a method, `ConstructCFG` creates a separate CFG with distinct entry and exit nodes, and inserts appropriate calls to the `finally` blocks in the CFG of the method that contains the `finally` blocks. `ConstructCFG` connects each throw-statement node directly to the catch-assignment node if the types of the thrown exception and the `catch` handler match. `ConstructCFG` saves two types of link information that are used by other algorithms:

1. The first type of link information saved by `ConstructCFG` pertains to call sites, and is used by the second pass of our CFG-construction algorithm (Section 3.4) and by the algorithm that builds interprocedural representations (Section 3.6). For each call site, `ConstructCFG` saves information about `try` statements that enclose the call site. The second pass of our CFG-construction algorithm uses such information to determine indirectly-raised and indirectly-propagated exception types for each method, and to create exceptional-finally-call and exceptional-exit nodes. The algorithm that constructs interprocedural representations uses such information to create interprocedural edges that emanate from exceptional-exit nodes.
2. The second type of link information saved by `ConstructCFG` pertains to `finally` blocks, and is used by the algorithm that builds interprocedural representations (Section 3.6). For each `finally` block, `ConstructCFG` saves the interprocedural destinations of transfer-of-control statements, such as `break`, `continue`, and `return`, that appear in the `finally` block. The interprocedural algorithm uses such information to create interprocedural edges that emanate from nodes representing transfer-of-control statements in the CFGs for `finally` blocks.

`ConstructCFG` uses several data structures:

[8] An alternative implementation for the first pass of our CFG-construction algorithm may build the CFG from Java bytecodes. It is argued that the primary benefit of such an approach is its simplicity over a syntax-directed construction [14]. Using bytecodes to build CFGs, however, raises additional issues such as mapping CFG nodes to source-code lines; depending on a particular implementation of a Java compiler, and the flags used when compiling the bytecodes, information that facilitates this mapping may not be available in the bytecodes [14]. The technique for building CFGs from bytecodes presented in [14] ignores exception-handling constructs. We believe that, using the information stored in the exception table constructed by a Java compiler, CFGs for Java programs containing exception-handling constructs can be built from the bytecodes.

- *linkInfo* saves the two types of link information described above. For the first type of link information, *linkInfo* has an entry, *callSites*, for each call site in a method; each such entry is comprised of a list of *enclTry*, that correspond to `try` statements that enclose a call site. Each enclosing-try-statement entry is comprised of information about `catch` handlers and `finally` block of a `try` statement: *catchList* stores catch-assignment nodes and types of the corresponding `catch` handlers, and *finallyBlock* stores a finally-entry node.

  For the second type of link information, *linkInfo* has an entry, *tocList*, for each `finally` block in a program; each entry contains a list of nodes representing transfer-of-control statements in that `finally` block, and the interprocedural destinations of those statements.
- *mre* is a pair — (*edge*, *label*) — that records the most recently created CFG edge.
- *tryRecord* stores information about a `try` statement that aids in the CFG construction; a *tryRecord* contains the following fields:
  - *unresEdges* stores edges that are to be connected to the first statement following a `try` statement.
  - *unresExceptions* keeps track of unhandled exceptions, that are processed in the context of the current `try` statement. *unresExceptions* stores lists of nodes grouped by exception types: for an exception of type T pending in the current scope, *unresExceptions*[T] stores a list of nodes such that each node is the final node along a path starting at an exception-raise point — a point where an exception of type T was raised; the CFG node that appears in such a list, therefore, may either be a throw-statement node or an exceptional-finally-return node. To illustrate with an example, consider the CFG for method E in Figure 5. After the `throw` statements in lines 23 and 25 are processed, *unresExceptions* contains entries for two types of exceptions, `Exception1` and `Exception2`; the list associated with `Exception1` contains node 23 and the list associated with `Exception2` contains node 25. After the `finally` block in line 26 is processed such that exceptional-finally-call/return nodes are created, *unresExceptions* is updated so that the list associated with `Exception1` contains node 26b, and the list associated with `Exception2` contains node 26b'.
  - *unresTransfers* stores nodes representing transfer-of-control statements that appear in the `try`, `catch`, or `finally` blocks of the current `try` statement.
  - *callList* stores a list of call-site entries; each call-site entry contains information that is to be stored in *linkInfo* for that call site; during the syntax-directed construction of CFGs, link information about call sites in built on *tryStack* and later saved in *linkInfo*.
  - *catchList* stores `catch` handlers that appear in a `try` statement; information about a handler includes catch-assignment node for the handler and the type of exception caught by the handler.
  - *finallyBlock* stores finally-entry node for the `finally` block of a `try` statement.
  - *enclLoop* stores the loop-header node for loop that encloses a `try` statement.
  - *finallyBlockTransfers* stores lists of *unresTransfers* — one *unresTransfers* list for each `finally` block that is enclosed by a `try` statement.
- *tryStack* keeps track of nested exception-handling constructs; each item in *tryStack* is a *tryRecord*.
- *throwTypes* stores the set of possible exception types at a `throw` statement.
- *propExceptions* stores information about exception types that are directly propagated by a method —

entries in *propExceptions* are similar to those in *unresExceptions*; *propExceptions* is used to accumulate all exceptions directly propagated by a method, and, at the end of a method, is used to create exceptional-exit nodes for each type of directly-propagated exception.

Apart from the above data structures, the overall algorithm uses other data structures that are used in a typical syntax-directed CFG-construction algorithm. For example, such an algorithm uses a control-flow stack to keep track of nested scopes in a program. We utilize such a stack to determine enclosing loops for exception-handling constructs, by searching the control-flow stack for nodes representing loop headers.

`ConstructCFG` uses several auxiliary functions:

- `Empty`, `Push`, `Pop`, and `Top` are normal stack operations.
- `Peek`(*stack*, *depth*) retrieves the element at the given depth in a stack, without modifying the state of the stack; `Top`(*stack*) is therefore equivalent to `Peek`(*stack*, 0).
- `CreateNode` creates a CFG node corresponding to a given AST node, or a CFG node of a specified type.
- `MakeEdge`(N_1, N_2) connects two given CFG nodes, N_1 and N_2.
- `AddNode`(N, *label*) links mre to the specified node N with the specified label on the edge, and updates mre to N's outbound edge (if N is a predicate node, mre is set to the true branch from N).
- `AddToList`(*elem*, *list*) adds an element to a list; *elem* may be either a CFG node or a list of nodes — in the latter case, an element of *list* is also a list of nodes.
- `BackPatch`(N, *list*) resolves edges from a list to a specified node N; *list* is made empty after each edge is resolved.
- `DetermineTypes` determines the set of possible types of exceptions that can be raised at a `throw` statement; we discuss type-inference in Section 3.7.

Figure 7 shows the auxiliary function `Pop` that is used by `ConstructCFG`. `Pop` performs several actions before removing a *tryRecord* from *tryStack*. In particular, `Pop` appends *unresExceptions* and *callList* from the top-most *tryRecord* to the corresponding lists of the next *tryRecord* (lines 90–91). If the next *tryRecord* is also enclosed by the same loop as the top-most *tryRecord*, `Pop` also appends *unresTransfers* and *finallyBlockTransfers* of the top-most *tryRecord* to the corresponding fields of the next *tryRecord* (lines 93–94).

`ConstructCFG` traverses the AST and performs actions based on the type of the AST node encountered. The algorithm uses *tryRecord* to store information pertaining to a `try` statement: on encountering a `try` statement, the algorithm pushes a *tryRecord* onto *tryStack*, and after processing that `try` statement, the algorithm pops off its *tryRecord*. Thus, the algorithm keeps track of nested `try` statements by manipulating the contents of *tryStack*. The algorithm uses a separate *tryRecord* to process a `finally` block. This enables the algorithm to create exceptional-exit nodes for the CFG of a `finally` block, and record link information for transfer-of-control statements in a `finally` block. For an AST node A, `ConstructCFG` performs actions based on the type of A.

If A is a transfer-of-control statement, such as `break`, `continue`, or `return`, (lines 3–8), the algorithm creates CFG node TOC, and adds it to the CFG (line 4). If this statement is within a `try` statement, which is the case if *tryStack* is not empty and the *enclLoop* field of the current *tryRecord* is not null, the algorithm adds the outbound edge of TOC to the *unresTransfers* list (line 6). If not, the node is processed normally as in [2] (line 7).

```
algorithm   ConstructCFG
input       AST: abstract syntax tree for try statement
output      CFG: control flow graph for P with try statement
globals     linkInfo, throwTypes
declare     mre, tryStack, propExceptions

begin ConstructCFG
  foreach node A in AST do
        case A is
        break/continue/return statement:
              TOC = CreateNode(A); AddNode(TOC, –)
              if ~ Empty(tryStack) and Top(tryStack).enclLoop ≠ null then
                    AddToList(TOC, Top(tryStack).unresTransfers)
              else ... endif // normal processing of break/continue/return statement
              mre = null
        throw statement:
              TS = CreateNode(A); AddNode(TS, –); DetermineTypes(TS, CFG);
              foreach type T in throwTypes do
                    if ~ Empty(tryStack) then
                          AddToList(TS, Top(tryStack).unresExceptions[T])
                    else
                          AddToList(TS, propExceptions[T])
                    endif
              endfor
              mre = null
        try statement:
              TSB = CreateNode(A); AddNode(TSB, –); Push(tryStack, tryRecord)
              if a loop-header node appears in control-flow stack then
                    Top(tryStack).enclLoop = loop-header node
              endif
        try block end:
              AddToList(mre, Top(tryStack).unresEdges); mre = null
        try statement end:
              if Peek(tryStack, 1).enclLoop ≠ Top(tryStack).enclLoop then
                    process nodes in Top(tryStack).unresTransfers as transfer-of-control statements
                    foreach list L in Top(tryStack).finallyBlockTransfers do
                          resolve destinations for nodes in L; save link information for finally block in linkInfo
                    endfor
              endif
              TSE = CreateNode(A); BackPatch(TSE, Top(tryStack).unresEdges)
              foreach call site C in Top(tryRecord).callList do
                    add new enclTry entry to Top(tryRecord).callList[C]
                    Top(tryRecord).callList[C].enclTry.catchList = Top(tryRecord).catchList
                    Top(tryRecord).callList[C].enclTry.finallyBlock = Top(tryRecord).finallyBlock
              endfor
              if Peek(tryStack, 1) = null then
                    foreach type T in Top(tryStack).unresExceptions do
                          append nodes in Top(tryStack).unresExceptions[T] to propExceptions
                    endfor
                    save Top(tryStack).callList in linkInfo
              endif
              Pop(tryStack); mre = (TSE, –)
        catch clause:
              CA = CreateNode(catch assignmment); AddToList(CA, Top(tryStack).catchList)
              foreach exception type T in Top(tryStack).unresExceptions do
                    if T is subtype of type of catch handler then
                          BackPatch(CA, Top(tryStack).unresExceptions[T])
                    endif
              endfor
              mre = (CA, –)
        catch clause end:
              AddToList(mre, Top(tryStack).unresEdges); mre = null
```

Figure 6: The first pass of CFG-construction algorithm for a *try statement* in Java.

```
56.            finally block:
57.                 FE = CreateNode(finally entry)
58.                 foreach node N in Top(tryStack).unresTransfers do
59.                      NFC = CreateNode(normal finally call); BackPatch(NFC, N)
60.                      NFR = CreateNode(normal finally return); MakeEdge(NFC, NFR)
61.                      AddToList(NFR, Top(tryStack).unresTransfers)
62.                 endfor
63.                 foreach exception type T in Top(tryStack).unresExceptions do
64.                      EFC = CreateNode(exceptional finally call); BackPatch(EFC, Top(tryStack).unresExceptions[T])
65.                      EFR = CreateNode(exceptional finally return); MakeEdge(EFC, EFR)
66.                      AddToList(EFR, Top(tryStack).unresExceptions[T])
67.                 endfor
68.                 NFC = CreateNode(normal finally call); BackPatch(NFC, Top(tryStack).unresEdges)
69.                 NFR = CreateNode(normal finally return); MakeEdge(NFC, NFR)
70.                 AddToList(NFR, Top(tryStack).unresEdges)
71.                 AddToList(NFC, Peek(tryStack, 1).callList)
72.                 Push(tryStack); Top(tryStack).finallyBlock = FE
73.                 mre = (FE, -)
74.            finally block end:
75.                 FX = CreateNode(finally exit); AddNode(FX, -)
76.                 foreach type T in Top(tryStack).unresExceptions do
77.                      EX = CreateNode(exceptional exit); BackPatch(EX, Top(tryStack).unresExceptions[T]
78.                 endfor
79.                 AddToList(Top(tryStack).unresTransfers, Peek(tryStack, 1).finallyBlockTransfers)
80.                 Peek(tryStack, 1).finallyBlock = Top(tryStack).finallyBlock
81.                 Pop(tryStack); mre = null
82.            method end:
83.                 foreach type T in propExceptions do
84.                      EX = CreateNode(exceptional exit); BackPatch(EX, propExceptions[T]
85.                 endfor
86.                 connect mre and unresolved edges (on control-flow stack) to exit node
87.            endcase
88.       endfor
     end ConstructCFG

     function   Pop
     input      tryStack
     begin Pop
89.       nextRec = Peek(tryStack, 1)
90.       append Top(tryStack).unresExceptions to nextRec.unresExceptions
91.       append Top(tryStack).callList to nextRec.callList
92.       if Top(tryStack).enclLoop = nextRec.enclLoop then
93.            append Top(tryStack).unresTransfers to nextRec.unresTransfers
94.            append Top(tryStack).finallyBlockTransfers to nextRec.finallyBlockTransfers
95.       endif
96.       remove top element from tryStack
     end Pop
```

Figure 7: Algorithm for first pass of CFG-construction continued; actions performed by the auxiliary function `Pop`.

If A is a `throw` statement (lines 9–18), the algorithm adds a throw-statement node TS to the CFG and invokes `DetermineTypes` to determine the possible types of the exception raised at that statement (line 10). For each type of exception T that can be raised at that point, the algorithm adds TS to either the *unresExceptions* list for type T if *tryStack* is not empty (line 13), or to the *propExceptions* list for type T for the current method if *tryStack* is empty (line 15). The latter indicates that the corresponding `throw` statement appears outside of a `try` statement and therefore, results in exceptions types being propagated by the current method. Because the algorithm potentially adds the throw-statement nodes to multiple lists, a throw-statement node may have multiple out-edges — one for to each type of exception that can be raised at that `throw` statement.

If A is a `try` statement (lines 19–23), the algorithm adds a try-statement node to the CFG, and pushes a new *tryRecord* onto *tryStack* (line 20). Next, the algorithm inspects the control-flow stack to determine if the `try` statement is enclosed by a loop, and sets the *enclLoop* field of the new *tryRecord* (lines 21–23).

If `ConstructCFG` encounters the end of a `try` block (lines 24–25), it adds *mre* to *unresEdges* list of the current `tryRecord`.

On reaching the end of a `try` statement (lines 26–45), `ConstructCFG` performs several actions. First, the algorithm determines if the `try` statement that encloses the current `try` statement is enclosed by the same loop as the current `try` statement. If this is not the case, the algorithm resolves nodes that appear in *unresTransfers* by connecting them to the destinations of the corresponding transfer-of-control statements (line 28). The algorithm also saves link information for each entry in *finallyBlockTransfers* (lines 29–31); because each entry in *finallyBlockTransfers* corresponds to one `finally` block and is a list of nodes, the algorithm resolves the destination of each of these nodes, and saves link information for that `finally` block. Second, the algorithm adds a try-statement-end node *TSE* to the CFG and resolves all nodes in *unresEdges* to *TSE* (line 33). Third, the algorithm accumulates call site information to be saved in *linkInfo*: this involves adding a new *enclTry* entry for each call site in *callList* (line 35), and setting the fields in that *enclTry* (lines 36–37). Next, the algorithm determines if the current `try` statement is enclosed by another `try` statement (line 39). If it is not, the algorithm adds lists (for different exception types) in *unresExceptions* to *propExceptions*; this step builds the exception types directly propagated by the current method (lines 40–42). The algorithm also saves *callList* information in *linkInfo* (line 43). Finally, the algorithm pops the `tryRecord` for the current `try` statement (line 45).

If A is a `catch` clause (lines 46–53), the algorithm adds a catch-assignment node to the CFG (line 47), and resolves all nodes from *unresExceptions* whose types match the type of the `catch` clause (lines 48–52).

On reaching the end of a `catch` clause (lines 54–55), the algorithm adds *mre* to *unresEdges* for the current *tryRecord*.

The remainder of the cases of `ConstructCFG` are continued in Figure 7. If A is a `finally` block (lines 56–73), the algorithm creates a finally-entry node (line 57). Next, the algorithm creates normal-finally-call/return nodes for condition (3). It creates one normal-finally-call node *NFC* for each node that appears in *unresTransfers*, and backpatches nodes from *unresTransfers* to *NFC* (line 59). The algorithm creates a normal-finally-return node *NFR*, connects *NFC* to *NFR*, and adds *NFR* to *unresTransfers* (lines 60–61). The algorithm then creates exceptional-finally-call nodes for condition (2) in a similar manner. It creates one exceptional-finally-call node *EFC* for each type of exception that appears in *unresExceptions* and backpatches all nodes from the list in *unresExceptions* for that type to *EFC* (line 64). The algorithm creates an exceptional-finally-return node *EFR*, connects *EFC* to *EFR*, and adds each *EFR* node to *unresExceptions* list for that exception type (lines 65–66). Next, the algorithm creates a normal-finally-call node *NFC* for condition (1), resolves nodes in *unresEdges* to *NFC*, creates a normal-finally-return node *NFR*, connects *NFC* to *NFR*, and adds *NFR* to *unresEdges* (lines 68–70). The algorithm also adds *NFC* to *callList* of the *tryRecord* for the enclosing `try` statement (line 71), which ensures that the second pass of our CFG-construction algorithm works correctly — if a `finally` block propagates an exception type, that exception type must be propagated back to the method that contains call sites to that `finally` block. Although multiple call nodes (to a `finally` block) may be created in the CFG for the method containing that `finally` block, adding only one of those call nodes to *callList* is sufficient to guarantee that the iterative propagation in the second pass of our CFG-construction algorithm works correctly; for this reason, we add just the *NFC* for condition (1) to *callList*.[9]

Finally, the algorithm pushes a new *tryRecord* onto *tryStack* and sets its *finallyBlock* field (line 72). The use of a separate *tryRecord* for a `finally` block enables `ConstructCFG` to keep track of unhandled exceptions and transfer-of-control statements from a `finally` block, and distinguish them from those that belong to the `try` or `catch` blocks of the current `try` statement. Because the algorithm is able to make this distinction, it correctly processes unhandled exceptions and transfer-of-control statements at the end of a `finally` block.

On reaching the end of a `finally` block (lines 74–81), the algorithm creates a finally-exit node and adds it to the CFG (line 75). Any unhandled exceptions (on *unresExceptions*) indicate that the `finally` block propagates exceptions. Therefore, the algorithm creates an exceptional-exit node *EX* for each type of exception that appears in *unresExceptions* and resolves nodes from the list for that type in *unresExceptions* to *EX* (line 77). Nodes that appear in *unresTransfers* list of the current *tryRecord* represent transfer-of-control statements that appear within a `finally` block. The algorithm adds *unresTransfers* as an element to *finallyBlockTransfers* for the next *tryRecord* (line 79). Finally, the algorithm sets *finallyBlock* field for the enclosing *tryRecord* (because this is required when saving *callList* at the end of a `try` statement), and pops the *tryRecord* for the `finally` block (lines 80–81).

On reaching the end of a method (lines 82–86), the algorithm inspects exception types that appear in *propExceptions* — these are the exception types that are directly propagated by the current method. Therefore, the algorithm creates an exceptional-exit node *EX* for each such type T, and resolves nodes that appear in the list associated with T in *propExceptions* to *EX* (line 84). Finally, the algorithm adds the exit node to the CFG and connects unresolved edges to it (line 86).

3.4 Second Pass of CFG-construction Algorithm

The second pass of our CFG-construction algorithm, `PropagateExceptionTypes` (Figure 8) determines those exception types that are indirectly raised or propagated by each method; information about indirectly-raised and indirectly-propagated exception types enables `PropagateExceptionTypes` to create exceptional-finally-call and exceptional-exit nodes for such exception types.

`PropagateExceptionTypes` uses the link information about call sites saved by `ConstructCFG`. `PropagateExceptionTypes` inputs the CFG for each method in a program, and using the link information for that method, adds several nodes to the CFG for that method. `PropagateExceptionTypes` iteratively propagates exception types from a called method to its callers until the fixed-point for the set of exception types propagated by each method is reached. When the algorithm discovers a new exception type that is indirectly raised by a method, it creates exceptional-finally-call nodes that may be required. Similarly, each time the algorithm discovers a new exception type that is indirectly propagated by a method, the algorithm creates an exceptional-finally-exit node for that exception type, if required. The algorithm uses a variable *change* to determine if a method's set of propagated exceptions changes from a previous iteration; if *change* evaluates to true, the algorithm processes each method one more time.

The algorithm inspects each call site C in a method (line 5), and iterates through the exceptional-exit

[9]The reason for adding the finally-call node for condition (1) but not one for conditions (2) or (3) is that a finally-call node for condition (1) is always created when a `finally` block encountered, whereas as call nodes for the other two conditions may not be created depending on whether a `finally` block encloses an exception-raise point, or whether a `finally` block contains a transfer-of-control statement, respectively.

```
algorithm   PropagateExceptionTypes
input       CFG: CFG for each method
output      CFG: CFG with additional exceptional-finally-call/return and exceptional-exit nodes
globals     linkInfo: link information
declare     mre, change, exceptionHandled

    begin PropagateExceptionTypes
1.     change = true
2.     while change = true do
3.         change = false
4.         foreach method M do
5.             foreach call site C in M (that calls method N) do
6.                 foreach exceptional-exit node EX in CFG for N do
7.                     mre = null; exceptionHandled = false
8.                     T = exception type associated with EX
9.                     foreach enclosing-try-statement entry TSentry in linkInfo[M].CallSites[C].enclTry do
10.                        if T is caught by a handler in TSentry.catchList then
11.                            exceptionHandled = true; break
12.                        else
13.                            if TSentry.finallyBlock ≠ null then
14.                                EFC = exceptional-finally-call node associated with
                                         TSentry.finallyBlock and T
15.                                if EFC ≠ null then
16.                                    exceptionHandled = true; break
17.                                else
18.                                    EFC = CreateNode(exceptional finally call); AddNode(EFC, –)
19.                                    EFR = CreateNode(exceptional finally return); AddNode(EFR, –)
20.                                endif
21.                            endif
22.                        endif
23.                    endfor
24.                    if ~exceptionHandled then
25.                        EX = exceptional-exit node of M for type T
26.                        if EX = null then
27.                            EX = CreateNode(exceptional exit); AddNode(EX, –)
28.                            change = true
29.                        else MakeEdge(mre, EX)
30.                        endif
31.                    endif
32.                endfor
33.            endfor
34.        endfor
35.    endwhile

    end PropagateExceptionTypes
```

Figure 8: Algorithm for the second pass of our CFG-construction algorithm.

nodes that appear in the CFG for the method called at C (line 6). Each exceptional-exit node in the CFG for the called method corresponds to an exception type that is propagated by the called method. Therefore, the algorithm next determines if an exception type T, propagated by the called method, is caught by a handler that encloses the call site C (lines 9–23). To do this, the algorithm examines each enclosing-try-statement entry *TSentry* in the link information for C (line 9). If a handler listed in *TSentry* catches T, the algorithm sets a boolean variable, *exceptionHandled*, to true, and does not examine further enclosing-try-statement entries for C (lines 10–11). If none of the handlers listed in *TSentry* catch T, the algorithm determines if the `try` statement corresponding to *TSentry* has a `finally` block (line 13), and provided there is such a `finally` block, the algorithm next tries to find an exceptional-finally-call node corresponding to that `finally` block and T (line 14). If such an exceptional-finally-call node exists, the algorithm again sets *exceptionHandled* to true, and does not examine further enclosing-try-statement entries for C (line 16). If however, the algorithm fails to find such an exceptional-finally-call node, it creates an exceptional-finally-call node *EFC* and adds it to the CFG for M (line 18); the algorithm also creates an exceptional-finally-return node corresponding to *EFC* and adds it to the CFG (line 19).

After examining all enclosing-try-statement entries for C, if the algorithm finds that *exceptionHandled* is set to false, an exceptional-exit node corresponding to T may need to be created. The algorithm determines if such an exceptional-exit node exists (line 25); if it does, the algorithm connects *mre* to that exceptional-exit node (line 29). If an exceptional-exit node corresponding to T does not exist, the algorithm has discovered a new exception type that is indirectly propagated by M; therefore, the algorithm creates an exceptional-exit node, adds it to the CFG, and sets *change* to true (lines 27–28).

As with any iterative algorithm, `PropagateExceptionTypes` could alternatively be implemented based on a topological ordering of the program's call graph; such an implementation would process each method that is not involved in recursion only once, and would iteratively process methods involved in recursion until a fixed point is reached.

3.5 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 [16].

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.

Figure 9 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 in A: (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 A's `try` block has a corresponding `finally` block, control flows to that `finally` block (in the ICFG this is represented as an exceptional call to that `finally` block); (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, exceptional-finally-call, or 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

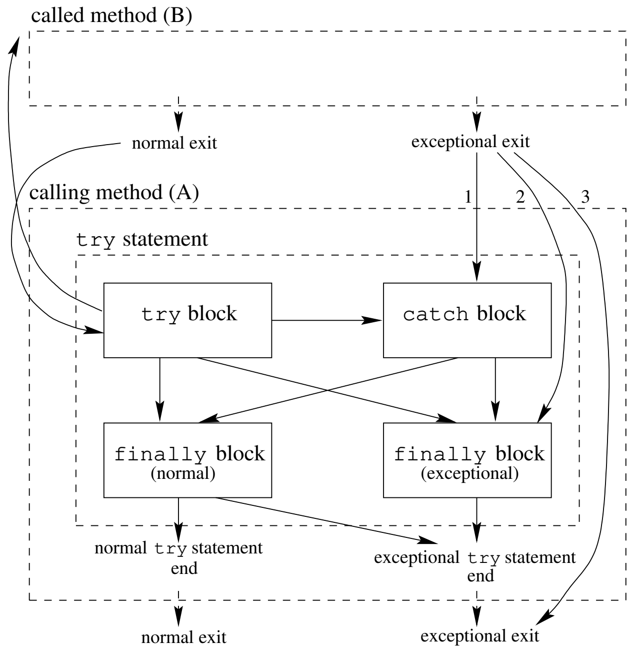


Figure 9: Interprocedural control flow in exception-handling constructs.

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.

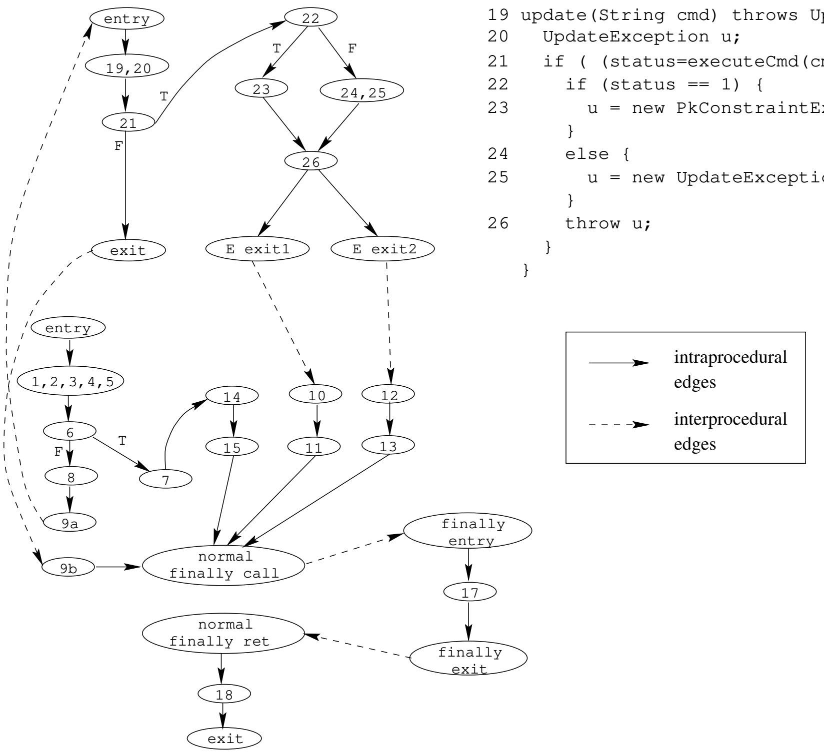
Figure 10 shows an implementation of the `update()` method for the partial Java program. This method accepts an SQL command as a parameter, and executes it in the database. If the command fails, `update()` throws an exception based on the cause of the failure. The ICFG for the partial Java program is also shown in Figure 10. 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 `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()`.

3.6 Algorithm for Construction of ICFG

The ICFG-construction algorithm, `ConstructICFG`, shown in Figure 11, connects the CFGs created by the CFG-construction algorithm using four kinds of edges: call edges, return edges, interprocedural-exception edges, and interprocedural-transfer edges.

`ConstructICFG` uses the two types of link information saved in *linkInfo* to create interprocedural edges: `ConstructICFG` uses the link information about enclosing `try` statements at call sites to create interprocedural-exception edges; similarly, `ConstructICFG` uses link information about transfer-of-control statements in `finally` blocks to create interprocedural-transfer edges.

`ConstructICFG` inputs the CFG and link information for each method, and processes all call sites in each method. For a call site C, the algorithm creates call and return edges (line 5). If the method called at C




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

propagates exceptions, the algorithm creates an interprocedural-exception edge from each exceptional-exit node in the CFG of the called method to an appropriate node in the CFG of the method that contains C (lines 6–25). To determine the target of an interprocedural-exception edge for an exception type T, `ConstructICFG` examines the information about `try` statements that enclose C (line 8). If a `catch` handler listed in an enclosing-try-statement entry *TSentry* catches T, the algorithm creates an interprocedural-exception edge to a catch-assignment node (lines 10–11). If no handler from *TSentry* catches T, but the `try` statement corresponding to *TSentry* has a `finally` block, the algorithm creates an interprocedural-exception edge to an exceptional-finally-call node (lines 15–16). Finally, if neither of the previous interprocedural-exception edges are created, the algorithm creates an interprocedural-exception edge to an exceptional-exit node (lines 22–23).

Next, `ConstructICFG` creates interprocedural-transfer edges that emanate from nodes representing transfer-of-control statements in `finally` blocks; the algorithm uses link information to determine the target of each transfer-of-control statement. If method M represents a `finally` block, the algorithm creates an interprocedural-transfer edge from each node representing a transfer-of-control statement in M to a node that is the destination of that transfer-of-control statement (lines 27–31).

The actions of `ConstructICFG` to create interprocedural-exception edges are similar to the actions that `PropagateExceptionTypes` performs to determine indirectly-raised and indirectly-propagated exception types. Therefore, the actions to create interprocedural edges can be incorporated into `PropagateExceptionTypes`, thereby obviating the need for a separate interprocedural algorithm. However, we choose to separate the two algorithms because the CFGs generated by `PropagateExceptionTypes` (the second pass of the CFG-construction algorithm) may then be used for other purposes, such as constructing augmented control-flow

```
algorithm              ConstructICFG
input                  CFG: CFG for each method M in program P
                       linkInfo: link information for each M saved by CFG-construction algorithm
output                 ICFG: interprocedural control-flow graph for P
auxiliary functions    RemoveEdge(N1, N2): remove edge between N1 and N2

begin ConstructICFG
  foreach method M in P do
      foreach call site C (with call node CN and return node RN) in the CFG for M do
          N = method called at C
          E_N = entry node of N; X_N = exit node of N
          MakeEdge(CN, E_N); MakeEdge(X_N, RN); RemoveEdge(CN, RN)
          foreach exceptional-exit node EX_N in CFG for N do
              T = exception type associated with EX_N
              foreach enclosing-try-statement entry TSentry in linkInfo[M].CallSites[C].enclTry do
                  if T is caught by handler H in TSentry.catchList then
                      CA = catch-assignment node for H
                      MakeEdge(EX_N, CA)
                      exceptionHandled = true; break
                  else
                      if TSentry.finallyBlock ≠ null then
                          EFC = exceptional-finally-call node associated with TSentry.finallyBlock and T
                          MakeEdge(EX_N, EFC)
                          exceptionHandled = true; break
                      endif
                  endif
              endfor
              if ~ exceptionHandled then
                  EX_M = exceptional-exit node of M for type T
                  MakeEdge(EX_N, EX_M)
              endif
          endfor
      endfor
      if M represents a finally block then
          foreach node TOC in linkInfo[M].tocList do
              MakeEdge(TOC, linkInfo[M].tocList[TOC].dest)
          endfor
      endif
  endfor
end ConstructICFG
```

Figure 11: Algorithm for construction of ICFG for a Java program with exception-handling constructs.

graphs [11].

Figure 12 shows the ICFG for the sample program from Figure 4.

3.7 Precision of Control-flow Representations

The preceding discussions on generating 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 a large 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 [17, 18] 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 [4] uses

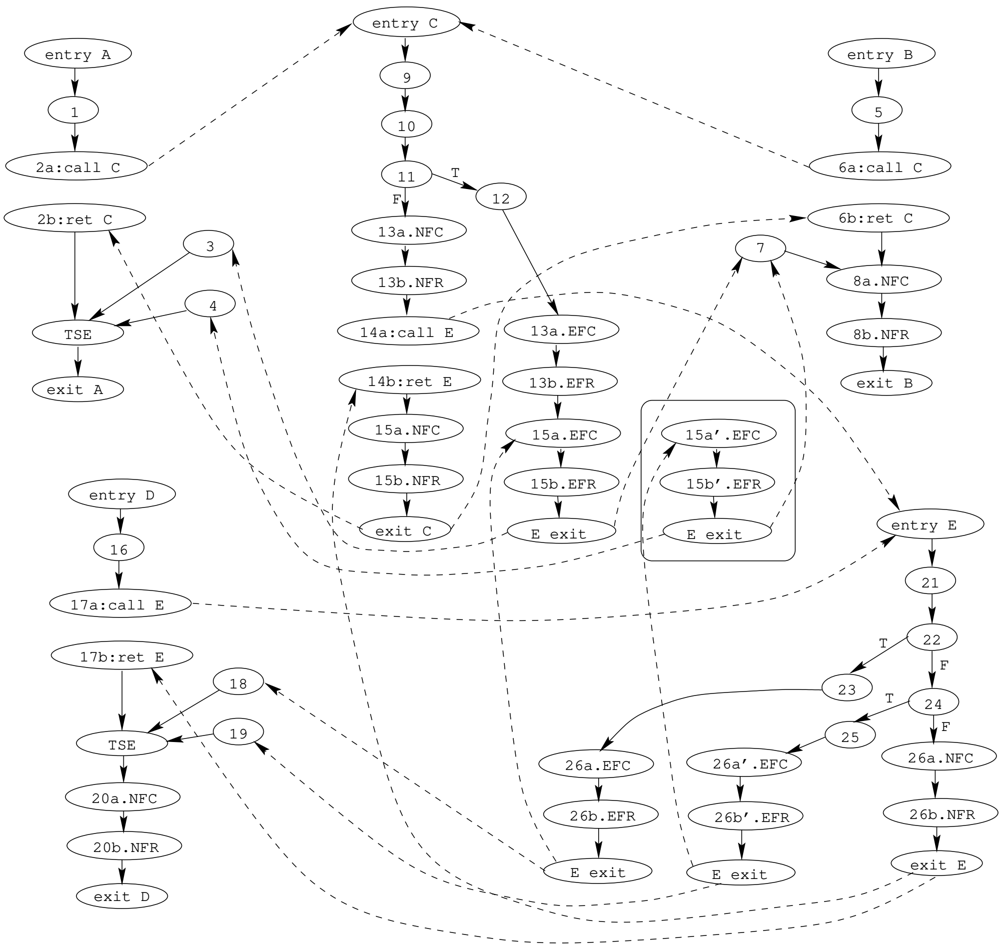


Figure 12: ICFG for methods from sample program; intraprocedural edges are shown in solid arrows; interprocedural edges are shown in dashed arrows.

---

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, `ConstructCFG` invokes the routine `DetermineTypes`, which computes types for exceptions raised at the corresponding `throw` statement. `DetermineTypes` performs reverse data-flow analysis starting from the throw-statement node in the CFG; on reaching a call site or the entry node of the CFG, `DetermineTypes` 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.

`DetermineTypes`, shown in Figure 13, is invoked (from line 10 of `ConstructCFG`) with a throw-statement node and a partially-constructed CFG as input parameters. `DetermineTypes` first examines the expression mentioned in the `throw` statement. If the `throw` statement mentions a new-instance expression, as does the `throw` statement in line 7 of partial Java program in Figure 3, type-inferencing becomes trivial — the only possible type for the raised exception is the type mentioned in the new-instance expression, and

```
     function   DetermineTypes
     input      N: a throw-statement node from the CFG for a method
                CFG: partially-constructed CFG that contains N
     globals    throwTypes: set of types that constitutes the solution for the given throw statement
     begin DetermineTypes
1.     expr = expression mentioned in throw statement
2.     if expr is new-instance expression then
3.         throwTypes = type mentioned in expr
4.     else if expr is method-call expression then
5.         retType = return type of method called in expr
6.         add all subtypes of retType to throwTypes
7.     else GetVarTypes(expr, N, CFG) endif
     end DetermineTypes

     function   GetVarTypes
     input      relVar: relevant variable whose type is to be inferred
                N: CFG node
                CFG: partially-constructed CFG that contains N
     globals    throwTypes: set of types that constitutes the solution for the given throw statement
     begin GetVarTypes
8.     foreach CFG-predecessor P of N do
9.         if P is an assignment node (lhs-var = rhs-expr) where lhs-var is relvar then
10.            if P is a catch-assignment node then
11.                type = declared type of corresponding catch handler
12.                add all subtypes of type to throwTypes; continue
13.            endif
14.            case rhs-expr is
15.            new-instance expression:
16.                throwTypes = type mentioned in rhs-expr
17.            method-call expression:
18.                type = return type of method called in rhs-expr
19.                add all subtypes of type to throwTypes
20.            variable:
21.                GetVarTypes(rhs-expr, P, CFG)
22.            endcase
23.        else if P is entry node of CFG then
24.            type = declared type of relVar
25.            add all subtypes of type to throwTypes
26.        else GetVarTypes(relVar, P, CFG) endif
27.    endfor
     end GetVarTypes
```

Figure 13: Type-inference algorithm used by `ConstructCFG` to determine types of exceptions.

`DetermineTypes` returns that type (line 3). If the `throw` statement mentions a method-call expression, `DetermineTypes` conservatively estimates all subtypes of the return type of the called method as the possible types of a raised exception (lines 5–6). If neither of the above is true, the `throw` statement must mention a variable, and `DetermineTypes` invokes `GetVarTypes` to determine the type of that variable (line 7).

`GetVarTypes` performs a reverse data-flow analysis (similar to slicing) [8] starting from a given CFG node and a given relevant variable *relVar*. Intuitively, `GetVarTypes` propagates *relVar* backwards in the CFG along all paths until it is killed: *relVar* is killed at an assignment node where it appears as the left-hand-side variable; if the right-hand side of such an assignment node mentions either a new-instance expression or a method-call expression, no new relevant variable is generated and propagation along that path terminates; if, however, the right-hand side of such an assignment node mentions a variable, a new relevant variable — the variable on the right-hand side — is generated and propagation continues along that path.

The algorithm recursively iterates through the CFG-predecessors of the given CFG node. If a predecessor P is an assignment node where *relVar* is assigned a value, the algorithm next determines whether the assignment node represents a catch-assignment node (line 10) — such as when a `catch` handler re-throws the exception that it catches — and returns all subtypes of the handler type as the solution (lines 11–12). If P does not represent a catch-assignment node, `GetVarTypes` takes actions based on the expression assigned to *relVar* (lines 14–22): the expression that is assigned to *relVar* may either be a new-instance expression, a variable, or a method-call expression. If the right-hand-side expression *rhs-expr* is a new-instance expression, `GetVarTypes` returns the type of that expression (line 16). If *rhs-expr* is a method-call expression, `GetVarTypes` return all subtypes of the return type of the called method (lines 18–19). If *rhs-expr* is a variable, `GetVarTypes` invokes itself recursively, with *rhs-expr* as the relevant variable (line 21). Finally, it may be the case that the relevant variable is a global variable, or is passed in as a parameter to the method containing the `throw` statement so that it is assigned a value outside the method being analyzed; lines 24–25 handle this case by adding all subtypes of the declared type of *relVar* to the solution.

Our type-inference algorithm is, thus, flow sensitive because it performs precise analysis in the method containing a `throw` statement, but is context insensitive because it makes conservative assumptions when it reaches a call site, a `catch` handler, or the method entry.

Some initial data that we gathered leads us to believe that in practice, `GetVarTypes` may not be invoked at all for a majority of `throw` statements in a program. We counted the total number of `throw` statements in programs from the groups listed in Table 1: out of a total of 2752 `throw` statements that appeared in the subject programs, 2618 mentioned a new-instance expression; of the remaining 134 `throw` statements, 114 mentioned a variable, whereas 20 mentioned a method call. Therefore, for 95.1% of the `throw` statements from our subjects, `DetermineTypes` would return the value computed in line 3, without invoking `GetVarTypes`.

Based on further experimentation, which would reveal the level of imprecision in the types inferred by `DetermineTypes` (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.

3.8 Handling Pre-defined Exceptions

The preceding discussions apply to programs where 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 3 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 need to 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.

The important question is how to determine those messages that can raise an exception. A simple, and conservative approach might be to 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. This information can help us decide whether to include a predicate node after a message send. 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` 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 may be raised by a message send, when one Java thread sends a `stop()` message to another Java thread. However, in such cases, the program point at which the message is sent and the point where the exception is raised belong in different threads; therefore, identifying the `stop()` message does not identify the point where the exception is raised. Because 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, determining the program point where such an exception may be raised does not serve any useful purpose.

3.9 Preliminary Results

We gathered preliminary data about the frequency with which exception-handling statements occur in Java programs. Table 1 shows how often our subjects contain `try` and `throw` statements, but it does not indicate how frequently such statements occur within a specific Java program. Thus, for example, Table 1 shows that 722 Java programs from our subjects contain `try` statements, but it does not tell us how many `try` statements appear in those 722 programs; gathering such data was the purpose of our second study.

Figure 14 shows a scatter plot of the frequency with which `try` statements occurred in our subjects. The plot contains 722 data points, one for each program that contained at least one `try` statement; each point shows the percentage of statements in a program that were `try` statements. The horizontal axis numbers the programs in the order in which the program groups are listed in Table 1; thus, for example, the first 271 data points correspond to programs from `jacorb`. The plot shows that for most of the programs, the number of `try` statements ranges from 0.5% to 8.0% of the total number of statements. For `jacorb`, 55 data points were clustered at the value 6.67%.

Figure 15 shows a similar scatter plot for the frequency of `throw` statements. The plot in Figure 15 contains 757 data points for the 757 programs that contained `throw` statements. The data points for `throw` statements are more widely dispersed than those for `try` statements; for most programs, the percentages of `throw` statements fall between 0.25% and 16.0% of the total statements. For `jacorb`, several data points were again clustered at one value — there were 57 data points with the value 3.33%.

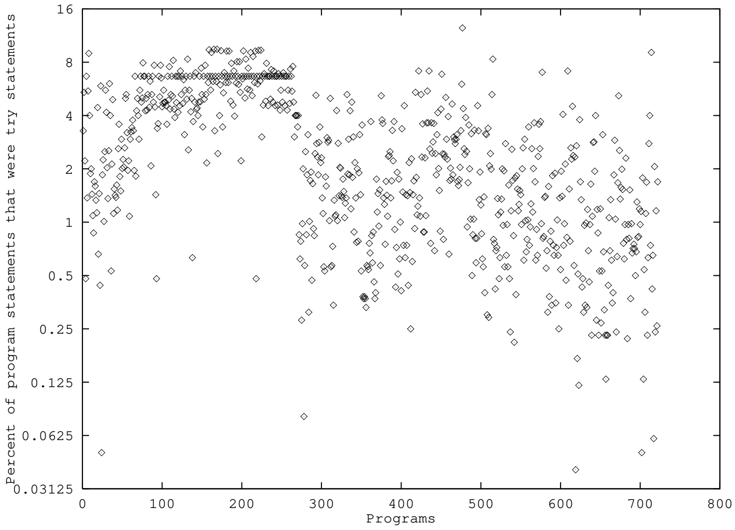


Figure 14: Frequency of occurrence of `try` statements in programs that contained `try` statements.

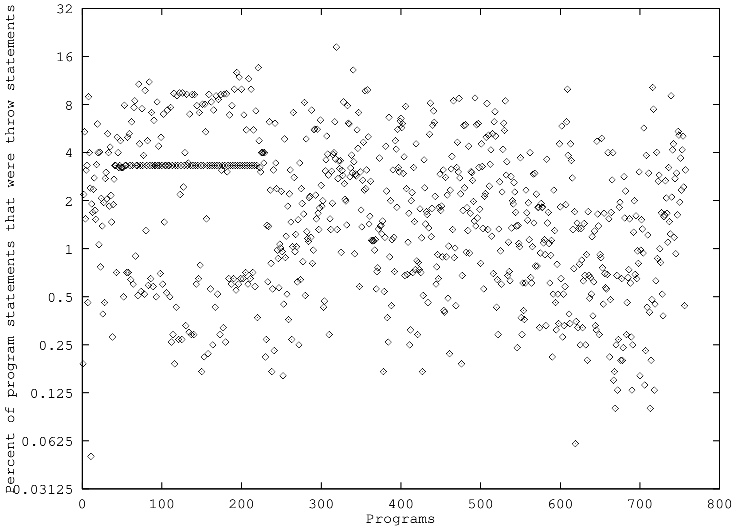


Figure 15: Frequency of occurrence of `throw` statements in programs that contained `throw` statements.

Figure 16 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 data points from Figure 14 in various ranges — for example, approximately 13% of the data points in Figure 14 fall in the range of 6% to 8% of the statements. Similarly, the segmented bar for `throw` statements illustrates the distribution of data points in Figure 15. The comparison of the two segmented bars in Figure 16 is not meaningful.

Figure 17 gives averages of the results of Figures 14 and 15 by each program group; the averages are over programs from a group that contained `try` statements, or those that contained `throw` statements, and not over all programs from that group. The figure shows that among the `jacorb` programs that contained `try` statements, on average, there was one `try` statement for every 24 statements; a `throw` statement appeared

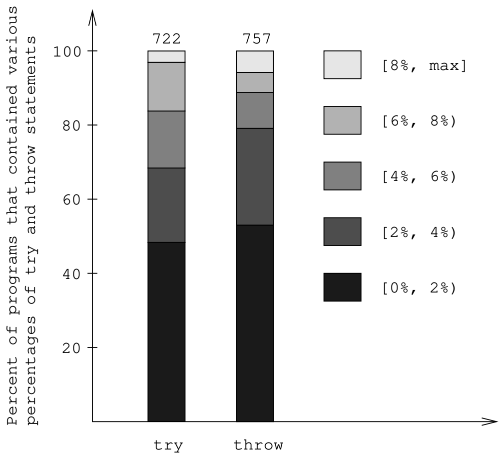


Figure 16: Frequency of occurrence of `try` and `throw` statements in programs that contained `try` and `throw` statements, respectively.

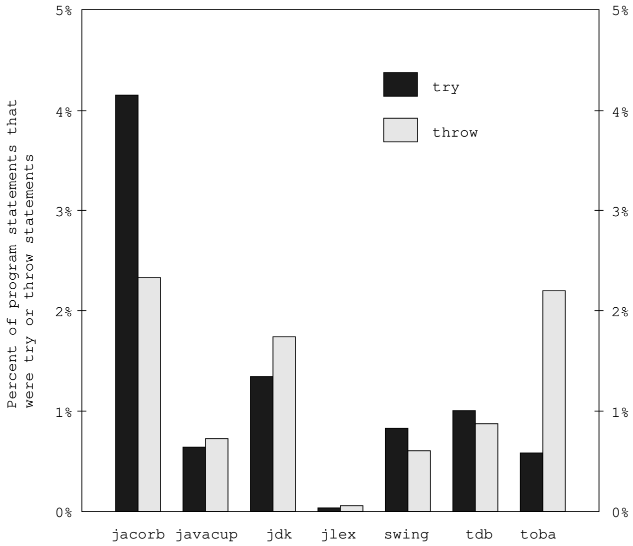


Figure 17: Frequency of occurrence of `try` and `throw` statements in programs from each program group.

once in every 43 statements in programs that contained `throw` statements. When averaged across all program groups, a `try` statement appeared once in every 58 statements, and a `throw` statement appeared once 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 Java front-end analyzer to generate control-flow representation that can be interfaced with other analysis tools from the Aristotle Program Analysis System [10]. Such a tool would facilitate future experiments to study the impact of exception-handling statements on program-analysis data.

Database

Emp ID	Name	Dept
101	Jones, Jim	Marketing
102	Neal, Mary	Info Systems
105	Scott, Sue	Accounting

Test Information

Test	SQL Commands	Branches Traversed
1	Q1	(6,7)
2	Q2, Q3	(6,8),(21,22),(22,23),(26,E exit1)
3	Q2, Q4	(6,8),(21,22),(22,24),(26,E exit2)
4	Q2, Q5	(6,8),(21,exit)

Queries

```
Q1: select * from employee where emp_id = 104
Q2: select * from employee where emp_id = 101
Q3: update employee set emp_id = 105 where emp_id = 101
Q4: update employee set last_name = "Black" where emp_id = 101
Q5: update employee set dept = "Sales" where emp_id = 101
```

Test History

Branches	Tests Cases That Cover	Branches	Tests Cases That Cover
(6,7)	1	(22,23)	2
(6,8)	2,3,4	(22,24)	3
(21,exit)	4	(26,E exit1)	2
(21,22)	2,3	(26,E exit2)	3

Figure 18: Testing information for program of Figures 5 and 7.

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, then we give an overview of an approach to performing that task for programs without exception-handling constructs, next, we describe how we can use this approach with our representation, and finally, we give an example that illustrates the technique.

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. *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.[10]

For testing Java programs with exception-handling constructs, CFGs presented in the previous sections can be used to identify the branches that require testing. Consider, for example, the partial Java program given in Figures 3 and 10. This program has eight branches, all of which are feasible: (6,7), (6,8), (21,exit), (21,22), (22,23), (22,24), (26,E exit1), and (26, E exit2). Suppose the database contains the records shown in the Database table in Figure 18. Suppose further that we construct the test suite with four tests that use queries `Q1` – `Q5`, also shown in the figure. The **Test Information** table lists the test cases, the queries from which the test cases are constructed, and the branches that are covered by the test cases. By considering the branches traversed, we see that all branches are exercised by some test case. Thus, this test suite covers all eight branches in the program and provides 100% branch coverage.

The precision of the type-inference algorithm used to generate control-flow representations has an impact on structural testing: an imprecise type-inference algorithm results in infeasible branches in a program, and

[10]For some branches, there may be no input that will cause the branch to execute — these branches are *infeasible*. We can achieve 100% branch coverage only if we remove infeasible branches from consideration.

therefore, no test suite would provide 100% branch 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. One regression testing strategy reruns all such test cases, but this *retest-all* approach 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.

Rothermel and Harrold developed a regression test selection technique [20] that uses a program's CFG. Their technique consists of a number of steps. First, the technique 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.

To date, this technique has been applied to object-oriented programs [19][11] but has not considered the effects of exception-handling constructs on the algorithms. With minor modification, however, the algorithm can use our CFGs, and thus, provide regression-test selection for Java programs that contain exception-handling constructs.

Consider for example, the partial Java program given in Figures 3 and 10; call this program `original`. The **Test History** (shown in Figure 18) shows the branches that are covered by the test cases in the test suite.[12] Suppose that we change line 8 in `original` to call `updateFields2()` instead of `updateFields()`; call this new program `modified`. The algorithm constructs CFG-modified for `modified` and begins the depth-first traversal of CFG and CFG-modified. When the traversal reaches node 6, and it considers the text associated with the sinks of edges (6,8) in `original` and `modified`, it finds that they differ. Thus, it selects test cases 2, 3, and 4 to execute during regression testing; test case 1 does not need to be rerun.

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

[11] The first version of the regression test selection algorithm used the control-dependence graph; the later version simplified the technique and, instead used the CFG.

[12] Rothermel and Harrold's algorithm actually gathers a test history for edges in the program; we can create such a test history from this branch-based test history. For brevity, however, we use the branch-based test history for our example.

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.

References

[1] H. Agrawal. On slicing programs with jump statements. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation. SIGPLAN Notices*, pages 302–12, June 1994.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Prin., Techniques, and Tools.* Addison-Wesley Publishing Company, Reading, MA, 1986.

[3] T. Ball and J. R. Larus. Efficient path profiling. In *Proc. of Micro 96*, pages 46–57, December 1996.

[4] R. K. Chatterjee, B. G. Ryder, and W. A. Landi. Complexity of concrete type-inference in the presence of exceptions. *To appear in European Symposium on Programming*, Apr. 1998.

[5] 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.

[6] P. Frankl and E. J. Weyuker. An applicable family of data flow criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, October 1988.

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

[8] R. Gupta, M. J. Harrold, and M. L. Soffa. Program slicing-based regression testing techniques. *Journal of Software Testing, Verification, and Reliability*, 6:83–111, 1996.

[9] M. J. Harrold and G. Rothermel. Syntax-directed construction of program dependence graphs. Technical Report OSU-CISRC-5/96-TR32, The Ohio State University, May 1996.

[10] M. J. Harrold and G. Rothermel. Aristotle: A system for research on and development of program-analysis-based tools. Technical Report OSU-CISRC-3/97-TR17, The Ohio State University, March 1997.

[11] 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.

[12] M. J. Harrold and M. L. Soffa. Selecting data for integration testing. *IEEE Software*, pages 58–65, March 1991.

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

[14] D. Lance. Java program analysis: A new approach using Java virtual machine bytecodes. Master's thesis, Middle Tennessee State University, 1997.

[15] 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.

[16] L. Larsen and M. J. Harrold. Slicing object-oriented software. In *Proceedings of 18th Internarional Conference on Software Engineering*, pages 495–505, March 1996.

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

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

[19] G. Rothermel and M. J. Harrold. Selecting regression tests for object-oriented software. In *Proceedings of the Conference on Software Maintenance - 1994*, pages 14–25, September 1994.

[20] 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.

[21] B. G. Ryder and M. C. Paull. Incremental data flow analysis algorithms. *ACM Trans. on Prog. Lang. and Sys.*, 10(1):1–50, Jan. 1988.