Analysis and Testing of Programs with Exception-Handling Constructs

Saurabh Sinha and Mary Jean Harrold
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
{sinha,harrold}@cc.gatech.edu

Abstract

Analysis techniques, such as control flow, data flow, and control dependence, are used for a variety of software-engineering tasks, including structural and regression testing, dynamic execution profiling, static and dynamic slicing, and program understanding. To be applicable to programs in languages such as Java and C++, these analysis techniques must account for the effects of exception occurrences and exception-handling constructs; failure to do so can cause the analysis techniques to compute incorrect results and thus, limit the usefulness of the applications that use them. This paper discusses the effect of exception-handling constructs on several analysis techniques. The paper presents techniques to construct representations for programs with explicit exception occurrences — exceptions that are raised explicitly through throw statements — and exception-handling constructs. The paper presents algorithms that use these representations to perform the desired analyses. The paper also discusses several software-engineering applications that use these analyses. Finally, the paper describes empirical results pertaining to the occurrence of exception-handling constructs in Java programs, and their impact on some analysis tasks.

Keywords: Exception handling, control-flow analysis, control-dependence analysis, data-flow analysis, program slicing, structural testing.

1 Introduction

Many software-engineering tasks, such as test-coverage analysis, test-case generation, regression testing, dynamic execution profiling, impact analysis, and static and dynamic slicing (e.g., [1, 2, 3, 4]), require information about the control flow, control dependence, and data dependence among statements in a program. Much research has addressed the problems of computing such analysis information for individual procedures (*intraprocedural*)[1] [5] and for interacting procedures (*interprocedural*) [6]. 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., [7]). 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 [8]. To be applicable to programs written in languages, such as Java [9] and C++,[2] however, these analysis techniques should, to the extent possible, account for the effects of exception-handling constructs.

[1] Analyses and representations that can be applied to individual procedures can also be applied to individual methods. Thus, we sometimes use "procedure" to mean both procedure and method.

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

Table 1: Frequency of occurrence of exception-handling statements in Java programs.

Subject		Number of	Number of	Methods with
Name	Description	classes	methods	EH constructs
antlr	Framework for compiler construction	175	1663	175 (10.5%)
debug	Sun's Java debugger	45	416	80 (19.2%)
jaba	Architecture for analysis of Java bytecode	312	1615	200 (12.4%)
jar	Sun's Java archive tool	8	89	14 (15.7%)
jas	Java bytecode assembler	118	408	59 (14.5%)
jasmine	Java Assembler Interface	99	627	54 (8.6%)
java_cup	LALR parser generator for Java	35	360	32 (8.9%)
javac	Sun's Java compiler	154	1395	175 (12.5%)
javadoc	Sun's HTML document generator	3	99	17 (17.2%)
javasim	Discrete event process-based simulation package	29	216	37 (17.1%)
jb	Parser and lexer generator	45	543	55 (10.1%)
jdk-api	Sun's JDK API	712	5038	582 (11.6%)
jedit	Text editor	439	2048	173 (8.4%)
jflex	Lexical-analyzer generator	54	417	31 (7.4%)
jlex	Lexical-analyzer generator for Java	20	134	4 (3.0%)
joie	Environment for load-time transformation of Java classes	83	834	90 (10.8%)
sablecc	Framework for generating compilers and interpreters	342	2194	106 (4.8%)
swing-api	Sun's Swing API	1588	12304	583 (4.7%)
Total		3951	30400	2467 (8.1%)

Exception-handling constructs provide a mechanism for raising exceptions, and a facility for designating protected code by attaching exception handlers to blocks of code. Failure to account for the effects of exception-handling constructs in performing analyses can result in incorrect analysis information, which in turn can result in unreliable software tools. For example, a branch-coverage testing tool for C++ that fails to recognize the flow of control among exception-handling constructs cannot adequately measure the branch coverage of a test suite. As a further example, a slicing tool for Java that fails to recognize the flow of control among exception-handling constructs cannot accurately compute control and 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 constructs, we conducted a study in which we examined a variety of Java programs. From each subject, we determined the percentage of methods that contained either a `throw` or a `try` statement. The number of methods in the subjects ranged from 89 to 12304. Table 1 summarizes the results of the study.

The data in the table illustrates that, on average, 8.1% of the methods contain some form of exception-handling construct. In a previous study [10], with a smaller suite of Java subjects, we had examined the occurrence of exception-handling constructs in classes. In that study, we observed that 23.3% and 24.5% of the classes contained `try` and `throw` statements, respectively. In another recent study, Ryder and colleagues [11] also studied the frequency with which Java programs use exception-handling constructs, and found that 16% of the methods that they examined contained exception-handling constructs. Our subjects include only four of the subjects that were used in their study, which explains the differences in the results. The results of the two studies are similar for the four subjects — `jas`, `jasmin`, `joie`, and `jflex` — that were common to both studies. These results support our belief that, in practice, the use of exception-handling constructs in Java programs is significant enough that it should be considered during various analyses.

Recently, several researchers have considered the effects of exception-handling constructs on various types of analyses. One approach constructs control-flow representation for exception-handling constructs, and uses the representation to perform data-flow analyses [12]. Another approach considers the control flow caused by exceptions while performing points-to and data-flow analyses [13, 14]. Other research has analyzed the flow of exceptions, and built tools to facilitate understanding of the exceptional behavior of programs [15, 16]. None of that research, however, considers the effects of exceptions on analysis techniques such as control-dependence analysis and program slicing.

To facilitate such analyses for software-engineering tasks, we investigated the effects of exception-handling constructs on various types of analyses, developed new techniques to perform these analyses in the presence of exceptions, and developed representations for the analysis information on which other analyses and applications can be applied. In this paper, we present our results for two analysis techniques: control-flow and control-dependence analysis. We also discuss briefly the use of our analysis information in two applications: program slicing and structural testing. We discuss the problems and solutions for Java-like exception-handling constructs; constructs in other languages, such as C++, can be analyzed similarly. In the Java exception-handling paradigm, an exception can be raised explicitly through a `throw` statement, or implicitly, through a call to a library routine or by the runtime environment. The techniques presented in the paper apply only to explicitly raised exceptions. Our current work includes investigation of ways to extend our techniques to include the analysis of implicitly raised exceptions. We also restrict our discussion to problems, representations, and analyses for exception-handling constructs; techniques for handling other features of object-oriented languages, such as polymorphism and object flow, are discussed elsewhere [13, 17, 18].

In this paper, apart from the study of the frequency with which exception-handling constructs occur in Java programs (Table 1), we also present the results of two other empirical studies. We performed these studies using our program analysis system, Java Architecture for Bytecode Analysis (JABA), written in Java, that analyzes Java bytecode files.[3] The first empirical study evaluates the precision of the control-flow representations that we construct for exception-handling constructs, and suggests that exhaustive type-inference analysis may not be required for determining exception types for `throw` statements. The second empirical study examines the effects of exception-handling constructs on control dependences. The results from this study indicate that a control-dependence computation that ignores the effects of exception-handling constructs can fail to identify a number of dependences in a program. These omitted dependences can have significant impact on the accuracy of tools that require such dependences.

The next section gives an overview of exception-handling constructs and specifies those constructs that our techniques handle. After introducing an example that is used through the rest of the paper, Section 3 discusses the effects of exception-handling constructs on several types of analyses. Next, Section 4 presents our analysis techniques, the representations that are constructed by those techniques, and some empirical studies pertaining to the techniques. Then, Section 5 briefly discusses the use of our representations for program slicing and structural testing. Section 6 evaluates our analysis techniques in terms of their accuracy and limitations, and discusses the tradeoffs involved in analyzing exception-handling constructs with various degrees of accuracy. Section 7 discusses related work. Finally, Section 8 presents conclusions and potential future work.

[3] JABA provides language-dependent analysis for Java programs (at the byte-code level) that is required for use in language-independent tools that are part of the Aristotle Analysis System [19].

```
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
   . . .
}
```

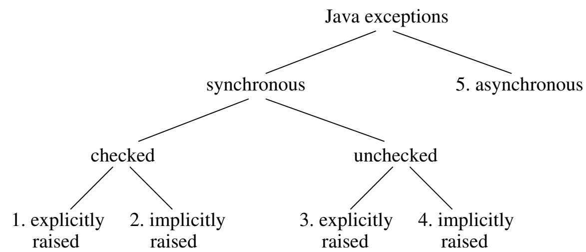


Figure 1: The syntax of exception-handling constructs in Java (left), and Java Exception types (right).

2 Exception-Handling Constructs

This section provides an overview of exception-handling constructs in Java, our language model; details of the Java language can be found in Reference [9]. Other languages, such a C++ and Ada, provide similar exception-handling mechanisms.

In Java, an exception is an object: each exception is an instance of a class that is derived from the class `java.lang.Throwable`. An exception can be raised at any point in the program through a `throw` statement. The expression associated with the `throw` statement denotes the exception object. The expression can be a variable (e.g., `throw e`), a method call (e.g., `throw m()`), or a new-instance expression (e.g., `throw new E()`). A `throw` statement can appear anywhere in the program; it may or may not be enclosed in a `try` statement.

A `try` statement provides the mechanism for designating guarded code, by associating exception handlers with the code. A `try` statement consists of a `try` block and, optionally, a `catch` block and a `finally` block. The legal instances of a `try` statement are `try–catch`, `try–catch–finally`, and `try–finally`. The code on the left in Figure 1 shows a typical `try` statement. A `try` block contains statements whose execution is 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 specifies the type of exception it handles, and contains a block of code that is executed when an exception of that type is raised in the associated `try` block. A `catch` clause also specifies a variable that is initialized with the handled exception, and whose scope is limited to the block of code for that `catch` clause. A `try` statement can have a `finally` block. The code in a `finally` block is always executed, regardless of the way in which control transfers out of the `try` block. Control may exit a `try` block by reaching the last statement in the `try` block, through an exception that may or may not be handled in the associated `catch` block, or because of a `break`, `continue`, or `return` statement.

Java follows the *nonresumable* model of exception handling: after an exception is handled, control does not return to the point at which the exception was raised, but continues at the first statement following the `try` statement that handled the exception. A Java exception can be propagated up on the call stack: if a method raises but does not handle an exception, the exception is reraised in the context of the caller of that method.

Exceptions in Java can be classified according to several criteria; the graph on the right in Figure 1 shows the classification criteria. These criteria reflect the semantics of raising an exception, and impose requirements on the way in which an exception must be handled. For example, a Java exception can be synchronous or asynchronous. A *synchronous exception* occurs at a particular program point and is caused by an expression evaluation, a statement execution, or an explicit `throw` statement. An *asynchronous exception*, on the other hand, can occur at arbitrary, nondeterministic points in the program. A synchronous exception can be checked or unchecked. For a *checked exception*, the compiler must find a handler or a signature declaration for the method that raises the exception. For an *unchecked exception*, the compiler does not attempt to find such an associated handler or a signature declaration. A synchronous exception is *explicitly raised* if the exception is raised by a `throw` statement in the application being analyzed. A synchronous exception is *implicitly raised* if the exception is raised through a call to a library routine or by the runtime environment. The source of an implicitly raised exception, therefore, lies outside the application being analyzed. For example, a call to the Java API method `java.util.Stack.pop()` can raise a `StackEmptyException`; any expression that dereferences an object reference can cause the Java runtime environment to raise a `NullPointerException`.

The techniques that we discuss in this paper do not apply to asynchronous exceptions; a safe approximation of program points that may raise such exceptions would include all statements in the program. The techniques also do not apply to implicitly raised exceptions. The analysis of this type of exceptions is beyond the scope of this paper; our current work includes investigating ways to extend our work to include them.

3 Effects of Exception-Handling Constructs on Analyses

Exception-handling constructs belong to a class of control structures that cause arbitrary interprocedural control flow, and affect program-analysis techniques in similar ways. Other examples of such control structures include interprocedural jump statements, such as the `setjmp()`–`longjmp()` calls in C, and halt statements, such as the `exit()` call in C. Such constructs affect the flow of control across procedures, and in doing so, affect all analyses that are derived from control-flow analysis. The common effect of such control structures is that, at a call site, control may not return from the called procedure. Instead, control may return to a different point in the calling procedure, or control may not return to the calling procedure at all. Through such an effect, the control structures influence program-analysis techniques, such as control-flow analysis, data-flow analysis, and control-dependence analysis.

The emphasis of previous [20] and ongoing work is to characterize formally the control structures, to provide not only a better understanding of the common effects of the control structures on analysis techniques, but to facilitate the development of a uniform approach to performing accurate analyses in their presence. Although recent research has addressed some of the issues for program analyses and program understanding that are caused by exception-handling constructs [13, 14, 12, 16, 15], none of that work describes the problems and the solutions for the general form of exception-handling constructs. Other research in the past [21, 7, 22] has addressed the problem of computing slices in the presence of control structures that cause arbitrary intraprocedural control flow, but those results do not apply to arbitrary control flow across procedures. We

```
public class VendingMachine {

  private int totValue;
  private int currValue;
  private int currAttempts;
  private Dispenser d;

  public VendingMachine() {
    totValue = 0;
    currValue = 0;
    currAttempts = 0;
    d = new Dispenser();
  }

  public void insert( Coin coin ) {
    int value = valueOf( coin );
    if ( value == 0 ) {
      throw new IllegalCoinException();
    }
    currValue += value;
    showMsg( "current value = "+currValue );
  }

  public void returnCoins() {
    if ( currValue == 0 ) {
      throw new ZeroValueException();
    }
    showMsg( "Take your coins" );
    currValue = 0;
    currAttempts = 0;
  }

  public void vend( int selection ) {
    if ( currValue == 0 ) {
      throw new ZeroValueException();
    }
    try {
      d.dispense( currValue, selection );
      int bal = d.value( selection );
      totValue += currValue - bal;
      currValue = bal;
      returnCoins();
    }
    catch( SelectionException s ) {
      currAttempts++;
      if ( currAttempts < MAX_ATTEMPTS ) {
        showMsg( "Enter selection again" );
      }
      else {
        currAttempts = 0;
        throw s;
      }
    }
    catch( ZeroValueException z ) {
    }
  }
}
```

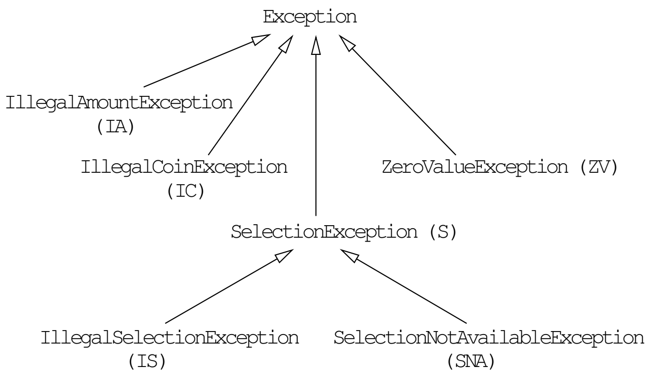
```
public class Dispenser {
  public void dispense( int currVal, int sel ) {
    Exception e = null;
    if ( sel < MIN_SELECTION || sel > MAX_SELECTION ) {
      showMsg( "selection "+sel+" is invalid" );
      e = new IllegalSelectionException();
    }
    else {
      if ( !available( sel ) ) {
        showMsg( "selection "+sel+" is unavailable" );
        e = new SelectionNotAvailableException();
      }
      else {
        int val = value( sel );
        if ( currVal < val ) {
          e = new IllegalAmountException( val-currVal );
        }
      }
    }
    if ( e != null ) {
      throw e;
    }
    showMsg( "Take selection" );
  }
}
```

```
public static void main() {
  VendingMachine vm = new VendingMachine();
  while ( true ) {
    try {
      try {
        switch( action ) {
          case INSERT: vm.insert( coin );
          case VEND:   vm.vend( selection );
          case RETURN: vm.returnCoins();
        }
      }
      catch( SelectionException s ) {
        showMsg( "Transaction aborted" );
        vm.returnCoins();
      }
      catch( IllegalCoinException i ) {
        showMsg( "Illegal coin" );
        vm.returnCoins();
      }
      catch( IllegalAmountException i ) {
        int val = i.getValue();
        showMsg( "Enter more coins"+val );
      }
    }
    catch( ZeroValueException z ) {
      showMsg( "Value is zero.  Enter coins" );
    }
  }
}
```

Figure 2: Java code for the vending-machine program: class `VendingMachine` (top), class `Dispenser` (bottom left), and method `main()` (bottom right).

discuss related work in more detail in Section 7.

In this paper, we restrict the discussion of the problems and solutions for exception-handling constructs. In the remainder of this section, we first describe a program that we use to illustrate the concepts presented in the paper. We then discuss the effects of exception-handling constructs on three program-analysis techniques:



Exception	Condition
IllegalAmount	User selects an item that costs more than the value of the coins inserted
IllegalCoin	User inserts an illegal coin
ZeroValue	User selects an item, or requests a refund without inserting any coins
IllegalSelection	User enters a number that does not correspond to a valid selection
SelectionNot-Available	User selects an item that is temporarily unavailable

Figure 3: Hierarchy of exception-related classes (and their abbreviated names) for the vending-machine program (left), and conditions that cause various exceptions to be raised (right).

control-flow analysis, data-flow analysis, and control-dependence analysis.

3.1 The Vending-Machine Program

The vending-machine program, shown in Figure 2, simulates the actions of a vending machine.[4] The machine lets a user insert coins, request a refund, or select an item using a numeric keypad. If the user selects a valid item and enters coins of value sufficient to cover the cost of the item, the machine dispenses the selected item. If the user makes an erroneous selection, the machine asks the user to reenter the selection. The user may reenter the selection or request a refund of the coins. The machine tracks the number of erroneous selections entered by a user; once the number of erroneous selections exceeds a predetermined value, the machine aborts the transaction and returns the user's coins. Figure 3 explains the various error conditions that may arise during a transaction and presents a class hierarchy of exceptions that correspond to those conditions.

Method **main()** (lines 42–58) presents the user with the three options, and based on the user's action, invokes one of three methods, defined in the class **VendingMachine**, to process the action. Method **insert()** (lines 5–9) first ensures that the user has entered a valid coin, and then increments the current value by the value of the coin; **insert()** raises an exception if the user enters an invalid coin. Method **returnCoins()** (lines 10–14) refunds coins with a value equal to the current value, and resets the current value; **returnCoins()** raises an exception if the current value is zero. Method **vend()** (lines 15–28) accepts the user's selection, and if the current value is not zero, invokes method **dispense()** defined in the **Dispenser** class. Method **dispense()** (lines 29–41) performs several error checks to ensure that the selection is a valid item (line 30), the selection is available for dispensing (line 33), and the current value covers the cost of the selection (line 37). If any of these checks fails, the code raises an appropriate exception. If all checks pass, **dispense()** simulates dispensing of the item by printing a message (line 41). On successful completion of **dispense()**, **vend()** updates appropriate state variables (lines 18–20), and calls **returnCoins()** to return the balance to the user (line 21). If **dispense()** raises an exception that signals an erroneous selection (lines 32, 35), **vend()** handles the exception (line 22) and increments **currAttempts**. When **currAttempts** exceeds the constant **MAX_ATTEMPTS**, **vend()** rethrows the caught exception (line 27), which causes **main()** to abort

[4] We adapted this example from one by Kung and colleagues that appeared in Reference [23].

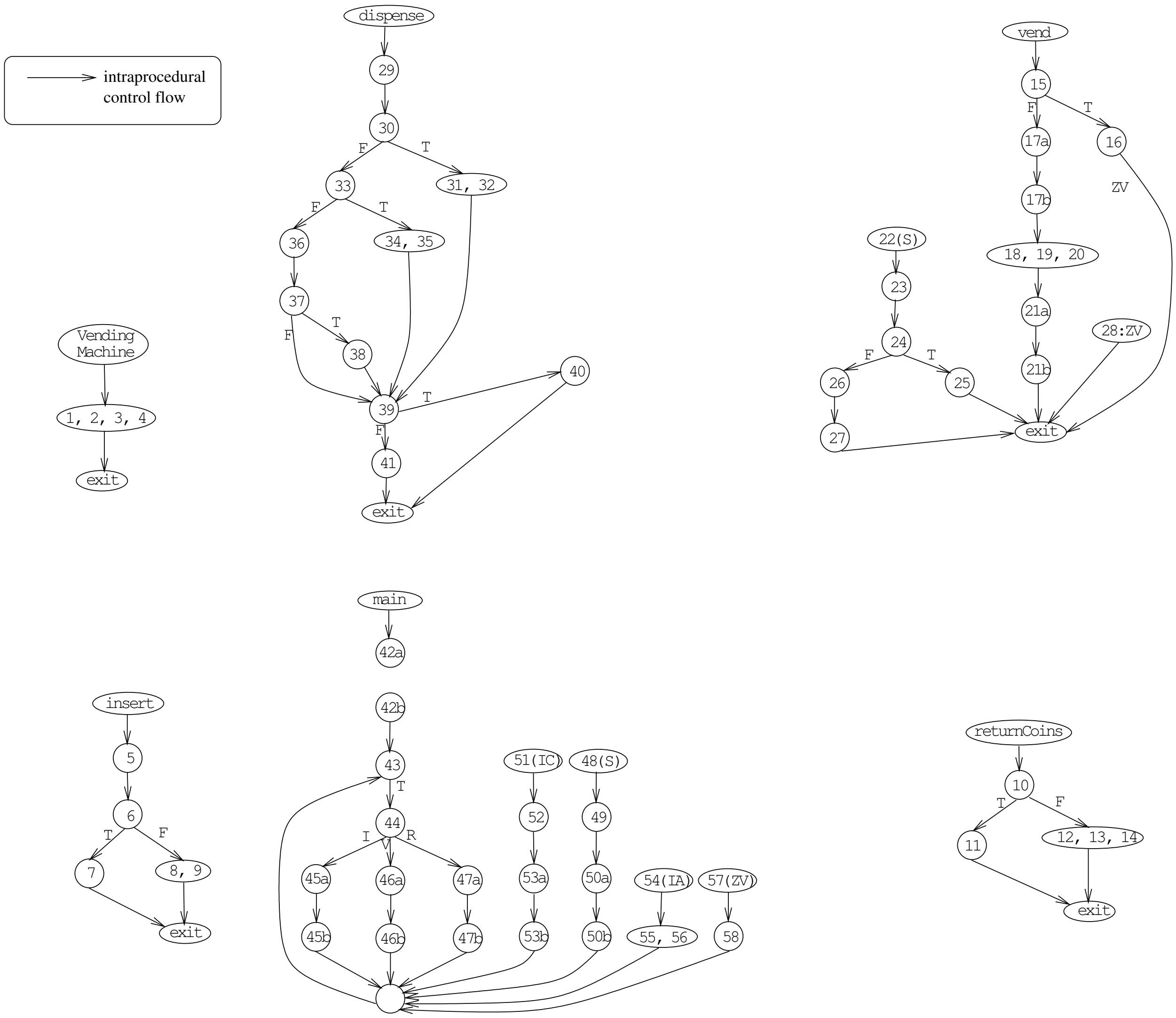


Figure 4: CFGs for methods of the vending-machine program constructed using an approach that does not consider the effects of exception-handling constructs.

---

the transaction.

We omit the details of methods, such as `value()` and `available()` in the `Dispenser` class, that are not relevant to our presentation; such methods raise no exceptions. For brevity, we also exclude the details of the constructors of some of the classes, and the initializations of constants such as `MAX_ATTEMPTS` and `MIN_SELECTION`.

3.2 Effect of Exceptions on Control-Flow Analysis

Control-flow analysis determines, for each program statement s, those statements in the program that could follow s in some execution of the program. Many program-analysis techniques, such as data-flow and control-dependence analyses, and software engineering tasks, such as structural and regression testing, use control-flow information about a program. These techniques typically construct a control-flow representation for the program. For these analyses to be useful, and for these applications to be effective in the presence of exception-handling constructs, the control-flow representation should incorporate the exception-induced control flow.

The vending-machine program of Figure 2 exemplifies the complexity that the presence of exception-handling constructs can introduce in the control flow in a program. For example, in `insert()`, control does not reach line 8 if the predicate in line 6 evaluates to true; instead, the exception raised in line 7 terminates the execution of `insert()`, and transfers control to a caller of `insert()`. For further example, consider the call to `dispense()` in line 17. Following the call, control may not return to the call site: if `dispense()` raises an exception, control may return to line 22 or control may not return to `vend()` at all. Through such effects, exception-handling constructs can influence control flow not only within a method, but across methods, and can introduce complex control-flow paths in a program.

The control-flow relation that exists in a program can be represented in a *control-flow graph* (CFG) in which nodes represent statements, and edges represent the flow of control between statements [24]. Figure 4 depicts the CFGs for the methods in the vending-machine program. Each node is labeled by the line numbers of the source statements that it represents. Each CFG has an entry node, which is labeled by the name of the corresponding method, and an exit node. Each call site is represented by a pair of nodes — a call node and a return node.[5] For example, the call site in line 17 is represented by call node 17a and return node 17b. The CFGs in Figure 4 do not represent exceptional control flow. Consequently nodes, such as 11 and 40, which represent `throw` statements, are connected to the exit nodes of their respective CFGs; nodes that represent `catch` statements, such as nodes 22 and 51, have no in edges.

Recent work [12] describes an intraprocedural representation of Java exception-handling constructs. That work, however, does not consider several issues related to control flow. For example, the work does not consider the control flow caused by the presence of `finally` blocks, and it does not model the propagation of exceptions by methods. In Section 4.1, we analyze the control flow caused by exception-handling constructs, and describe our approach for creating intraprocedural and interprocedural representations of programs that contain those constructs.

3.3 Effect of Exceptions on Data-Flow Analysis

Data-flow analysis techniques compute data-flow facts, such as definition-use pairs, reaching definitions, available expressions, and live variables, that hold at different program points. Data-flow information is used in activities such as program slicing [25, 26, 27], data-flow testing [1, 2, 28], and compiler optimizations [24]. A data-flow problem can be formulated as a set of equations that compute data-flow facts, and those data-flow facts are computed and propagated iteratively throughout the program using a control-flow representation. The solution of the data-flow problem is the fixed-point solution of the equations. In the presence of exception-handling constructs, the data-flow facts must be propagated also along the exceptional control-flow paths for the computed data-flow solutions to conservatively approximate the true data-flow solutions.

To illustrate, consider the effect of exception-handling constructs on the computation of definition-use pairs. A *definition-use pair* is a pair (d, u), where d is a statement that defines a variable v (references and changes v), u is a statement that uses v (references but does not change v), and there is a path in the

[5] Although separate call and return nodes are not required at a call site in the intraprocedural control-flow representation, we add them to facilitate later construction of the interprocedural control-flow representation.

program from d to u along which v is not redefined. Exception-handling constructs may cause a definition-use computation to miss definition-use pairs in two ways. First, a definition-use pair may not be detected because the pair occurs along only an exceptional-control flow path that is not modeled by the control-flow representation. For example, in the vending-machine program, there exists such a pair that includes statement 42, where `vm` is defined, and statement 53, where `vm` is used. This definition-use pair is not detected if the exceptional control flow from statement 7 to statement 51 is not modeled by the control-flow representation. Second, exception-handling constructs introduce additional definition-use pairs in a program through the exception object. The definition of exception object `e` in statement 38, and its subsequent use as `i` in statement 55 in an example of such a definition-use pair.

A data-flow relation can be represented as a *data-dependence graph* in which nodes represent program statements and edges represent the data dependence between statements. In such a graph, for definition-use pairs, a data-dependence edge exists between nodes n_1 and n_2 if (n_1, n_2) is a definition-use pair.

Several researchers have recently addressed the problem of performing data-flow analyses in the presence of exception-handling constructs. Some researchers [13, 14] do not explicitly create a control-flow representation for exceptions; instead, they modify the data-flow analyses to compute and traverse the intraprocedural and interprocedural exceptional control-flow paths while performing the desired analyses. Other researchers [12] represent some exceptional control flow explicitly, and modify the data-flow analyses to compute the remaining exceptional control flow while performing the analyses. With the control-flow representation that we define in Sections 4.1.1 and 4.1.2, existing algorithms for data-flow analyses require either no modifications or trivial modifications to function in the presence of exception-handling constructs. The other approaches work as well for performing the data-flow analyses; our representation provides an alternative approach to performing the analyses.

3.4 Effect of Exceptions on Control-Dependence Analysis

Control-dependence analysis [29, 5] determines, for each program statement, the predicates that control the execution of that statement. Informally, a statement s is control dependent on a predicate p if, in the CFG, there are two edges out of the node for p such that following one of the edges always results in the node for s to be reached, whereas following the other edge may cause that node not to be reached [5]. A statement in procedure P that is control dependent on no predicates in P is control dependent on entry into P. Control-dependence information is required for analyses, such as slicing, that are used for software-engineering tools, such as debuggers, impact analyzers, and regression testers.

Traditional definitions of, and algorithms for computing, control dependence [29, 30, 5] function at the intraprocedural level, and inaccurately model the correct control dependences when they are applied to programs that contain exception-handling constructs. One factor that causes the traditional definitions and algorithms to be inadequate is the presence of *potentially non-returning call sites* (PNRCs) [8]: call sites to which control may not return from the called methods. Through their effects on interprocedural control flow, exception-handling constructs cause PNRCs in a program, and necessitate the computation of interprocedural control dependence. For example, in the vending-machine program, the call site in line 17 is a PNRC because, following the call, control may return to statement 22 rather than to statement 17, or control may not return to `vend()` at all. This causes statements that follow the call site, such as statement 20, to be control dependent on conditional statements in the called methods. For example, statement 20 is

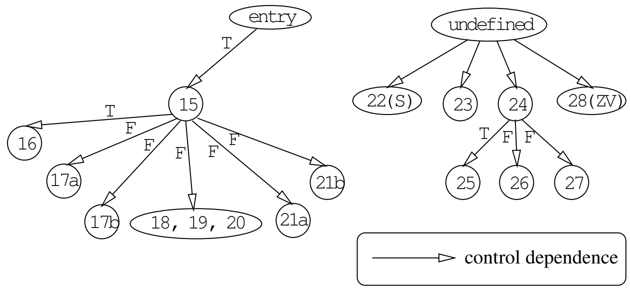


Figure 5: Control-dependence graph for method `vend()`.

control dependent on statement 39 in `dispense()`. Traditional techniques, however, identify statement 20 as control dependent on entry into `vend()`.

In the presence of exception-handling constructs, control dependences of certain statements — those that appear in a `catch` block — might be computable only in the interprocedural context; such statements have no intraprocedural control dependences. For example, the execution of statements 51–53, which belong in `main()`, is controlled solely by decisions that are made in `insert()`. Therefore, to identify the control dependences for such statements, interprocedural control dependences must be computed.

The control-dependence relation is represented as a graph. A *control-dependence graph* (CDG) [5] contains a node for each predicate and statement in a procedure, and an edge from predicate p to statement s if s is control dependent on p. A unique root node denotes the entry predicate, and represents the control dependences of those statements that are reached when control enters the procedure. Figure 5 shows the CDG for method `vend()` that is constructed from the CFG of `vend()` shown in Figure 4. The CDG in the figure has disconnected components because the control dependences of the `catch` handlers in `vend()` (and statements that are control dependent on entry into those handlers) can be computed only in the interprocedural context; intraprocedurally, the control dependences of those statements is undefined. In the figure, a unique node labeled "unknown" connects the nodes whose control dependences are undefined.

Past work that has attempted the computation of interprocedural control dependence [31] considers only the effects of halt statements, and suffers from several drawbacks. Recent work [8] has addressed and corrected those drawbacks, but considers only the effects of halt statements. In Section 4.2, we present an approach that identifies PNRCs, and computes correct control dependences in the presence of exception-handling constructs.

4 Analysis Techniques to Accommodate Exception-Handling Constructs

In this section, we describe techniques for control-flow and control-dependence analysis that account for the effects of exception-handling constructs, and therefore, can be applied to programs that contain such constructs.

To facilitate the ensuing discussion, we introduce some terminology and provide an informal definition of each term. A *program unit* is a program block or a method. A program unit *directly raises* an exception if it lexically encloses a `throw` statement. A program unit *indirectly raises* an exception if it lexically encloses

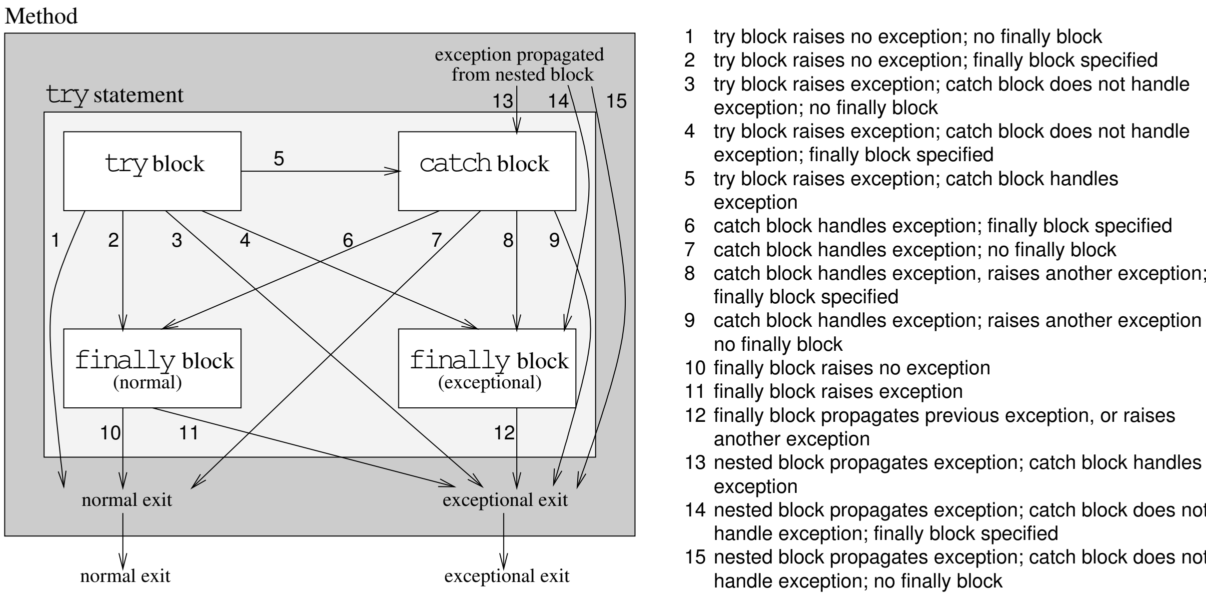


Figure 6: Intraprocedural control flow in Java exception-handling constructs.

a call site such that the called method propagates an exception.[6] A program unit *propagates* an exception if it raises an exception, but does not handle that exception. A program unit *directly* propagates an exception if it directly raises, but does not handle an exception; similarly, a program unit *indirectly* propagates an exception if it indirectly raises, but does not handle an exception. A `catch` clause is a *local handler* if it handles only those exceptions that are directly raised in the `try` block associated with that `catch` clause. A `catch` clause is an *interprocedural handler* if it handles only those exceptions that are indirectly raised in the `try` block associated with that `catch` clause. A `catch` clause is a *global handler* if it handles both directly and indirectly raised exceptions.

4.1 Control-Flow Analysis

As we saw in Section 3.2, the presence of exception-handling constructs creates control-flow paths within and across methods. To be useful, the intraprocedural and interprocedural control-flow representations must contain these paths.

4.1.1 Intraprocedural analysis

When an exception is raised in a `try` block, control transfers to the `catch` clause that handles the raised exception. This `catch` clause may be associated with the `try` block in which the exception is raised, or may be associated with a lexically enclosing `try` block. The parameter of the matching `catch` clause is bound to the thrown object, and the handler code is executed. Following the execution of the handler code, normal execution resumes at the first statement that follows the `try` statement in which the exception was handled. 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.

[6] The distinction between direct and indirect exceptions is not the same as the distinction between explicit and implicit exceptions. For example, an explicit exception can be raised directly or indirectly.

The block-level control-flow graph, shown in Figure 6, summarizes 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. As the figure illustrates, there are several control-flow paths within a `try` statement. For example, the path (5, 8, 12) is taken if the `try` block raises an exception, the `catch` block handles the exception but raises another exception, and the `finally` block raises no exception. Paths starting at edges 13, 14, or 15 are taken if a nested `try` statement propagates an exception. For example, path (13, 6, 11) is taken if a nested `try` statement propagates an exception that is handled in the `catch` block, and then the `finally` block raises another exception.

Figure 6 illustrates that a Java `finally` block can execute in one of two contexts: a normal context or an exceptional context. A `finally` block executes in a *normal context* when (1) control reaches the end of a `try` block or a `catch` block, or (2) control leaves a `try` statement because of an unconditional transfer statement, such as `break`, `continue`, or `return`. A `finally` block executes in an *exceptional context* when control leaves a `try` statement because of an unhandled exception; the unhandled exception may have been raised directly or indirectly within the `try` statement. The context of execution of a `finally` block determines where control flows from that `finally` block: in a normal context, control flows to the statement that follows the `try` statement, or control flows to the target of an unconditional transfer statement; in an exceptional context, control flows to an enclosing `finally` block, an enclosing `catch` handler, or control exits the method with an unhandled exception.

To represent exceptional-handling constructs, the CFG contains nodes that represent `throw` statements, `catch` handlers, and `finally` blocks, and edges that represent the normal and exceptional control flow caused by those constructs. A throw node can have multiple successors in the CFG; these successors are determined by the types of exceptions that can be raised at the corresponding `throw` statement. To determine the potential exception types, we perform type inference and create one out edge from the throw node for each type of exception.[7] We label each edge with the type of exception that causes that edge to be traversed during program execution. Figure 7 presents the CFGs for the methods of the vending-machine program that are constructed using our approach. The `throw` statement in line 40 of the program can raise one of three types of exceptions: `SNA`, `IA`, or `IS`. Therefore, the CFG node for that statement has three out edges — one for each type of exception — that are labeled by the corresponding exception types. If a `throw` statement raises only one type of exception, the CFG node for that statement is also labeled by the exception type. For example, node 11 has a single out edge labeled "ZV" because the corresponding `throw` statement raises an exception of type `ZV`.

To model propagation of exceptions out of a method, the CFG contains exceptional-exit nodes. An *exceptional-exit node* is an exit point in the CFG that has a type T associated with it, and represents the propagation of an exception of type T by the corresponding method. The CFG of a method has as many exceptional-exit nodes as the distinct types of exceptions that are directly propagated by the method. In Figure 7, the CFG for `vend()` has four exceptional-exit nodes because `vend()` directly propagates four types of exceptions: `S`, `IS`, `SNA`, and `ZV`. The CFGs for `insert()` and `returnCoins()` have one exceptional-exit node each because both these methods directly propagate a single exception type. Method `vend()` indirectly propagates `IA`, through the call to `dispense()`; the exceptional-exit node for the indirectly propagated exception is created in the interprocedural representation (Section 4.1.2).

[7] Section 4.1.3 provides further discussion of the type inferencing.

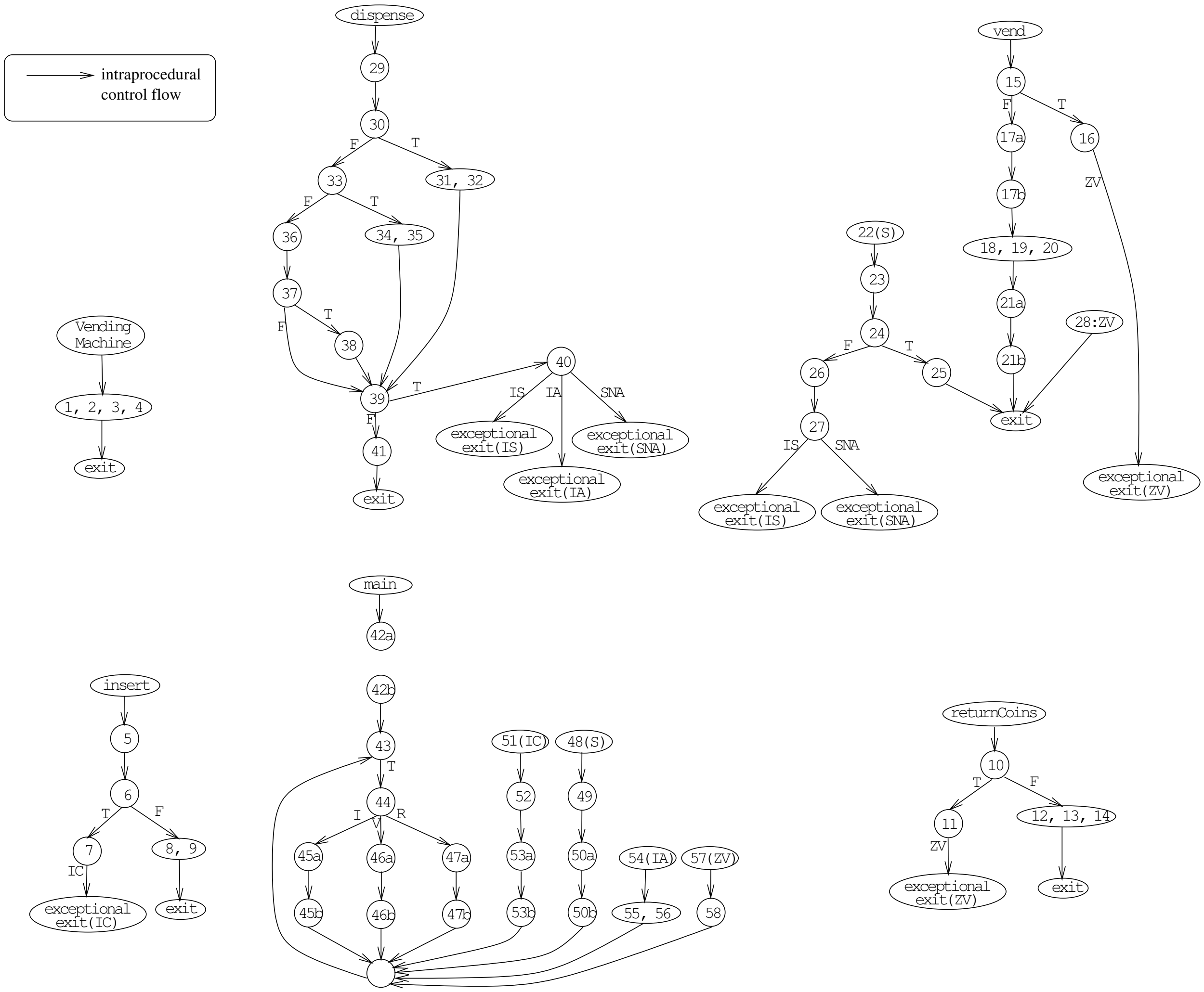


Figure 7: CFGs for methods of the vending-machine program constructed using our approach.

In the CFG, the node for a local handler has in edges for exceptions that are caught by that handler. A global handler also catches directly raised exceptions, and therefore, the node for such a handler also has in edges in the CFG. An interprocedural handler, however, catches only indirectly raised exceptions — exceptions that are raised in methods called from the method that contains the handler. Therefore, the node for an interprocedural handler has no in edges in the CFG; such a handler has in edges in the interprocedural representation. Because all handlers in the vending-machine program are interprocedural handlers, nodes in the CFGs that correspond to those handlers have no in edges.

A `finally` block can execute in several different contexts such that following the execution of statements in the block, control flows to different points in different contexts. There are two alternative approaches to to model such control flow without introducing paths that represent illegal entry–exit sequences for `finally` blocks. One approach creates a separate CFG for each `finally` block, and inserts call nodes to the `finally` blocks for both contexts of execution. The other approach avoids creating a separate CFG for each `finally` block, and instead inlines a `finally` block once for each of its different contexts of execution. The second approach becomes impractical if `finally` blocks appear frequently and are large.

```
algorithm ConstructCFG
input     AST : abstract-syntax tree for procedure P
output    CFG : control-flow graph for procedure P
begin ConstructCFG
    /* Step 1 — construct incomplete CFG */
1.  construct control-flow graph with no out edge from throw nodes
    /* Step 2 — perform type inference7 */
2.  perform local flow-sensitive type analysis
3.  perform global flow-insensitive type analysis
    /* Step 3 — construct complete CFG */
4.  create out edges from throw nodes
5.  create exceptional-exit nodes for propagated exception types
end ConstructCFG
```

Figure 8: Overview of the CFG-construction algorithm.

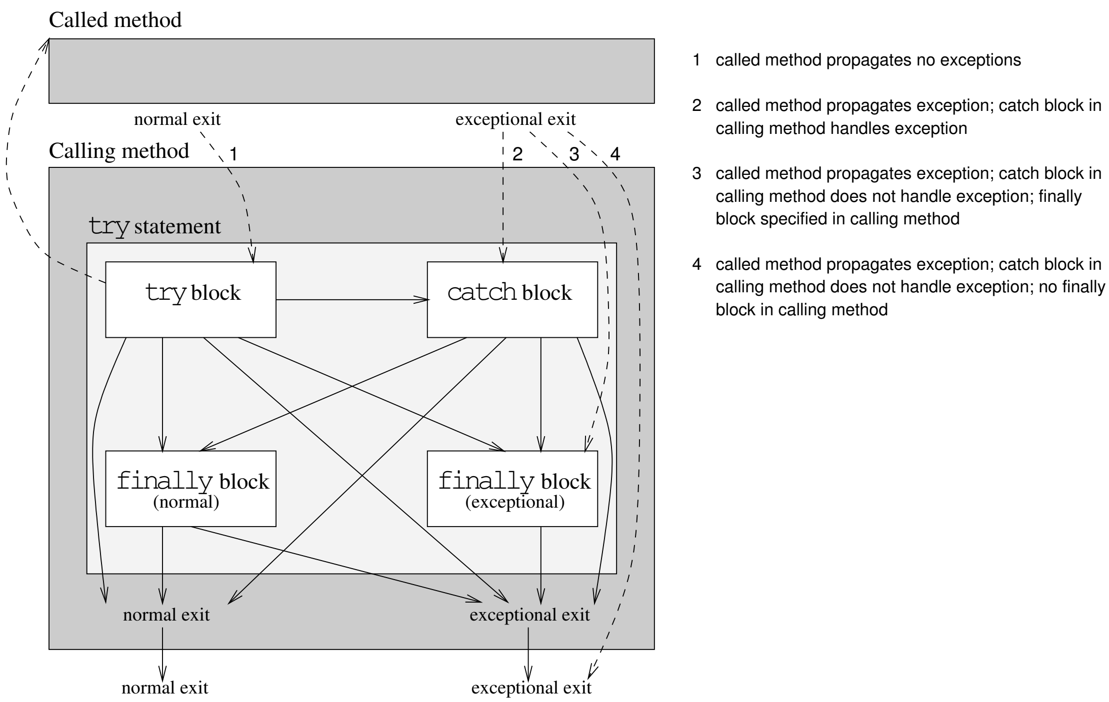


Figure 9: Interprocedural control flow in exception-handling constructs .

Figure 8 provides an overview of the CFG-construction algorithm [32]. The algorithm operates in three steps: First, the algorithm creates an incomplete CFG in which throw nodes have no out edges; next, the algorithm performs type inferencing using the incomplete CFG to determine potential exception types for **throw** statements;[7] finally, the algorithm completes the CFG by adding out edges from the throw nodes, and creating the necessary exceptional-exit nodes and call nodes for **finally** blocks. The first step of algorithm can be implemented using either an abstract-syntax tree, as described in Reference [32], or the Java bytecodes. Our Java analysis tool constructs the CFGs using the bytecode-based implementation of the algorithm.

4.1.2 Interprocedural analysis

The propagation of exceptions on the call stack creates interprocedural exceptional control flow. Interprocedural control flow is represented in an interprocedural control-flow graph. An *interprocedural control-flow graph* (ICFG) for a program $\mathcal{P}$ consists of CFGs for each method or procedure in $\mathcal{P}$; at each call site, the call node is connected to the entry node of the called method by a call edge, and the exit node of the called method is connected to the corresponding return node by a return edge.

Figure 9 presents an interprocedural block-level control-flow graph (similar to Figure 6) 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. Following the execution of B, control can return to A in one of four ways; the edges corresponding to these returns are labeled in the figure. If B propagates no exceptions, control returns normally to the statement following the call site in A. If B propagates an exception, control does not return to the call site. If the `try` block in A has an associated `catch` handler that handles the raised exception, control flows to that handler. If there is no such `catch` handler associated with the `try` block but that block has a corresponding `finally` block, control flows to the `finally` block. If neither of the above is true, method A also propagates the exception, and the search for a handler continues in the caller of A.

To represent the interprocedural exceptional control flow, the ICFG contains exceptional-return edges. An *exceptional-return edge* is an interprocedural edge that connects an exceptional-exit node of the called method to a catch node, a call node that calls a `finally` block, or an exceptional-exit node in the calling method.

Figure 10 shows the ICFG for the vending-machine program. Each call node is connected to the entry node of the CFG of the called method, and the exit node of that CFG is connected to the corresponding return node. If a method propagates an exception that is caught in the caller of that method, the exceptional-exit node for that exception type is connected to the appropriate node in the caller by an exceptional-return edge. For example, `insert()` propagates `IC` that is caught in statement 51 of `main()` (the caller of `insert()`). Therefore, the exceptional-exit node in the CFG for `insert()` is connected, by an exceptional-return edge, to node 51 in the CFG for `main()`. A method may propagate an exception that is not handled in the immediate caller of that method, but is handled in a method that lies further up in the call sequence. For example, `main()` calls `vend()`, and `vend()` calls `dispense()`. `dispense()` propagates an exception of type `IA`; `vend()`, however, does not handle the exception but (indirectly) propagates it up to `main()`. The chain of exceptional-return edges in the ICFG reflect the exception propagation: the exceptional-exit node for type `IA` in the CFG for `dispense()` is connected to the exceptional-exit node for the same type in the CFG for `vend()`, which in turn is connected to the catch node 54 in the CFG for `main()`.

The ICFG-construction algorithm [32] iteratively determines indirectly propagated exception types for each method, and adds exceptional-exit nodes to the CFG of the method for those exception types. For example, the ICFG-construction algorithm determines that `vend()` indirectly propagates `IA` (through the call to `dispense()`), and adds an exceptional-exit for type `IA` to the CFG of `vend()`. Figure 11 provides a high-level view of the algorithm. The algorithm initializes a worklist with the methods in the program (line 1), and then repeatedly removes a method N from the worklist and processes all callers of N, until the worklist becomes empty (lines 2–14). For each call site that calls N, the algorithm creates call and return edges (line 5). The exceptions types that are propagated by N (indicated by the exceptional-exit nodes

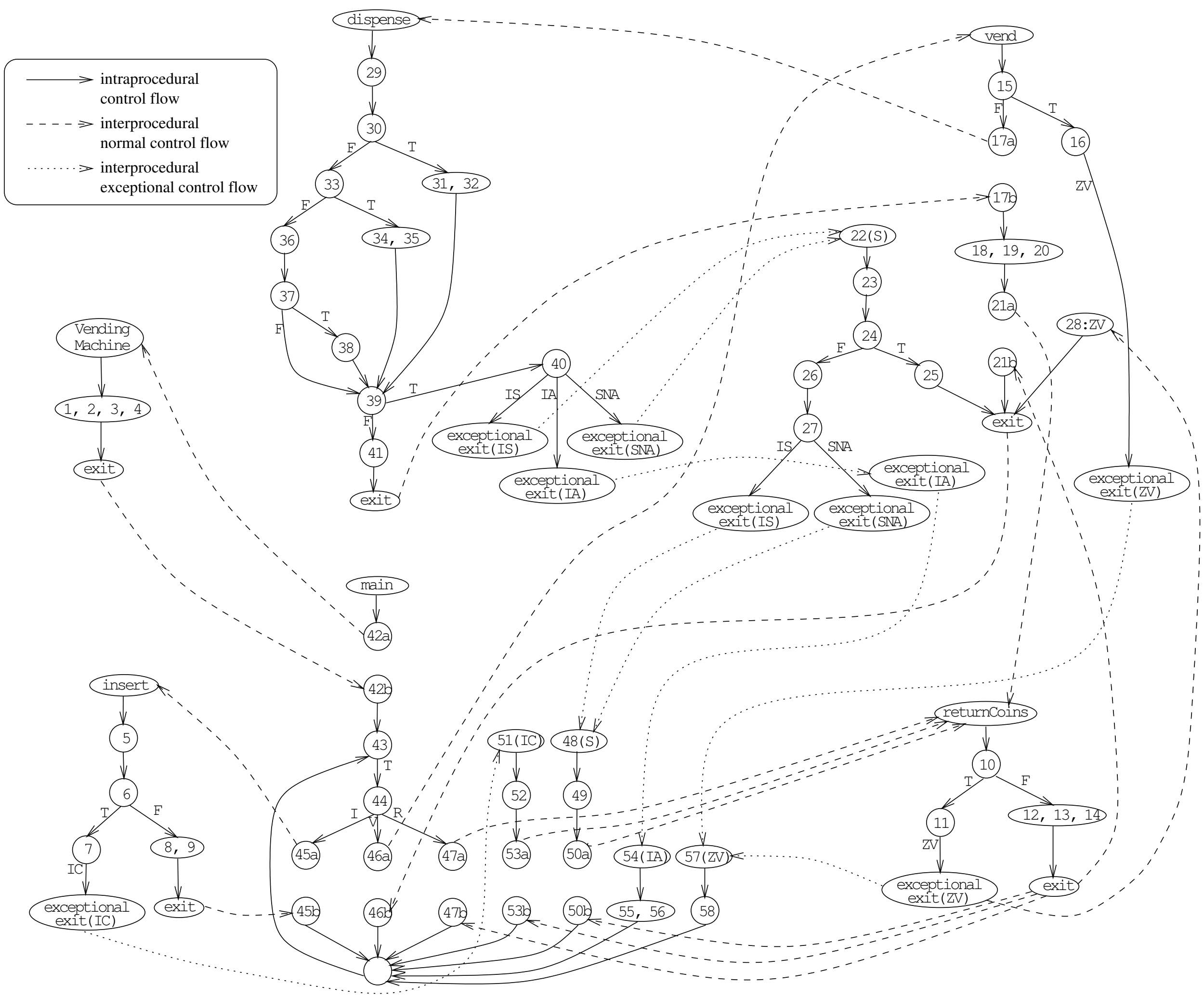


Figure 10: ICFG for the vending-machine program.

in the CFG of N) are indirectly raised in M. Therefore, for each such exception type, the algorithm adds finally-call nodes and an exceptional-exit node to the CFG of M (line 7), if such nodes are required; for example, if an exception type is propagated by N and is not caught by M, an exceptional-exit node for that type is added to the CFG of M. For each exception type propagated by N, the algorithm also creates an exceptional-return edge (line 8). If the algorithm adds an exceptional-exit node to the CFG of M, it adds M to the worklist (line 10), to ensure that all callers of M are reprocessed.

Like other iterative data-flow algorithms, the ICFG-construction algorithm can be implemented efficiently to process nodes in a reverse topological order of the program's call multigraph.[8] Such an implementation processes each nonrecursive method only once, and the recursive methods iteratively, as shown in Figure 11, until a fixed point is reached.

[8] A *call multigraph* for a program $\mathcal{P}$ contains a node N for each method n $\mathcal{P}$, and an edge from node N_i to node N_j for each call site in the method corresponding to N_i that calls the method corresponding to N_j.

```
algorithm ConstructICFG
input      CFG : control-flow graph for each method in P
output     ICFG : interprocedural control-flow graph for P
declare    worklist : methods that are processed iteratively
begin ConstructICFG
initialize worklist with methods in P
while worklist is not empty
    remove method N from worklist
    foreach call site in method M that calls N
        create call edge, return edge
        foreach exceptional-exit node in the CFG of N
            add finally-call nodes, exceptional-exit node to
            the CFG of M, if required
            create exceptional-return edge
            if exceptional-exit node added to the CFG of M
                add M to worklist
            endif
        endfor
    endfor
endwhile
end ConstructICFG
```

Figure 11: Overview of the ICFG-construction algorithm.

4.1.3 Type inferencing for exception types

The CFG construction requires information about exception types that can be raised at **`throw`** statements. *Precise type information* at a **`throw`** statement includes only those types that can be raised at that **`throw`** statement in some execution of the program. The precision of the type inference determines the extent to which infeasible paths[9] are introduced in the control-flow representations. An imprecise (but safe) approximation of exception types causes the addition of unnecessary edges emanating from throw nodes; programs paths that contain such edges are infeasible.

Type-inference algorithms (e.g., [33, 34]) attempt to determine the types for each expression in a program by solving type constraints or by propagating local type information throughout a program. Such techniques have been applied traditionally to optimization of dynamically dispatched function calls. Recent work [13] uses points-to analysis to infer types in programs that contain exception-handling constructs.

Type inference for exception types is required only for those **`throw`** statements whose exception types cannot be determined by an inspection of the **`throw`** statement; the expressions of such **`throw`** statements are variables or method calls. For example, a **`throw`** statement, such as the one in line 11 of the vending-machine program, requires no type inference because its expression is a new-instance expression; the only type of exception that can be raised at that statement is **`ZV`**. The **`throw`** statement in line 40, however, requires type inference because it raises the exception object referenced by a variable, and different exception objects are created and assigned to that variable along different paths to the **`throw`** statement.

Our empirical evidence, based on the subjects listed in Table 1, suggests that, in practice, the expressions of an overwhelming majority of **`throw`** statements are new-instance expressions, and therefore, require no type-inference analysis. Table 2 lists the types of throw-statement expressions that appear in our subjects. As the data illustrates, out of the 2490 **`throw`** statements that appear in the subjects, only 59 have either a variable or a method call as their expressions. Among these **`throw`** statements, a variable expression appears

[9] A path is *infeasible* if there exists no input to the program that causes the path to be executed.

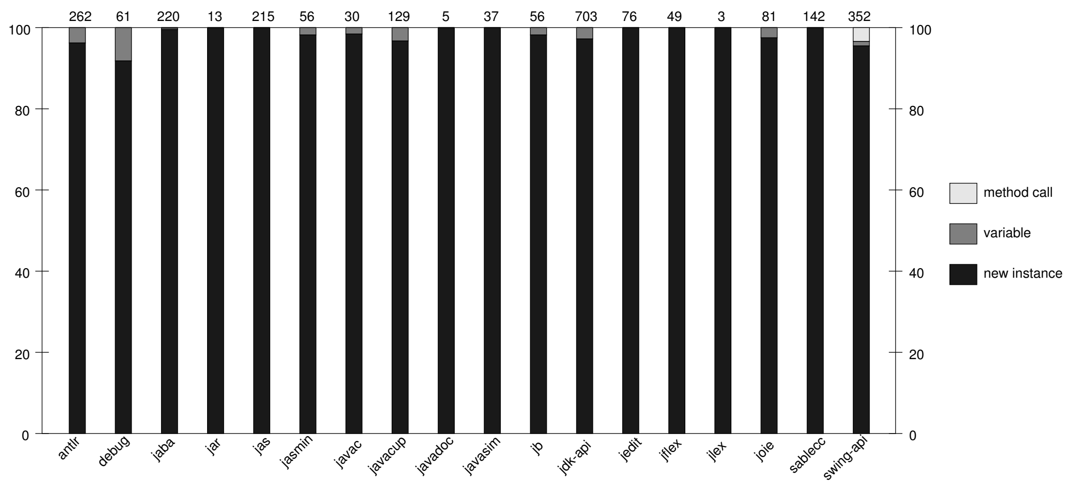


Figure 12: Types of expressions of the `throw` statements.

Table 2: Types of `throw` statement expressions.

Subject	`throw` statements	`throw` statement expressions		
		new instance	variable	method call
antlr	262	252	10	0
debug	61	56	5	0
jaba	220	219	1	0
jar	13	13	0	0
jas	215	215	0	0
jasmin	56	55	1	0
javacup	30	29	1	0
javac	129	127	2	0
javadoc	5	5	0	0
javasim	37	37	0	0
jb	56	55	1	0
jdk-api	703	683	20	0
jedit	76	76	0	0
jflex	49	49	0	0
jlex	3	3	0	0
joie	81	79	2	0
sablecc	142	142	0	0
swing-api	352	336	4	12
Total	2490	2431	47	12

four times more frequently than a method-call expression. The remaining `throw` statements, which constitute over 97% of the total `throw` statements, require no type-inference analysis. Figure 12 presents the data as segmented bars.

Because our empirical evidence indicates that `throw` statements that require type inferencing occur rarely, we believe that the use of an exhaustive type-inference algorithm may not be justified. To determine types for `throw` statements whose expressions are not new-instance expressions, we consider four computationally inexpensive approaches. The first approach is a conservative approximation that includes all subtypes of the relevant exception type. For example, to determine the exception types for the `throw` statement in line 40 of the vending-machine program, the conservative-approximation approach identifies all subtypes of class `Exception` as the potential exception types.

The second approach is a local flow-sensitive analysis[10] [32]. The analysis performs a reverse data-flow

[10] A *flow-sensitive* analysis considers the control flow among statements, whereas a *flow-insensitive* analysis ignores the control flow.

Table 3: Effectiveness of the local type-inference algorithm.

Type-inference approach	Number of throw statements with single type	2–10 types	11–20 types	21–30 types	>30 types	Average number of types
Conservative approx.	26	6	3	0	12	13.3
FS analysis	28	6	3	0	10	11.5
FI analysis	29	8	2	8	0	5.6
FS and FI analysis	31	8	1	7	0	4.9

analysis starting at a `throw` statement that raises an exception object dereferenced through a variable, and searches for statements that assign a type to that variable. If the analysis reaches statements that define the types on all paths to the `throw` statement, it precisely identifies the exception types that (statically) reach the `throw` statement. For example, this approach traverses backward on all paths from the `throw` statement in line 40, and precisely determines the types `IS`, `SNA`, and `IA` for that `throw` statement. The analysis traverses backwards only in the method that contains the `throw` statement. On reaching the method boundary (at the method entry, a call node, or a catch node), the analysis uses the conservative approximation approach, and includes in the solution all subtypes of the relevant exception type.

The third approach is a global flow-insensitive analysis that starts with the conservative approximation, and refines that approximation by examining object-creation sites and return types of all library calls. The refined approximation contains only those types that are either instantiated in the program, or returned by a library routine. For example, to determine exception types for the `throw` statement in line 27, the global analysis first approximates `S`, `IS`, and `SNA` as the potential exception types. The analysis then examines object-creation sites and return types of library calls, and eliminates `S` from the type-inference solution. This global flow-insensitive analysis can omit potential exception types from the type-inference solution because a library routine can return an exception object by encapsulating it in a class, and the analysis would fail to detect that exception type.

The final approach is a combination of the local flow-sensitive and the global flow-insensitive analyses. This approach first performs the flow-sensitive analysis, and if that analysis results in a conservative approximation, the approach uses the flow-insensitive analysis to improve the precision of the type inference information.

To evaluate these four approaches, we performed an empirical study. The goal of the study was to compare the precision of the type-inference information computed using the approaches. Using each of the four approaches, we determined the potential exception types for the 47 `throw` statements in the subjects that mention a variable. Table 3 presents the data from the empirical study. For each type-inference approach, the table lists the number of `throw` statements for which the number of inferred exception types fall in various ranges.

The data in the table shows that the conservative approximation computed a single type for 26 `throw` statements, but computed over 30 types for 12 `throw` statements. For two of those 12 `throw` statements, the flow-sensitive analysis succeeded in reducing the number of exception types to one. However, the flow-sensitive analysis did not cause a significant reduction in the inferred types compared to the conservative approximation. The average number of exception types decreased marginally from 13.3, for the conservative approximation, to 11.5, for the flow-sensitive analysis. The flow-insensitive analysis, however, caused a significant reduction in the inferred types. With the flow-insensitive analysis, no `throw` statement had more than 30 exception types. When the flow-insensitive analysis was used in isolation, the average number of exception types was 5.6. This average was slightly higher than the average when the flow-insensitive analysis was used in conjunction with the flow-sensitive analysis.

The results from the study indicate that, in practice, the flow-sensitive analysis may not offer much benefit over the conservative approximation approach. The flow-insensitive analysis improves significantly the precision of the type-inference solution, but as noted, the analysis may omit potential types from the solution. The scarcity of the data points is a threat to the validity of these observations. Further experimentation is required to establish the verity or the fallacy of the observations.

4.2 Control-Dependence Analysis

In Section 3.3, we saw that exception-handling constructs affect the control dependence relations by causing potentially nonreturning call sites (PNRCs), and require the computation of interprocedural control dependence. The interprocedural control-dependence computation proceeds in two phases [8]. Phase 1 identifies PNRCs that are caused by `throw` statements and halt statements, and uses this information to compute partial control dependences. Phase 2 uses partial control dependences to propagate control-dependence information across methods to compute interprocedural control dependence.

4.2.1 Computation of partial control dependences

The first step of the algorithm that computes partial control dependences identifies call sites that are PNRCs. To identify PNRCs, the algorithm computes, for each call site, the set of nodes to which control can return following the call site. A call site, where control returns to only the associated return node, is definitely returning, and has no effect on control dependences. A call site, where control can return to nodes other than the corresponding return node, is a PNRC, and that call site affects the control dependences of statements that follow the call site. For example, the set of nodes to which control can return following the call at node 17a includes nodes 17b, 22, and ex-exit (IA); that call site, therefore, is a PNRC.

The PNRC-identification algorithm identifies PNRCs caused by `throw` and halt statements. Figure 13 provides an overview of the algorithm. The algorithm first initializes, for each method, the exception types that are directly propagated by that method (lines 1–3). If a method contains a halt statement, the algorithm uses a placeholder type for the propagated type. The algorithm then iteratively processes those methods that propagate at least one type, and builds the set of return sites for each call site, until a fixed point is reached (lines 4–14). The algorithm removes a method N from the worklist (line 6), and processes all call sites that call N (lines 7–10). For a call site in method M, using the information about types that are propagated by N, the algorithm determines the nodes in the CFG of M to which control can return following that call site (line 8). If N contains a halt statement, the algorithm adds a super-exit node (explained below) to the set of return sites. The algorithm also augments the set exception types propagated by M by using the types propagated by N (line 9); these types are indirectly propagated by M through the call to N. After processing each call site in M that calls N, if the set of types propagated by M increases from a previous iteration, the algorithm adds M to the worklist, so that all callers of M are reprocessed. The algorithm continues in this manner until no more methods remain to be processed. When the algorithm terminates,

```
algorithm ComputePNRC
input      CG : call multigraph of program P
output     returnSites : set of return sites for each call site
declare    worklist : methods that are processed iteratively
           types(M) : exception types propagated by method M

begin ComputePNRC
  foreach method P
      initialize types(P) with types directly propagated by P
  endfor
  initialize worklist with methods that have nonempty types
  while worklist is not empty
      remove method N from worklist
      foreach call site in method M that calls N
          determine return sites for the call site
          update types(M) with elements of types(N) that are
           indirectly propagated by M
      endfor
      if types(M) changes from a previous iteration
          add M to worklist
      endif
  endwhile
end ComputePNRC
```

Figure 13: Overview of the PNRC-identification algorithm.

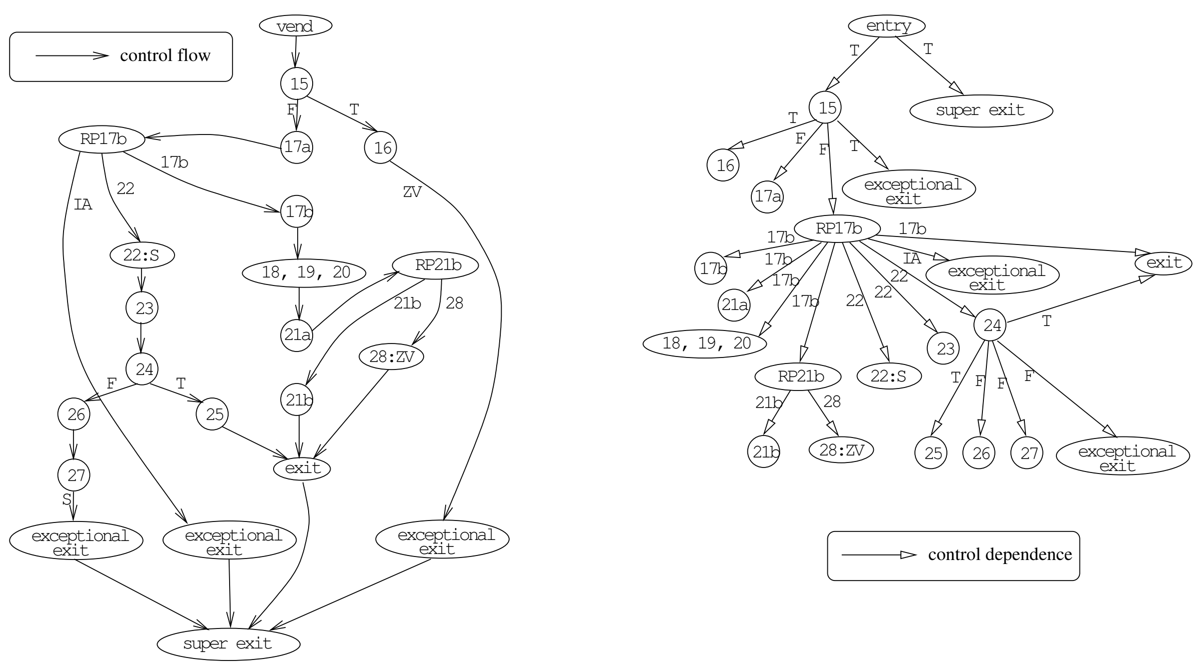


Figure 14: The ACFG for method `vend()` of the vending-machine program (left), and the CDG constructed from the ACFG (right).

it has computed the set of return sites for each call site, and this information is used by the second step of the partial control dependence computation.

The iterative part of the PNRC-identification algorithm is similar to the ICFG-construction algorithm. In building the set of types propagated by a method, the algorithm computes the indirectly propagated exception types, which is the same information that is computed during the ICFG construction. Additionally,

Table 4: Intraprocedural control dependences for **vend()** computed using the CFG, and partial control dependences for the same method computed using the ACFG.

Node	Control dependent on (in CFG)	Control dependent on (in ACFG)	Node	Control dependent on (in CFG)	Control dependent on (in ACFG)
15	entry	entry	23	undefined	(RP17b,22)
16	(15,T)	(15,T)	24	undefined	(RP17b,22)
17a	(15,F)	(15,F)	25	(24,T)	(24,T)
17b	(15,F)	(RP17b,17b)	26	(24,F)	(24,F)
18, 19, 20	(15,F)	(RP17b,17b)	27	(24,F)	(24,F)
21a	(15,F)	(RP17b,17b)	28	undefined	(RP21b,28)
21b	(15,F)	(RP21b,21b)	exit	(15,F)(24,T)	(RP17b,17b)(24,T)
22	undefined	(RP17b,22)			

the PNRC computation also propagates information about halt statements from called methods to calling methods. The call-multigraph-based PNRC algorithm is flow-insensitive, and therefore, can suffer from imprecision in the presence of statically unreachable code. A more precise, and therefore computationally more expensive, version of the algorithm, that is based on the ICFG, identifies and removes statically unreachable code before performing the PNRC analysis. In practice, however, we do not expect the imprecision caused by statically unreachable code to be significant.

After computing the set of return sites for each call site, Phase 1 of the control-dependence computation constructs an augmented control-flow graph that summarizes the effects of external control dependences on statements in a method. An *augmented control-flow graph* (ACFG) for a method M is a control-flow graph, augmented with placeholder nodes that represent predicates in other methods on which statements in M are control dependent. For each PNRC in M, the ACFG contains a unique conditional node, *return predicate*, that acts as a placeholder for the predicates on which return from the called method is control dependent. A return-predicate node has an edge (labeled with the target of the edge) to each node that appears in the set of potential return sites that was computed for the corresponding PNRC. An ACFG also contains a unique node, *super exit*, that represents all exits from M.

Figure 14 shows the ACFG for method **vend()**. The ACFG contains two return-predicate nodes, RP17b and RP21b, because the call sites in statements 17 and 21 are PNRCs. Node RP17b has three out edges, one to each node that appears in the set of return sites for call node 17a; the edge to the exceptional-exit node is labeled by the type of that exceptional-exit node. Node RP21b, likewise, has an out edge to each node that appears in the set of return sites for call node 21a.

Partial control dependences are the intraprocedural control dependences that are computed using the ACFG. The graph on the right in Figure 14 shows the CDG that is constructed using the ACFG for method **vend()**. The partial control dependences for nodes that are control dependent on predicates in called methods contain a return-predicate node. For example, the partial control dependence of node 17b includes return-predicate node RP17b because node 17b is control dependent on the predicate in statement 39 of method **dispense()**.

Table 4 lists the intraprocedural control dependences and the partial control dependences for the nodes in the CFG for **vend()**. The comparison shows that the control dependences computed using the ACFG differ from those computed using the CFG for nine nodes, whereas they are the same for the remaining six nodes. Those six nodes — nodes 15, 16, 17a, 25, 26, and 27 — are not reachable from the PNRCs in **vend()**, and therefore, are unaffected by the PNRCs. The conditions that control the execution of the corresponding statements do not change because of the PNRCs.

```
algorithm ComputeInterCD
input     ICFG : interprocedural control-flow graph of program P
output    interCD : interprocedural control dependences for P
declare   worklist : nodes that are relevant for propagation
          rlist(N) : nodes from which control dependences are
          propagated to node N
          CD(N) : control dependences of node N
          pred(N) : N's predecessor in the ACFG
begin ComputeInterCD
  foreach node N in ICFG
     if N's partial CD includes an entry or a return predicate
        initialize rlist(N)
        if N is the source or sink of an interprocedural edge
           add N to worklist
        else add N to adjust list
        endif
     endif
  endfor
  while worklist is not empty
     remove node N from worklist
     case N is sink of interprocedural edge (with source M)
        if M is CD on entry and N is not the entry node
           CD(N) = CD(N) ∪ CD(pred(N))
        else
           CD(N) = CD(N) ∪ CD(M)
        endif
        if N's CD changes from a previous iteration
           foreach node R such that N ∈ rlist(R) and R
           is the source or sink of an interprocedural edge
              add R to worklist
           endfor
        endif
     case N is source of interprocedural edge (with sink M)
        foreach node R ∈ rlist(N)
           CD(N) = CD(N) ∪ CD(R)
        endfor
        if N's CD changes from a previous iteration
           add M to worklist
        endif
     endcase
  endwhile
  update CD of nodes in adjust list
end ComputeInterCD
```

Figure 15: Overview of the algorithm for computing interprocedural control dependences.

Partial control dependences have several useful applications: they can be used for computing slices [25], for computing procedure-level control dependences, and for computing interprocedural control dependences [8].

4.2.2 Propagation of interprocedural control dependences

The partial control dependences contain correct control dependences for all nodes that are control dependent on non-placeholder nodes. However, the control dependences of nodes that are control dependent on entry or placeholder (return-predicate) nodes must be adjusted. Phase 2 of the control-dependence computation performs this adjustment by propagating control dependences across methods, and computes interprocedural control dependences.

Figure 15 presents an overview of the algorithm that computes interprocedural control dependences from partial control dependences. The algorithm propagates control dependences along interprocedural (call, return, and exceptional-return) edges in the ICFG, until a fixed point is reached. During the fixed-point computation, the algorithm processes only nodes, such as call, entry, return, exit, and catch, that are either sources or targets of interprocedural edges in the ICFG. The algorithm first identifies nodes that require their control dependences to be adjusted (lines 1–9). For each such node N, the algorithm stores, in *rlist*, the set of nodes from which control dependences are propagated to N (line 3). The algorithm places on the worklist those nodes that require their control dependences to be adjusted and are sources or sinks of interprocedural edges (line 5). For example, node 21a is a call node that is control dependent on a return predicate in the ACFG (Figure 14). Therefore, the algorithm adds node 21a to the worklist, and adds node 17b to *rlist*(21a), so that control dependences are propagated to node 21a from node 17b. If a node is not the source or sink of an interprocedural edge, the algorithm places the node on an adjust list, to be processed after the fixed-point computation (line 6). For example, the algorithm adds nodes 18, 19, and 20 to the adjust list; the algorithm also adds node 17b to the *rlist* for these nodes.

Next, the algorithm processes the nodes on the worklist, until the worklist becomes empty (lines 10–31). If a node N is the sink of an interprocedural edge, with source M, the algorithm updates N's control dependences by adding to it the control dependences of M (line 16). For example, when the algorithm processes node 17b, it propagates the control dependence (39, F) from the exit node of the CFG of **`dispense()`** to node 17b. If N's control dependences change from a previous iteration, the algorithm adds to the worklist those nodes whose control dependences may require to be updated (lines 18–22). For example, after updating the control dependences of node 17b, the algorithm adds node 21b to the worklist.

While processing N, the algorithm ensures that control dependences are not propagated along illegal call–return sequences (lines 13–14).

If a node N is the source of an interprocedural edge, with sink M, the algorithm updates N's control dependences by adding to it the control dependences of each node that appears in *rlist*(N) (lines 24–26). For example, the algorithm updates the control dependences of node 21b by adding to it the control dependences of node 17b. If the control dependences of N change from a previous iteration, the algorithm adds M to the worklist because M's control dependences now need to be updated (lines 27–29). Thus, after updating the control dependences of node 21b, the algorithm adds the entry node of **`returnCoins()`** to the worklist.

The algorithm continues in this manner until the worklist becomes empty. Next, algorithm updates the control dependences of the nodes that appear on the adjust list (line 32). The algorithm does this by adding to the control dependences of each node N the control dependences of nodes that appear in *rlist*(N).

Table 5: Interprocedural control dependences for **`vend()`**.

Node	Control dependent on	Node	Control dependent on
15	entry	23	(40,SNA)(40,IS)
16	(15,T)	24	(40,SNA)(40,IS)
17a	(15,F)	25	(24,T)
17b	(39,F)	26	(24,F)
18, 19, 20	(39,F)	27	(24,F)
21a	(39,F)	28	(10,T)
21b	(10,F)	exit	(39,F)(24,T)
22	(40,SNA)(40,IS)		

Table 5 lists the interprocedural control dependences for nodes in the CFG of method **`vend()`**. As the table illustrates, the control dependences of nodes that are control dependent on placeholder nodes in the

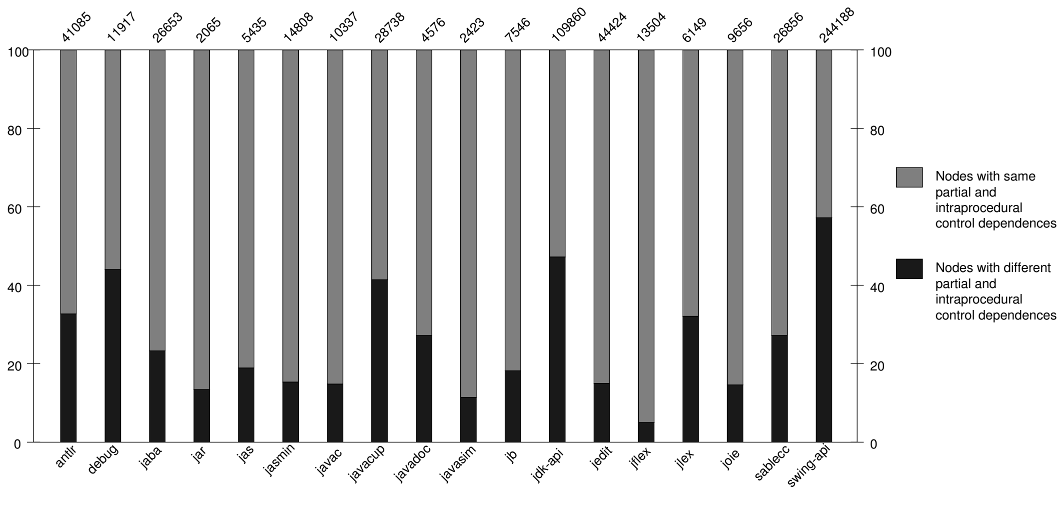


Figure 16: Effects of exception-handling constructs on partial control dependences.

partial control dependences (Table 4) is adjusted by the computation of interprocedural control dependences. For example, node 18 is control dependent on the placeholder node RP17b in the partial control dependences, and on node 39 in the interprocedural control dependences.

4.2.3 Effect of exceptions on partial control dependences

To determine the extent to which the presence of exception-handling constructs affect control dependences, we conducted a preliminary empirical study. The goal of the study was to examine how the presence of exception-handling constructs in our subjects causes the partial control dependences to differ from the intraprocedural control dependences. For each subject, we constructed the CFGs and the ACFGs for the methods in that subject using JABA. To factor out the effects of halt statements on control dependences, we replaced each halt statement[11] with a no-op. We then used the analysis tools from Aristotle Analysis System to construct two CDGs for each method, one using the CFG for that method and the other using the ACFG for that method. Finally, for each node in the CFG (excluding non-statement nodes such as entry and exit), we determined whether that node had different control dependences in the two CDGs.

Figure 16 presents the results of the study. It shows, for each subject, the percentage of nodes that have the same partial and intraprocedural control dependences, and those that have different partial and intraprocedural control dependences. The number at the top of each bar represents the number of nodes in the CFG of the corresponding subject. The figure illustrates that the control dependences of a significant number of the nodes was affected. On average, the control dependences of over 41% of the nodes were affected by the presence of exception-handling constructs. The percentage of affected nodes ranged from 5.0%, for `jflex`, to 57.2%, for `swing-api`.

These results are preliminary in that they do not indicate the actual differences in the control dependences; further empirical studies are required to determine such differences. Further experimentation is also required to study the effects of the differences in control dependences on other analysis techniques, such as slicing, that use the control dependences.

[11] The library call `System.exit()` is the halt statement in Java.

5 Other Analyses and Applications

Control-flow and control-dependence analyses are useful for software-engineering and maintenance tasks, such as slicing and structural testing. The representations and analyses described in the previous section can be applied to perform slicing and testing of programs that contain exception-handling constructs.

5.1 Program Slicing

Program slicing is a technique for identifying transitive control and data dependences in a program. A *backward slice* for a program P, computed with respect to a *slicing criterion* $< s, V >$, where s is a program point and V is a set of program variables referenced at s, includes statements in P that may influence the values of the variables in V at s [27]. A slice can also be computed in the forward direction; a forward slice includes those statements in P that are influenced by the values of the variables in V at s.

There are two alternative approaches to computing slices that either propagate solutions of data-flow equations using a control-flow representation [27, 25] or perform graph reachability on dependence graphs [26]. The slicing algorithms presented in References [27] and [26] make the limiting assumption that, at each call site, control definitely returns from the called procedure, and therefore, consider only intraprocedural control dependences while computing the slices. When applied to programs that contain control structures, such as halt statements and exception-handling constructs, those techniques fail to include those statements in the slices that are related to the slicing criterion through the effects of the control structures on control dependence. Reference [25] extended the slicing algorithm of Reference [27] to use partial control dependences during the computation of slices; that extension correctly accounts for the effects of halt statements, on control dependence, while computing the slices. Using our control-flow representations, and with minor modifications, that extension can be adapted to compute slices that also account for the effects of exception-handling constructs. In recent work [20], we have also extended the alternative slicing technique — one that uses dependence graphs to compute slices — to account for the effects of exception-handling constructs on control dependence.

5.2 Structural Testing

Structural testing techniques [35] develop test cases to cover various structural elements of a program. Control-flow-based structural testing criteria select test cases based on the flow of control in a program. For example, branch testing [36] develops test cases by considering inputs that cause certain branches in the program under test to be executed. Similarly, path testing [37, 38] develops test cases that execute certain paths in the program. Data-flow-based structural testing criteria use the data-flow relationships to guide the selection of test cases [39, 28, 40, 41]. For example the all-uses criterion [41] requires that each du-pair in the program under test be covered by test cases.

Exception-handling constructs introduce new structural elements, such as exceptional control-flow paths, that should be considered for coverage by structural testing techniques. Existing tools for developing structural tests for Java programs [42] provide simple coverage criteria, such as the coverage of `throw` statements and `catch` handlers. Such criteria require the coverage of statements that raise exceptions and those that catch exceptions, and are similar in nature to the traditional criteria that require the coverage of statements or branches. Previous work has shown that criteria, such as branch testing, have weak fault-detection capabilities [43, 44]. We therefore expect the all-throw and all-catch criteria to also be weak in detecting faults. The criteria do not require the testing of various exceptional control-flow paths; they do not consider the different types of exceptions that can be raised at a statement, or the complex control and data interactions both within and across modules that can result in the presence of exception-handling constructs. There are simple types of faults, such as a missing handler, that may not be detected by these criteria. For example, consider a faulty version of the vending-machine program that is missing the `catch` handler in line 54. That handler catches exceptions of type `IA` that are raised by the `throw` statement in line 40. To detect this fault, a test case must cause that `throw` statement to raise an exception of type `IA`. However, the all-throw criterion simply requires that the `throw` statement be covered, and does not consider the types of exceptions. Therefore, a test case might cover that statement but raise an exception of type other than `IA`; a test suite developed in such a manner satisfies the all-throw criterion but fails to detect the fault.

In recent work [45], we have developed a class of exception testing criteria to adequately test the behavior of exception-handling constructs. These criteria subsume[12] the all-throw and all-catch criteria, and test exception-handling constructs with varying degrees of thoroughness. For example, some of the criteria examine activations and deactivations of exception objects, and require the coverage of various paths between the activations and deactivations. It is possible that in practice exception handling may be used ways that the coverage of `throw` statements and `catch` handler suffices for testing the programs. However, it is still beneficial to have a hierarchy of testing criteria that offer the testers flexibility in the level of testing that they perform. Furthermore, by exploring the different types of interactions caused by exception handling, the criteria provide a better understanding of the types of interactions that are significant. Such insight is valuable not only to provide automated support to generate appropriate test cases, but also to verify the interactions informally through inspection. Our current work includes theoretical and empirical evaluations of the exception testing criteria.

6 Safety, Precision, and Practical Utility of the Techniques

Program-analysis techniques often deal with intractable problems whose correct solutions either have a prohibitive expense associated with their computation or are uncomputable. Faced with such impediments, different approaches to performing the analyses compute solutions that are approximations to the true solutions. Such approximations lend to an evaluation of the approaches in terms of the relative safety and precision of the solutions computed by the approaches. A *safe solution* is one that omits no necessary element from the solution whereas a *precise solution* is one that includes no spurious element in the solution. Although increase in safety and precision increase the usefulness of a solution, in practice, the benefits of a safer and more precise analysis must be weighed against the cost of performing the additional analysis. Both safety and precision involve tradeoffs with the efficiency of the technique, and different approaches sacrifice either depending on the level of precision and safety that is desired in the application of the solutions. For certain applications, such as compiler optimizations, safety is required, to avoid invalid program transformations, whereas, for other applications, such as reverse engineering, safety is desirable but not strictly necessary [46].

Our approach to the analysis of exception-handling constructs suffers both unsafety and imprecision. Our

[12] A criterion *subsumes* another if any test case that satisfies the first criterion also satisfies the second criterion.

approach is unsafe because it ignores the control flow caused by implicit exceptions. Implicit exceptions are raised either in library routines or by the runtime environment. For the implicit exceptions that are raised in library routines, we can create summarized CFGs for those library methods that propagate exceptions, and add them to the ICFG. The summarized CFG for a library method would contain nodes for only the entry, the exit, and the exceptional exits.

For the implicit exceptions that are raised by the runtime environment, however, representing each potential control flow with explicit control-flow edges may cause the control-flow representation to become too unwieldy to be useful. Moreover, considering the effects of such implicit exceptions on program-analysis techniques may cause the techniques to generate solutions that are too large to be useful. For example, if a statement s can raise runtime exceptions, a statement that follows s is control dependent on s because s determines whether that statement executes. If statements that raise runtime exceptions occur very frequently, their effects would cause the control-dependence relation to be too cumbersome to be useful. On the other hand, ignoring such implicit exceptions, causes the control-dependence analysis to miss dependences, some of which may be significant.

In future work, we will investigate how the analysis of implicit exceptions influences software-maintenance tasks and practical utility of program-analysis tools. Depending on the particular application and the cost of analysis, we may be willing to accept the unsafety, or we may be able to summarize implicit exceptions and consider their effects on analysis techniques differently than the effects of explicit exceptions.

The four type-inference approaches that we described offer different levels of precision and safety in the information that they generate. The most precise of the four approaches is the one that combines flow-sensitive and flow-insensitive analyses, but that approach can still include unnecessary types in the type-inference solution. The approaches that use the flow-insensitive analysis can omit potential exception types from the type-inference solution, and therefore, are unsafe. The unsafety and imprecision of these approaches causes missing paths and infeasible paths, respectively, in the control-flow representations. Such effects on control-flow representations influence applications that use the representations. For example, the presence of infeasible paths causes a structural testing criteria to generate test requirements that are satisfied by no input to the program. Missing paths cause a structural testing criteria to fail to test certain relationships in a program, and therefore, inadequately test the program.

7 Related Work

Choi and colleagues [12] describe an intraprocedural control-flow representation called the factored control-flow graph (FCFG) to analyze efficiently programs written in languages, such as Java, that may have frequently occurring exceptional control flow. The FCFG represents exceptional control flow caused by both explicit and implicit exceptions. For explicit exceptions, the approach creates edges that are similar to the edges created in our approach. For implicit exceptions, however, the approach does not create edges from each potentially exception-throwing instruction (PEI) because such instructions occur very frequently. Instead, the approach merges several such instructions in the same basic block, and creates factored control-flow edges from the basic block to `catch` handlers to summarize the exceptional control flow for that basic block; the approach creates one factored edge for each type of implicit exception that can be raised by the statements in a basic block. The approach derives the target of the implicit exceptional exits from each PEI in a basic block on demand. Choi and colleagues also describe modifications to data-flow analysis techniques, such as reaching-definition and live-variable analysis, that allow the techniques to work correctly on the FCFG. That work differs from ours in several ways. First, the work does not model the propagation of exceptions across methods. Although Choi and colleagues discuss alternative representations for interprocedural control flow, their current tool does not construct interprocedural representations. Second, the work does not describe the behavior of, and representations for, `finally` blocks. Third, the work does not discuss issues relating to inferring exception types, and how they affect precision of the FCFG and the analyses performed on the FCFG. Finally, the scope of the work is limited to data-flow analysis, and it does not consider the effects of exceptions on control dependence, slicing, and structural testing.

Chatterjee and Ryder [13] describe an approach to performing points-to analysis that incorporates exceptional control flow in languages, such a Java. Their approach derives the exceptional control flow during the points-to analysis, and does not represent it explicitly in an interprocedural control-flow graph. Their approach does not consider implicit exceptions. In subsequent work [14], Chatterjee and Ryder provide an algorithm for computing du-pairs that arise because of exception variables, and along exceptional control-flow paths. In this work, however, they ignore the control flow within `finally` blocks. Chatterjee and Ryder do not describe representations for exceptional control flow, and the scope of their work is limited to points-to and data-flow analysis.

Schaefer and Bundy [16] analyze the flow of exceptions in Ada programs, and extract information that describes how exceptions are propagated across modules. They define several relations that let them formally specify the set of exceptions propagated by different blocks of code. The goal of their analysis is to identify potential violations in the code of application-specific guidelines that govern the usage of exception handling. Robillard and Murphy [15] have similar goals for Java programs. They describe a tool that extracts the flow of exceptions in a Java program, and generates views of the exception structure. These views enable a developer to reason about the flow of exceptions across modules, and identify program points where exceptions are caught unintentionally, or where finer-grained exception handling may be possible. The tool extracts potential implicit exceptions by examining module interface and documentation. The techniques described by both Schafer and Bundy [16] and Robillard and Murphy [15] omit reporting several common implicit exceptions because including them can generate too much information, which adversely affects the usability of their tools. Their techniques are primarily intended for program understanding and detection of inconsistencies in coding. They therefore, do not consider the effects of exceptions on various program-analysis techniques and testing. Using our control-flow representations, we can generate information that is similar to the information generated by their techniques.

Melski and Reps [47] present techniques for interprocedural path profiling, and briefly discuss how path profiles for interprocedural exceptional control flow may be generated. Their work neither describes representations for exceptional control flow, nor analyzes the effects of exceptional control flow on program-analysis techniques.

Other researchers have addressed the problem of computing accurate slices for programs that contain arbitrary intraprocedural control flow [21, 7, 22]. Such control flow is caused by intraprocedural `goto` statements and statements such as `break` and `continue`. Because statements, such as `break` and `continue`, neither control other statements nor use data values, they are never included in a slice. References [21, 7, 22] present solutions in which the statements are included in the slices, when necessary. The same problem can occur in the presence of exception-handling constructs: statements, such as `throw` and `catch` can be excluded from slices. Our slicing technique for exception-handling constructs [20] ensures that `throw` and `catch` statements are included in the slices, when necessary.

Ryder and colleagues [11] conducted a study of the usage patterns of exception-handling constructs in Java programs. They studied a suite of thirty-one Java programs, which contained from two to 2096 methods. They examined 10161 methods, and found that, on average, 16% of the methods contained either a `throw` statement or a `try` statement. Our subjects contain four of the subjects that were included in their study. For those four subjects, our results are consistent with theirs. Their study thus offers further evidence to support our belief that exception-handling constructs are used frequently in Java programs.

8 Conclusions

We have discussed the effects of exception-handling constructs on analysis techniques such as control flow, data flow, and control dependence. We have presented techniques to create intraprocedural and interprocedural representations for Java programs that contain exception-handling constructs. These representations are useful for performing other analyses and constructing other representations. The representations show explicitly the exception types that can be raised at `throw` statements, and exception types that are propagated across methods. Therefore, the representations can provide a valuable aid in understanding the behavior of exception-handling code. We have also presented algorithms for computing control dependences in the presence of exception-handling constructs.

We have presented the results of three empirical studies that we performed using JABA, our analysis tool for Java programs. In the first empirical study, we determined the frequency with which exception-handling constructs occur in Java programs. The results from that study indicate that, in practice, exception-handling constructs can occur frequently: 8.1% of the 30400 methods that we examined contained either a `throw` statement or a `try` statement (Table 1).

In the second empirical study, we evaluated the need for, and approaches to performing, type inferencing for determining exception types at `throw` statements. Based on the results from these studies, we made several observations:

- Type inferencing to determine exception types at `throw` statements may not be required for a majority of the `throw` statements. In over 97% of the `throw` statements in our subjects, the exception object is instantiated at the `throw` statement (Table 2).

- A `throw` statement that does not instantiate the exception object is more likely to raise an exception that is referenced by a variable than an exception that is returned by a method call. In our subjects, four out of every five `throw` statements that do not mention a new-instance expression mention a variable (Table 2).

- The conservative approximation for determining exception types worked well for over half of the `throw` statements for which it was used, but generated very imprecise results for a quarter of the `throw` statements (Table 3).

- In cases where a `throw` statement mentions a variable, the exception object is rarely instantiated in the method that contains the statement. Therefore, the local flow-sensitive type-inference analysis failed to provide any significant improvement in the precision of the type-inference information (Table 3).

These observations provide insight into the usage patterns of exception-handling constructs in Java programs. They can help guide the development of a practical approach to analyze exception-handling constructs, and improve the techniques that we have developed.

In the third empirical study, we evaluated the effects of exceptions on control-dependence analysis. The results of that study indicate that the control dependences of a significant number of statements are affected by the presence of exception-handling constructs (Figure 16). Control dependences that are computed for such statements by traditional techniques can omit necessary dependences and include unnecessary dependences. Incorrect control dependences affect the computation of program slices. Further experimentation with control-dependence computation and program slicing will reveal the extent to which the presence of exception-handling constructs affect these techniques.

We have also discussed how our representations and analyses can be used for other applications such as program slicing and structural testing. Finally, we have evaluated our approach for analyzing exception-handling constructs in terms of the safety and the precision of the approach. Our approach ignores the exceptional control flow caused by implicit exceptions. In future work, we will investigate the effects of implicit exceptions on analysis techniques, and ways to perform the analysis of implicit exceptions. We will evaluate empirically the efficiency of our techniques for constructing control-flow representations for exception-handling constructs, and the trade-offs among the different type-inference approaches. We will also conduct further empirical studies to evaluate the effects of exception-handling constructs on control-dependence computation, program slicing, and structural testing.

Acknowledgments

This work was supported in part by NSF under NYI Award CCR-9696157 and ESS Award CCR-9707792 to Ohio State University. Jim Jones led the design and implementation of JABA, which we used for all empirical studies reported in this paper. He also helped with the empirical studies and made many useful suggestions that improved the presentation of the work. Chaitanya Kodeboyina contributed to the development of the intraprocedural control-flow analysis technique and representation. Gerald Baumgartner, Donglin Liang, and Gregg Rothermel made many helpful comments on previous versions of the paper. The anonymous reviewers provided useful feedback that improved the presentation of the work.

References

[1] P. Frankl and E. J. Weyuker, "An applicable family of data flow testing criteria," *IEEE Trans. on Softw. Eng.*, vol. 14, no. 10, pp. 1483–1498, Oct. 1988.

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

[3] B. Korel, "Automated software test data generation," *IEEE Trans. on Softw. Eng.*, vol. 16, no. 8, pp. 870–879, Aug. 1990.

[4] G. Rothermel and M. J. Harrold, "A safe, efficient regression test selection technique," *ACM Trans. on Softw. Eng.and Meth.*, vol. 6, no. 2, pp. 173–210, Apr. 1997.

[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.*, vol. 9, no. 3, pp. 319–349, July 1987.

[6] 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.*, June 1992, pp. 235–248.

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

[8] 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*, Mar. 1998, pp. 11–20.

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

[10] S. Sinha and M. J. Harrold, "Analysis of programs that contain exception-handling constructs," in *Proc. of Int'l Conf. on Softw. Maint.*, Nov. 1998, pp. 348–357.

[11] B. G. Ryder, D. Smith, U. Kremer, M. Gordon, and N. Shah, "A static study of Java exceptions using JSEP," Tech. Rep. DCS-TR-403, Rutgers University, Nov. 1999.

[12] J.-D. Choi, D. Grove, M. Hind, and V. Sarkar, "Efficient and precise modeling of exceptions for analysis of Java programs," in *Proceedings of PASTE '99 ACM SIGPLAN–SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, September 1999, pp. 21–31.

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

[14] R Chatterjee and B. G. Ryder, "Data-flow-based testing of object-oriented libraries," Tech. Rep. DCS-TR-382, Rutgers University, Mar. 1999.

[15] M. P. Robillard and G. C. Murphy, "Analyzing exception flow in Java programs," in *Proc. of ESEC/FSE '99 Seventh European Softw. Eng. Conf. and Seventh ACM SIGSOFT Symp. on the Found. of Softw. Eng.* September 1999, vol. 1687 of *Lecture Notes in Computer Science*, pp. 322–337, Springer-Verlag.

[16] C. F. Schaefer and G. N. Bundy, "Static analysis of exception handling in Ada," *Software—Practice and Experience*, vol. 23, no. 10, pp. 1157–1174, Oct. 1993.

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

[18] D. Liang and M. J. Harrold, "Slicing objects using system dependence graphs," in *Proc. of Int'l Conf. on Softw. Maint.*, Nov. 1998, pp. 358–367.

[19] M. J. Harrold and G. Rothermel, "Aristotle: A system for research on and development of program-analysis-based tools," Tech. Rep. OSU-CISRC-3/97-TR17, The Ohio State University, Mar. 1997.

[20] S. Sinha, M. J. Harrold, and G. Rothermel, "System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow," in *Proc. of the 21st Int'l Conf. on Softw. Eng.*, May 1999, pp. 432–441.

[21] T. Ball and S. Horwitz, "Slicing programs with arbitrary control flow," in *Proc. of 1st Int'l Workshop on Automated and Algorithmic Debugging.* Nov. 1993, vol. 749 of *Lec. Notes in Computer Science*, pp. 206–222, Springer-Verlag.

[22] J-D. Choi and J. Ferrante, "Static slicing in the presence of goto statements," *ACM Trans. on Prog. Lang. and Sys.*, vol. 16, no. 4, pp. 1097–1113, July 1994.

[23] D. Kung, N. Suchak, J. Gao, P. Hsia, Y. Toyoshima, and C. Chen, "On object state testing," in *Proc. of COMPSAC '94*, 1994.

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

[25] M. J. Harrold and Ning Ci, "Reuse-driven interprocedural slicing," in *Proceedings of the International Conference on Software Engineering*, April 1998.

[26] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Trans. on Prog. Lang. and Sys.*, vol. 12, no. 1, pp. 26–60, Jan. 1990.

[27] M. Weiser, "Program slicing," *IEEE Trans. on Softw. Eng.*, vol. 10, no. 4, pp. 352–357, July 1984.

[28] J. W. Laski and B. Korel, "A data flow oriented program testing strategy," *IEEE Trans. on Softw. Eng.*, , no. 3, pp. 347–354, May 1983.

[29] G. Bilardi and K. Pingali, "A framework for generalized control dependence," in *Proc. of SIGPLAN'96 Conf. on Prog. Lang. Design and Implem.*, May 1996, pp. 291–300.

[30] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F.K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. on Prog. Lang. and Sys.*, vol. 13, no. 4, pp. 450–90, Oct. 1991.

[31] J.P. Loyall and S.A. Mathisen, "Using dependence analysis to support the software maintenance process," in *Proc. of the Conf. on Softw. Maint.*, Sept. 1993, pp. 282–91.

[32] S. Sinha and M. J. Harrold, "Control-flow analysis of programs with exception-handling constructs," Tech. Rep. OSU-CISRC-7/98-TR25, The Ohio State University, 1998.

[33] J. Palsberg and M. Schwartzbach, "Object-oriented type inference," in *Proc. of Object-Oriented Prog. Sys., Lang. and Appl.*, Oct. 1991, pp. 146–161.

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

[35] S. Ntafos, "A comparison of some structural testing strategies," *IEEE Transaction on Software Engineering*, vol. 14, no. 6, pp. 868–874, June 1988.

[36] C. Huang, J., "An approach to program testing," *ACM Computing Surveys*, vol. 7, no. 3, pp. 114–128, Sep. 1975.

[37] W. E. Howden, "Methodology for the generation of program test data," *IEEE Trans. on Computers*, vol. C-24, no. 5, pp. 554–559, May 1975.

[38] J. McCabe, T., "A complexity measure," *IEEE Transaction on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, Dec. 1976.

[39] M. J. Harrold and M. L. Soffa, "Interprocedural data flow testing," in *Proc. of the Third Symp. on Softw. Testing, Analysis, and Verification*, Dec. 1989, pp. 158–167.

[40] S. Ntafos, "On required elements testing," *IEEE Trans. on Softw. Eng.*, vol. SE-10, no. 6, pp. 795–803, Nov. 1984.

[41] S. Rapps and E. J. Weyuker, "Selecting software test data using data flow information," *IEEE Trans. on Softw. Eng.*, , no. 4, pp. 367–375, Apr. 1985.

[42] Sun Microsystems, *JavaScope User's Guide*, Aug. 1998, www.sun.com/suntest/products/JavaScope.

[43] P. G. Frankl and S. N. Weiss, "An experimental comparison of the effectiveness of branch testing and data flow testing," *IEEE Transactions on Software Engineering*, vol. 19, no. 8, pp. 774–787, August 1993.

[44] P. G. Frankl and E. J. Weyuker, "Provable improvements on branch testing," *IEEE Transactions on Software Engineering*, vol. 19, no. 10, pp. 962–975, October 1993.

[45] S. Sinha and M. J. Harrold, "Criteria for testing exception-handling constructs in Java programs," in *Proc. of the Int'l Conf. on Softw. Maint.*, September 1999, pp. 265–274.

[46] G. C. Murphy and D. Notkin, "Lightweight lexical source model extraction," *ACM Trans. on Prog. Lang. and Sys.*, vol. 5, no. 3, pp. 262–292, July 1996.

[47] D. Melski and T. Reps, "Interprocedural path profiling," in *Proceedings of 8th International Conference on Compiler Construction*, March 1999.